Espen Aune Sande

Investigating Monocular Depth Estimation for UAV Navigation and Localization

Master's thesis in MTTK
Supervisor: Sigmund Johannes Ljosvoll Rolfsjord
Co-supervisor: Kristin Ytterstad Pettersen
June 2023

Master's thesis

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

NTNU
Norwegian University of
Science and Technology

Espen Aune Sande

Investigating Monocular Depth Estimation for UAV Navigation and Localization

Master's thesis in MTTK
Supervisor: Sigmund Johannes Ljosvoll Rolfsjord
Co-supervisor: Kristin Ytterstad Pettersen
June 2023

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

Norwegian University of
Science and Technology

Faculty of Information Technology and Electrical Engineering

TTK4900

Master's Thesis:
Investigating Monocular Depth Estimation for UAV Navigation and Localization

Author:
Espen Aune Sande

Spring, 2023

Problem description

Lightweight UAVs have limited capacity for both high precision navigation hardware and camera gimbals, making accurate georeferencing of both observations and the UAV itself extremely challenging. Video depth estimation has the potential to solve a range of challenges within navigation, localization, obstacle-avoidance and mapping of the environment. Traditional methods for visual localization fail for many applications, such as when navigating in non-urban environments which have less distinctly textured terrain.

Deep learning has demonstrated impressive results within self-supervised monocular depth estimation (MDE) for self-driving cars. Are similar results possible to achieve for more challenging UAV drone footage of rural environments, and are they accurate enough for navigation and localization through point cloud registration? While other methods for navigation and localization have had large amounts of time and resources to improve, it is interesting to start investigating this novel approach and see how mature modern AI MDE models and ICP algorithms are for the task.

Abstract

In recent years, Unmanned Aerial Vehicles (UAVs) have become increasingly prominent for private, military, and industrial purposes, particularly in the areas of inspection and observation. To do this, it is required that the drone knows its position, as well as the location of observed objects. The objective of this thesis is to investigate an alternate method of drone navigation and localization without the use of GNSS.

Generally, drones require the help of satellite signals for navigation, but certain applications have higher accuracy requirements that cannot be met without the use of additional expensive RTK equipment. Furthermore, reliance on GNSS makes drones susceptible to interference from malicious actors who can disrupt satellite signals. These problems emphasize the necessity for localization methods that are dependable and accurate without external support. While visual localization methods have demonstrated high precision within localization, they face challenges in areas with few distinctive landmarks. Additionally, there are also high demands of the age, lighting, viewing angles, and season of the database to localize within, which poses a challenge for their use. To address these issues, we propose a purely 3D-model based method of localization, which does not rely on feature descriptors or even color. This method could not only enable accurate navigation but also facilitate effective and precise geolocalization of objects captured by the camera.

The method we propose involves employing monocular depth estimation coupled with point cloud registration to situate the drone within a 3D-model of the area accurately. Given the extensive research on deep-learning depth estimation in recent

years, there may be a vast untapped potential for drone navigation and geo-localization of drone observations. As we see very little research focusing on this use case for MDE, we want to start investigating their effectiveness.

Improving on an effective self-supervised MDE model, we embed and train multiple state-of-the-art depth estimation networks on our own UAV dataset consisting of forested environments. Choosing the strongest performer, we attempt ICP point cloud matching, both for estimating drone motion as well as localization within the 3D-model of the area. Our results show the challenges of depth estimation from UAVs in non-urban environments, as well as problems of localization within imperfect database representations of an area.

Sammendrag

I løpet av siste årene har ubemannede luftfartøyer (UAVer) blitt stadig mer tatt i bruk for private, militære og industrielle formål, spesielt innen inspeksjon og observasjon. For å kunne gjøre dette kreves det at dronen kjenner egen posisjon, samt plasseringen av observerte objekter. Målet med denne masteroppgaven er å undersøke en alternativ metode for dronenavigasjon og lokalisering uten bruk av GPS.

Droner bruker vanligvis satelitter for navigasjon, men av og til har man høyere krav til nøyaktighet som ikke kan oppfylles uten å bruke dyrt RTK-utstyr. I tillegg er avhengighet av GPS en risiko, siden ondsinnede aktører kan utnytte satelittjamming for å stoppe dronene. Disse problemene nødvendiggjør lokaliseringsmetoder som er pålitelige og nøyaktige uten GPS. Mens visuelle lokaliseringsmetoder kan gi presise lokaliseringsestimater, har de problemer i områder med få særegne landemerker. I tillegg er det også høye krav til databasen som brukes, for eksempel innen alder, belysning, årstider, visningsvinkel, som kan være arbeidskrevende å oppfylle. For å løse disse problemene foreslår vi en rent 3D-modell-basert metode for lokalisering, som ikke er avhengig av bilde-features eller farger. Denne metoden vil ikke bare kunne gi nøyaktig navigasjon og lokalisering dersom det virker, men også muliggjøre effektiv og presis geolokalisering av observerte områder.

Metoden vi foreslår innebærer å bruke monokulær dybdeestimering kombinert med punktskyregistrering for å plassere dronen i en 3D-modell av området. Gitt de store framskrittene innen dyp-læring for dybdeestimering de siste årene så virker det som om vi har et enormt potensial for å bruke dette innen drone-navigasjon og

geolokalisering. Siden vi ser svært lite forskning som fokuserer på denne bruken av KI-basert dybdeestimering, ønsker vi å begynne å undersøke effektiviteten til denne metoden.

For å forbedre en effektiv selvledet dybdeestimering-modell, legger vi inn og trener flere toppmoderne dybdeestimeringsnettverk med vårt UAV-datasett. Etter vi velger ut den beste modellen, prøver vi ICP-punktskymatching, både for å estimere dronebevegelser så vel som lokalisering innen 3D-modellen av området. Resultatene våre viser utfordringene med dybdeestimering fra UAV-er, samt problemer med lokalisering innenfor utdaterte og detaljfattige databaserepresentasjoner av et område.

Contents

Problem description **ii**

Abstract **iii**

Sammendrag **v**

Preface **x**

1 Introduction **1**

1.1 Motivation and Related Work 1
1.2 Background 5
1.2.1 Neural Network Basics 6
1.2.2 Monocular depth estimation (MDE) 7
1.2.3 Convolutional Neural Networks (CNN) 11
1.2.4 Transformers 13
1.2.5 Vision Transformer 16
1.2.6 Point cloud registration (PCR) 17
1.2.7 Pinhole camera model and the photometric loss 20
1.3 Contributions 23
1.4 Outline 24

2 Model Overview **25**
2.1 DynaDepth . . . 26
2.1.1 Model Information . . . 26
2.1.2 Implementation and Training Details . . . 30
2.2 ManyDepth . . . 31
2.2.1 Model Information . . . 31
2.2.2 Implementation and Training Details . . . 34
2.3 MiDaS and the Dense Prediction Transformer . . . 35
2.3.1 Model Information . . . 35
2.3.2 Implementation and Training Details . . . 38
2.4 MonoViT . . . 39
2.4.1 Model Information . . . 39
2.4.2 Implementation and Training Details . . . 41

3 UAV-dataset **43**

4 Monocular Depth Estimation Results and Discussion **47**
4.1 Problems and Improvements to DynaDepth . . . 47
4.2 Depth evaluation on the test-set . . . 51
4.3 Observations and Discussion . . . 55
4.3.1 Depth Errors . . . 55
4.3.2 Auxiliary Network Estimates . . . 59
4.3.3 ManyDepth . . . 64
4.3.4 MiDaS . . . 66
4.3.5 Incorporation of GPS pose supervision . . . 66
4.3.6 Effects of self-supervision . . . 69
4.3.7 Training Data . . . 70
4.4 Chosen Model for PCR . . . 72

5 Point Cloud Registration **73**
5.1 Setup . . . 73
5.2 Estimating ego-motion . . . 80

5.3 Localization 84
5.4 Overall results 88

6 Conclusion **91**

References **93**

Preface

This master's thesis is submitted as a part of the requirements for the master degree at the Department of Engineering Cybernetics at the Norwegian University of Science and Technology. The work presented in this thesis has been carried out under the supervision of Prof. Kristin Y. Pettersen and researcher Sigmund Rolfsjord at the Defence Research Institute (FFI).

This master's thesis is a continuation of a specialization project I conducted during the latter half of 2022[49], also in collaboration with the Defence Research Institute (FFI). As is customary, the specialization project is not published. This means that important background theory and methods from the project report will be restated in full throughout this report to provide the best reading experience. Below, a complete list of the material included from the specialization project is listed.

- Section 1.2 with some changes, except for subsection 1.2.5 which is new
- Section 2.1 and 2.3, with new additions and some cuts.

This project largely builds upon the depth estimator model DynaDepth. I would like to thank the authors Zhang et al. [65] for publishing their work and source-code so that I could utilize it for this project. I would also like to thank the Defence Research Institute for capturing the data I use for training.

Unless otherwise stated, all figures and illustrations have been created by the author.

Espen Aune Sande
Trondheim, spring 2023

Chapter 1

Introduction

Drones are gradually becoming more and more prevalent in today's society, both for the average hobbyist as well as industry professionals. A vital part of a drone's operation is the navigation system responsible for finding its current position, which is the focus of this Master's Thesis. We propose a method involving deep learning depth estimation and point cloud registration. This chapter outlines the motivation and theoretical background, and gives a more detailed explanation of our method.

1.1 Motivation and Related Work

Today, drones usually rely on GNSS satellite signals as the key component of their navigation. This generally works well for many forms of use, though there are some instances where standard satellite-based navigation would not be sufficiently accurate, requiring the additional aid of expensive RTK equipment. Furthermore, this dependency is a drawback in situations where GNSS is not available, as well as representing a significant vulnerability that malicious actors can exploit: Jamming satellite signals can be an effective way to disable drones.

To circumvent this, there is a need for robust, cheap, and accurate localization methods

to replace the role of GNSS, and this is the motivation for this thesis. A way of solving this is to utilize *Visual Localization*: Using camera-observations to place yourself within a database representation of an area. If you are able to spot an object and know its position relative to yourself, it will theoretically let you place yourself very accurately in the database, as well as geolocalize observed areas.

This project focuses on a variant of Visual Localization, though our approach is different from the traditional well-researched methods. Visual localization methods can for instance involve image-retrieval within a large database, or sparse point matching between extracted keypoints in the image and a 3D area-model, or other variants [6]. Finding distinct features for generating good point-correspondences can be a bottleneck in these visual localization pipelines, especially in monotonous terrain. Figure 1.1 shows an example of matched point correspondences between two images using SIFT, where the 30 strongest matches are included. We see that most of these 30 are far away from the camera, and there are even some point correspondences in the sky. The areas closest to the camera are the clearest and most detailed, and one would hope that this is where the strongest correspondences could be found, though they are instead further away in sections with less detail.

If we only want to perform ego-motion estimation on sequential images, such as the two images in the figure, we may use RANSAC to perform spatial verification of the matched points to filter out erroneous correspondences and calculate relative pose. However, in a different scenario, such as when performing absolute localization with image retrieval, we are liable to experience geometric bursts [50], i.e. erroneous matches with different scenes that pass spatial verification, because trees are generally very similar. This problem can also occur when feature-matching with a 3D model of the environment. Using the standard methods of localization, much work needs to be done when constructing the database, for example needing images of the same object from multiple angles[40]. Also, if the database is old or captured during different seasons, localization using feature-matching is more prone to fail as the appearance of the scene changes. Figure 1.2 shown an example of this happening, where the right

image is during autumn and of a slightly different position and angle. While it may be easy to match between two images taken seconds apart, as done in figure 1.1, it can be much more difficult if the images are years apart, or if the database image is of a significantly different angle or a different season[69]. The risk of creating false

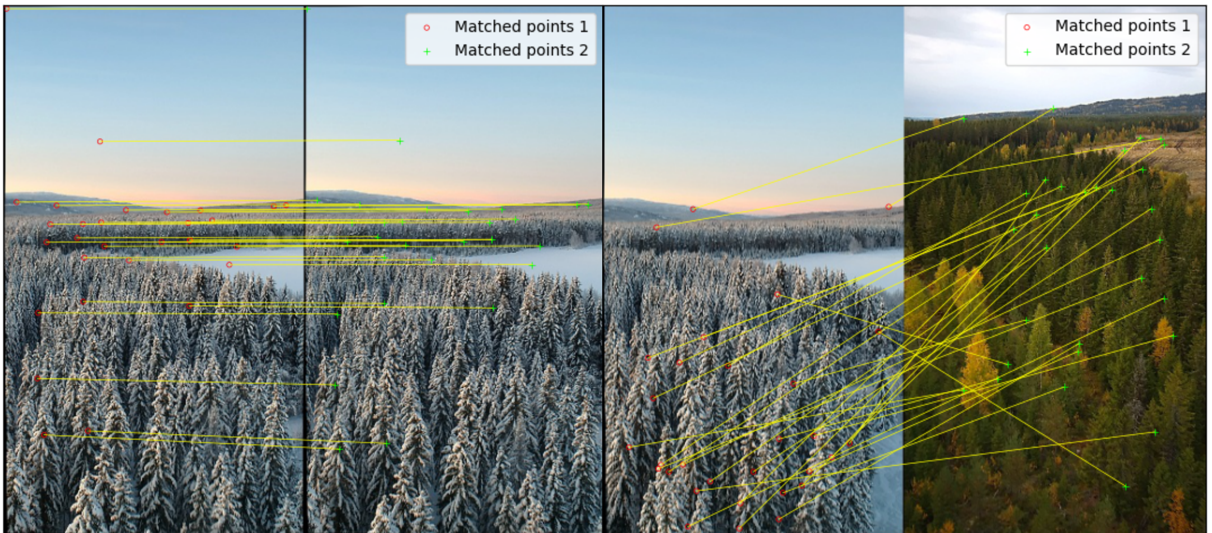


Figure 1.1: SIFT matching between two images of our dataset, where only the 30 strongest matches are shown. While some matches are erroneous, most seem correct.

Figure 1.2: SIFT matching between two pictures only a few meters apart but captured during different seasons and at different angles, showing the challenge of using feature-matching.

matches in monotonous terrain, as well as the work required to construct a good database, is a large motivation for our approach. Instead of focusing on a set of point-correspondences for the purpose of localization, we could attempt to match the entire 3D-structure of the images.

We see a potential for a good solution utilizing *Monocular Depth Estimation*(MDE), which should not suffer from the same drawbacks of many visual localization methods. While it is common for visual localization models to use both local features and a

3D-model of the scene [67], our method will only involve 3D matching through point cloud registration:

- Given an image of the area, generate a depth estimate of the scene
- Produce a 3D point cloud of the observed area using the estimated depth at each pixel and the camera-parameters
- Using the 3D-model database, generate a point cloud from a known position and match it with the MDE point cloud, giving a localization estimate.

This matching between the area-model and the point-clouds generated through the depth estimates will make no use of local image descriptors nor even colors, and the hope is that this will make it robust in areas with few distinct landmarks where traditional methods with feature detectors and descriptors might struggle. The database we will use for localization is a six year old 3D-model captured from an airplane passing above. This means that the model is built from only a single-angle viewing and is likely heavily outdated, which would generally not be sufficient for generating databases for other forms of visual localization[69].

While monocular depth estimation and point cloud registration are well researched fields of study, the combination we are attempting has not seen much work. The usage of MDE from UAVs is not the most prevalent, but there has been work done, such as for collision avoidance[66]. Autonomous vehicle research is one of the largest fields where MDE is being developed, and many AI depth estimation models are designed for those datasets. One such dataset, KITTI, is commonly used for training and benchmarking MDE models The specialization project I conducted during the latter half of 2022[49] revealed that UAV-datasets pose a much more challenging depth-estimation problem than KITTI, though it has been shown that these models can indeed be used as a base for further developments to achieve good results from drones [36]. Fortunately, there are also other models available which are not primarily designed for autonomous cars, such as MiDaS [44] and M4Depth [15]. Since MDE models are achieving increasingly impressive results on their respective datasets, we want to attempt to apply these to

our drone data which could yield large advancements within drone navigation and geolocalization.

Point cloud registration for the purpose of navigation is a well-researched use-case[43], though it is typically used with laser-scanners instead of MDE, and more often with ground vehicles instead of UAVs. The *Iterative Closest Point* (ICP) algorithm has long been a staple for solving the registration problem, and we will utilize it in this thesis.

Despite this being a very active research field, with decades of research regarding SIFT, SLAM, image-retrieval, etc., this approach to the visual localization problem in drones is not much explored. To our knowledge, there are no published works relating to it. There seems to be much untapped potential for this fundamentally different approach given the advancements within monocular depth estimation, and this makes it interesting to investigate if the models and methods are mature enough for localization and navigation for UAVs.

1.2 Background

To ensure the reader can understand the problem we are attempting to solve, as well as the techniques and considerations needed, we give background information pertaining to the themes and theory utilized.

Since the thesis in large part consists of AI depth estimation, we briefly touch on artificial neural networks before giving an outline of the process of monocular depth estimation using deep learning, as well as introducing important neural network architectures that form the foundation of the MDE models we use. Our plan is to use self-supervision to train the neural networks. As such, we provide a detailed explanation of the main self-supervised loss function we use: the photometric loss.
The localization step of our proposed method consists of point cloud registration, so we introduce the general Iterative Closest Point algorithm.

1.2.1 Neural Network Basics

In recent years, the field of machine learning has been increasingly dominated by *Artificial Neural Networks*. A neural network consists of several layers of nodes with connections between the nodes in each layer, and its purpose is to generate some meaningful output by finding structures and relations from the input. An example could be the estimated price of a house based on the size, number of rooms, location, etc. The parameters of a neural network are the weights and biases. The output of a node is determined by the weighted sum of its inputs in addition to its bias, which is then optionally passed through an activation function (e.g Sigmoid). Activation functions are necessary if the goal is to create a non-linear relation, otherwise we end up with a purely linear function. The weights and biases are updated based on how well accurate the network is in its prediction, which is measured by a loss function we wish to minimize. The training of the network consists of using gradient descent of the loss function, where gradients are calculated by a method called *backpropagation*. We differentiate between three different types of layers in a neural network: the input layer, the output layer, and the hidden layers. Input and output layers represent the input data and outputted results respectively, while the hidden layers are in between them and are not ”seen” from the outside. Further information on the design and usage of neural networks can be found in the book by Szeliski [57], as well as several free resources online.

Explanation of commonly used neural network terms the reader should be familiar with:

- *Fully Connected Neural Network*(FCNN): All nodes share a connection with all nodes in the next layer.
- *Feedforward Neural Networks*(FNN): All connections flow in the same direction; no loops.
- *Multilayer Perceptron*(MLP): Somewhat ambiguous term in the litterature. Generally means a fully connected, feedforward neural network with at least one

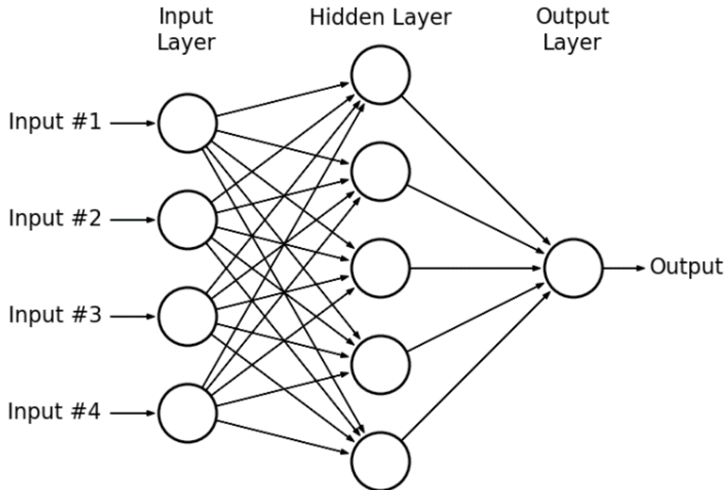


Figure 1.3: Example of an MLP with a single hidden layer. It also fulfills the criteria of an FNN and an FCNN. Figure from Hassan et al. [22].

hidden layer. Figure 1.3 shows an example of a simple MLP.

- *Linear Layer*: Neural network with only an input and output layer, without activation functions.

1.2.2 Monocular depth estimation (MDE)

In order to infer the 3D-geometry of a scene captured by a video, we need a way to estimate depth at each of the pixels in the pictures. If we have multiple images of the same scene, such as sequential images or images from different cameras, we can apply *Structure from Motion (SfM)* [57, chapter 11]. First, you extract features and match them between images to generate point correspondences, for example using SIFT [33]. Then you apply SFM to calculate the 3D points as well as the relative pose between the two images, though without knowing the true scale. The problem of scale ambiguity is illustrated in figure 1.4. If you know the distance between the two frames

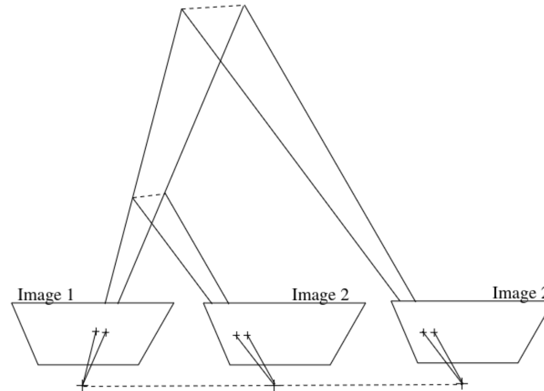


Figure 1.4: Similar to Monocular Depth Estimation, SfM suffers from scale ambiguity. There are an infinite number of different configurations that fit the observations in the two images. Figure by Andreff et al. [1].

(often referred to as the baseline), you can find the true scale of the depth. However, there are constraints on how accurate the depth estimates are based on the baseline between the images. Longer baselines are required for high accuracy at larger depths [52].

If the area has few distinct textures we might want to avoid having to match between two images and instead try to infer the depth from a single image, which is known as *Monocular Depth Estimation* (MDE). While this is a challenging task, the advances of deep neural networks and continuously improving computational capabilities are making such methods increasingly effective at estimating depth [37]. Monocular depth estimation is difficult because there are an infinite number of 3D scenes that give the same 2D-image (is that object far away and large, or close and small?), and because you need to take into account global context for the entire image [38]. While the SfM approach uses the scene shift between two images as the depth cue (also called ”disparity”), a deep neural network could be able to learn several cues, such as [58]:

- **Texture gradient**: Objects further away have less detailed surfaces.
- **Linear Perspective**: Parallel lines converge at the horizon at large distances.

- **Shades and Shadows**: Objects casting shadows on others can reveal which ones are close and which ones are far away.

The first step of any MDE model is to extract information from the image, commonly referred to as ”features”. Older methods of monocular depth estimation could involve using hand-crafted feature extractors [51], though they have been rendered obsolete by deep convolutional neural networks that have been shown to have superior results [4]. In addition to CNN feature extractors, there are also non-CNN types, such as the *Swin Transformer*[31], which is a transformer[59] based feature extractor.
The next step of an MDE-model is to utilize the features in some way that yields an estimate of the depth. There are many different ways to achieve this, and examples include using several layers of additional convolutions and upsampling that yield a per-pixel depth estimate (like U-Net [46], figure 1.5), or more sophisticated methods such as utilizing Conditional Random Fields [64]. Obviously, there are many additional tactics you could employ in your MDE-model, and we will see some of them later in section 2.

While we generally talk about estimating depth, models often estimate inverse depth instead. This has the benefit of being more easily able to represent large depth values, such as one will get when the sky is included in the image. While a neural network that directly estimates depth would need to output a value up to infinity, estimating inverse depth allows you to represent it as zero instead. Additionally, inverse depth is often used in the field of localization and navigation because it has better statistical properties [9]. Borrowing from the field of stereo-vision, the term "disparity" is often used instead of inverse depth [18]. While disparity technically refers to the scene shift caused by motion between two images, it is inversely proportional to depth[20], which is why it is sometimes used in articles regarding MDE.

There are two different ways to view the estimation problem: Either per-pixel regression, or per-pixel classification by discretizing the depth into several levels. Regression is the most common method, since it is able to capture depth in finer detail than the discretized case, but can be more difficult to train [16]. Discretization can yield

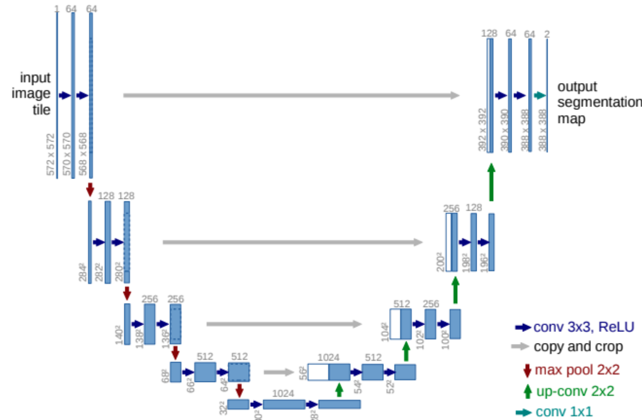


Figure 1.5: The structure of U-Net [46]. Though initially created for biomedical image segmentation, it can be used for monocular depth estimation as demonstrated by Godard et al in their article on Monodepth2 [18]

better results and faster convergence, but the estimated depth image will be more coarse depending on the discretization levels. It is also possible to combine these two approaches [29]. The choice of approach will determine which loss functions you can employ. An example of a loss-function for each is the mean square error for regression and cross-entropy loss for classification, though it is common to have more than one loss function [4]. Loss functions can be a major part of the design of a machine learning model, and we will see later that they can become quite complicated.

Monocular depth estimation is an actively researched field, having uses within for example 3D-modeling, and autonomous driving. One of the main difficulties of monocular depth estimation is getting enough labeled data to train on, and for this reason, many self-supervised methods have been developed. These methods usually employ some sort of 3D-reconstruction to estimate the ground truth required for training or utilize pairs of images in other ways[62]. In addition, other sensors such as the IMU could be used for aid, as we will soon see.

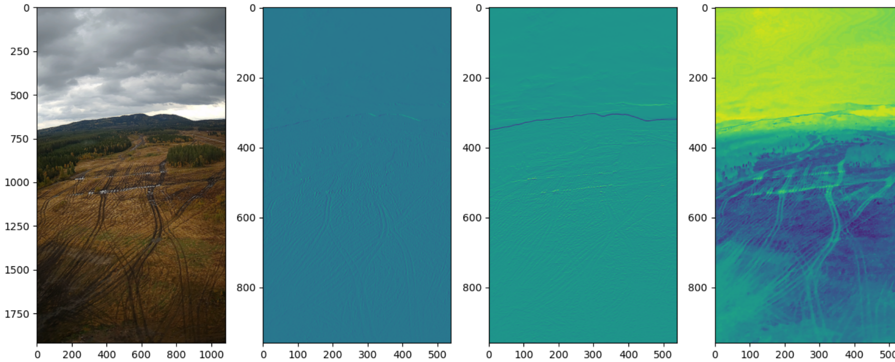


Figure 1.6: Example of the processing of an image from our dataset: Original image on the left, followed by the output of three kernels (index 14, 26, 32) from the first convolutional layer of ResNet18[24] to its right.

1.2.3 Convolutional Neural Networks (CNN)

As mentioned above, *Convolutional Neural Networks* are commonly used as the "backbone", sometimes also called the feature extractor or encoder, in many MDE models. This is the part that first takes in the picture we want to estimate depth for, and consists of several layers of convolutional filters with learnable parameters. The filter consists of several kernels, and we see an example of the output from select kernels in the first convolutional layer of a CNN in figure 1.6. From the figure, we see that it is not always immediately clear what kind of feature a specific kernel is supposed to pick up on, such as the first example (second image from the left), however in the second example we see that it has detected the outline of the mountains in the distance. It is common to use a variant of *ResNet*[24] as the backbone, such as ResNet18, which is used in the figure.

CNNs typically also have other some sort of down-sampling layer, such as max-pooling, between each convolutional layer [57, chapter 5]. This way we can get features of

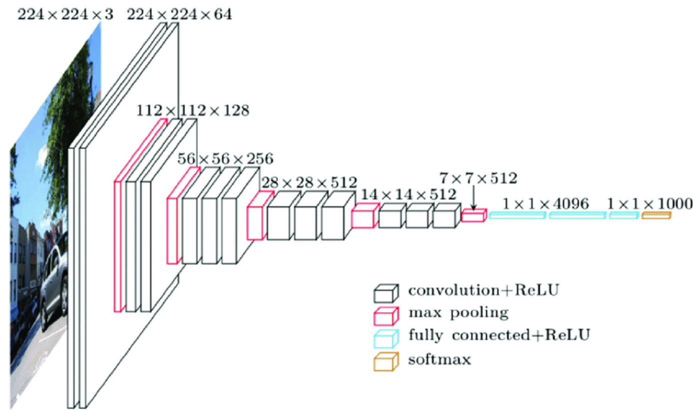


Figure 1.7: The architecture of VGG16, designed by Simonyan and Zisserman [55]. Note that this figure also shows features being fed into a FCNN for the purposes of image classification, whereas we are generally interested in gathering features of multiple resolutions. Figure from [39].

different resolution sizes as the original picture is propagated along the layers of the CNN. As is the case with other neural networks, CNNs use activation functions which are generally placed right after a convolutional layer. An example of a CNN architecture is shown in figure 1.7

The cascading structure of the CNN, where the feature maps go from fine to coarse because of the downsampling, means that the receptive fields of the features gradually increase. Higher resolution features from the early layers contain information about smaller sections of the image, i.e. a smaller receptive field. On the other end, the lower resolution features represent larger parts of the image and have larger receptive fields[34]. For our depth estimation usage, high-resolution features (for example edges and corners) may be useful for estimating the depth of small objects or fine details, while low-resolution features may describe object boundaries and scene layout, which is very important for estimating the depth of larger objects or the overall structure of the image. Despite the CNN features covering multiple receptive fields, the convolution-operation involves the features only interacting with their local neighbors

as they propagate through the CNN. This means they can struggle to capture long-range information in an image and are instead better at utilizing local information [28]. This can be problematic for our depth estimation, where the entire context of the image is important for inferring depth.

1.2.4 Transformers

First proposed by Vaswani et al. [60] for the purposes of *Natural Language Processing* (NLP), the Transformer has since been used for various other tasks within the field of artificial intelligence. The transformer works by utilizing the attention mechanism, the concept of which is that some parts of the data are more important than others, and should be paid more attention to. Attention has three inputs, the query, key, and value. Sometimes the data used with a transformer needs to be converted to a numerical vector-format, such as when working with written sentences. In this case, sentences are transformed into "tokens" and then converted to the correct dimension, d_{model}. We refer to this transformed data as embeddings, which is a representation of the data that is compatible with the transformer model. In addition to this, positional encoding is summed with the embedding and together they form the transformer input. Positional encodings carry information about the position of the inputs, such as the order of words in a sentence in the case of NLP. In the original transformer article, sinusoidal signals based on input dimension and position are used.

The structure of the transformer is shown in figure 1.8. The transformer-layers are repeated N times sequentially, incrementally refining the results. The model has an encoder-decoder structure, where the inputs are encoded, (on the left in 1.8) and then fused together with the previous output of the transformer through cross-attention in the decoder (on the right).

Attention (specifically: scaled dot-product attention) is calculated as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where Q is the query, K is the key, V is the value, and d_k is the dimension of the key. Generally, the input will be split up and processed by multiple attention modules in parallel, known as multi-head attention. In the multi-head attention, the inputs are divided by passing them through several parallel linear neural network layers (i.e. no activation function) that split up the data to the parallel attention modules. These neural networks are in essence learnable linear projections to the specified dimensions of the queries, keys, and values. After concatenating the outputs from the attention-heads, they are passed through a final FNN. Equation 1.1 displays the structure of the Multi-Head Attention module. The weight matrices W_i^Q, W_i^K, W_i^V, W^O represent the FNNs, where the first three are the projection-layers for each attention head and the last one is the output-projection layer.

$$\begin{aligned} \text{MultiHead}(Q, K, V) &= \text{Concat}\left(\text{head}_1, \ldots, \text{head}_\text{h}\right) W^O \\ \text{where } \text{head}_i &= \text{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right) \end{aligned} \tag{1.1}$$

The dimensions of the projection matrices depend on the dimension of the original inputs d_model, and the number of parallel heads h you use, with the relation $d_{k,v} = d_\text{model}/h$.

$$W_i^Q \in \mathbb{R}^{d_\text{model} \times d_k}, W_i^K \in \mathbb{R}^{d_\text{model} \times d_k}, W_i^V \in \mathbb{R}^{d_\text{model} \times d_v}, W^O \in \mathbb{R}^{hd_v \times d_\text{model}}$$

There are two general types of attention: self-attention and cross-attention. These differ in the inputs they get, where a cross-attention module receives its query embeddings from a different source than the key and value. The multi-head self-attention block of the transformer takes in embeddings as key, value, as well as query, while the multi-head cross-attention in the decoder uses the output from the encoder as key and value, but the query comes from the previous layer of the decoder. The cross-attention serves the purpose of letting the transformer use information from past data along with the current input.

Transformers nowadays have expanded further than the original from 2017, and

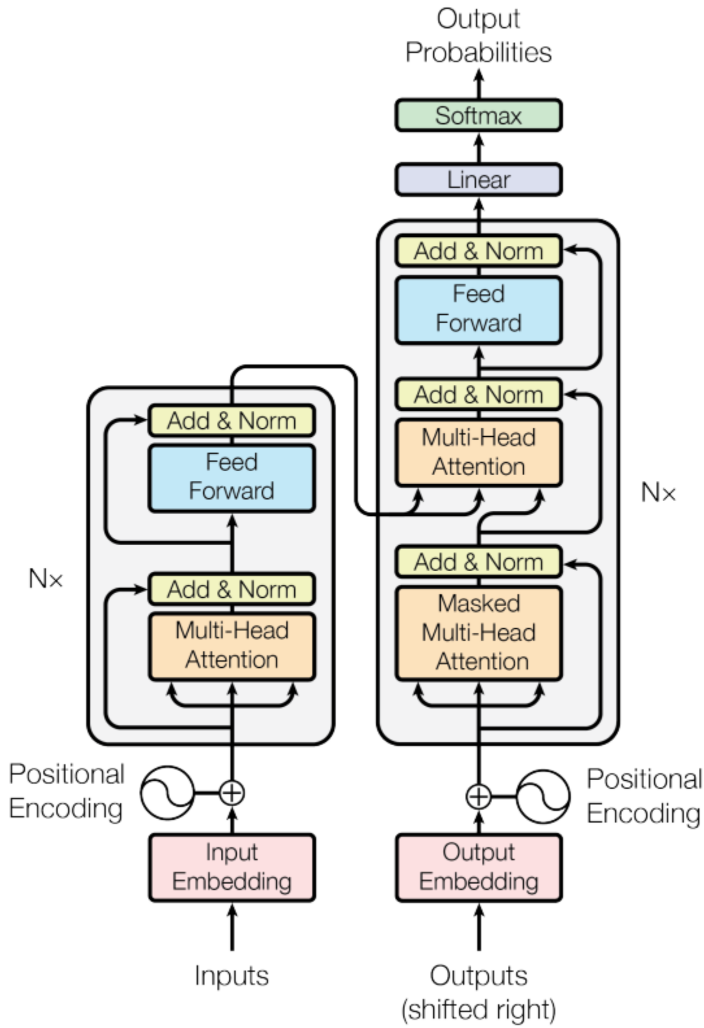


Figure 1.8: The structure of the original transformer as introduced by Vaswani et al. Encoder on the left, decoder to the right. For further detail, we refer to the article "Attention Is All You Need"[60].

we will see some transformer based models that are not of the same structure as the original since they are not applied to NLP. Though they are not identical, they still use a similar combination of attention-layers, feed-forward layers, and normalization & addition layers.

1.2.5 Vision Transformer

An important transformer variant for our use case is the Vision Transformer (ViT) [10], which was designed for the purpose of image classification. In order to apply transformers to vision-related tasks, we need a way to generate embeddings from images. A naive way would be to use each pixel in the image as tokens, though this would be computationally intractable for larger images since the attention mechanism has quadratic complexity with regard to the sequence length (number of tokens) [59]. Instead, Dosovitskiy et al. [10] proposed splitting the image into patches, which are then flattened into vectors. Since the transformer uses a fixed latent vector size, the flattened vectors are projected into the required dimension using a learnable linear projection. The resulting projections are referred to as the image embeddings. Positional embeddings are added to the image emeddings, and together with an additional classification token they are fed into the transformer. The classification token's state at the transformer's output is passed through a classification head to generate the predicted image class. The ViT architecture is intended to be as similar to the original transformer as possible and is shown in figure 1.9.

This Vision Transformer was designed for image classification, but this image-patch transformer framework often sees use as the replacement of the CNN backbone in various computer-vision tasks, including monocular depth estimation[70]. When a vision transformer is applied as the encoder/backbone in this way, the image is sent through the multiple sequential transformer-layers where sets of features can be extracted, similar to how they are extracted from the multiple layers of a CNN[21]. In addition to purely transformer-based computer vision architectures, hybrid models also exist, where both transformers and CNNs are used. For example, convolutional layers can

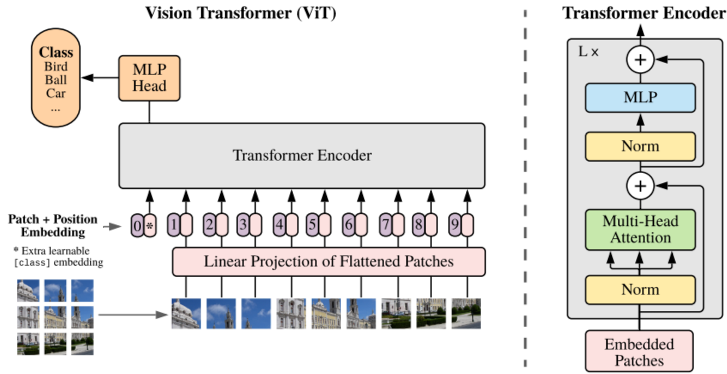


Figure 1.9: The structure of ViT [10]. Since the model was designed for image classification, an additional embedding representing the image class is included in addition to the nine image patches. Contrary to the original transformer model, this one does not have a transformer decoder.

be used to generate features to use as embeddings for the transformers instead of the image-patch method.

An important motivation for including CNNs is their strong capability of modeling local information, whereas the transformer is more liable to not capture local information as well as it models the global context[70]. The attention mechanism of the transformer is not bound by locality the same way a CNN is, since the multi-head attention involves all tokens generated from the image. This means that a transformer-layer has a global receptive field, letting it capture global information with only the use of a single attention layer[28].

1.2.6 Point cloud registration (PCR)

The problem of matching two point clouds by estimating a rigid body transform $(\mathbf{R}, \mathbf{t})$ that aligns the clouds is referred to as *Point Cloud Registration*. A common way to solve this problem is by using the *Iterative Closest Point* (ICP) algorithm, as proposed by Besl

& McKay [3] and Chen & Medioni [8]. This algorithm does not only apply to point clouds, but also for example surfaces and lines. The ICP algorithm has seen widespread use within computer vision and robotics and is being continuously developed to this day [43].

1.2.6.1 Iterative Closest Point

The ICP algorithm can be separated into six stages, and variants of ICP can be classified by which stages they affect [48]:

- Selection of points from each set,
- Matching points between the sets,
- Weighting the pairs,
- Rejecting some pairs
- Assigning error metric,
- Minimizing the error metric

A basic ICP algorithm for aligning point clouds $\boldsymbol{P}$ and $\boldsymbol{Q}$ is as follows:

- Start with initial guess of $(\mathbf{R_0}, \mathbf{t_0})$
- Apply the transform to $\boldsymbol{P}$ and match each of the N points to the closest (euclidean length) in $\boldsymbol{Q}$
- Minimize $\sum_{i=1}^{N} ||\mathbf{R}\mathbf{p_i} + \mathbf{t} - \mathbf{q_i}||^2$ w.r.t $(\mathbf{R}, \mathbf{t})$, where $\mathbf{p_i}$ and $\mathbf{q_i}$ are matched points,
- Set $(\mathbf{R_{next}}, \mathbf{t_{next}}) = (\mathbf{R}, \mathbf{t})$
- Start next iteration or terminate if the error metric is small enough or is stationary, or if the maximum number of iterations is reached.

This version does not do any weighting or rejecting of point pairs, and is computationally expensive for large point clouds since it uses all points. It also may struggle in situations where the point clouds only have partial overlap, because many points will not have a true match and there is no method of rejecting. Despite this, it will likely converge successfully if the initial alignment is good and there is little noise. The minimization step can be solved analytically using *Singular Value Decomposition*(SVD) [56].

There are many point cloud registration libraries available that allow for many different variants of the stages of ICP [25]. Some examples are:

Selection: Select all points, random sampling, or uniform sampling of the 3D-space to capture the structure better. Another method is to perform ICP at multiple resolutions, starting with sampling fewer points from the source cloud $\boldsymbol{P}$ at first and ending with all points at the final level. The intermediate pose estimates $(\mathbf{R}, \mathbf{t})$ are used as starting poses at the next resolution level.

Matching: If the point clouds are from depth images, you could project $\boldsymbol{Q}$ to the image plane where $\boldsymbol{P}$ was generated from, and match based on distance in pixel-space $\|\mathbf{u_p} - \mathbf{u_q}\|$

Rejecting: Reject duplicated matches from $\boldsymbol{P}$ to the same point $\mathbf{q_i}$ in $\boldsymbol{Q}$ and only retain the closest one. Another method is to use RANSAC [14]. Rejections schemes are also called outlier filters, and are vital when the point clouds have errors or only have partial overlap.

Error metric: The example used what is called the point-to-point error metric, but there are also others, such as point-to-plane which estimates the surface normal at $\mathbf{q_i}$ as n_{q_i} and incorporates it into the error: $\sum_{i=1}^{N}((\mathbf{Rp_i} + \mathbf{t} - \mathbf{q_i}) \cdot n_{q_i})^2$
Another variant is Generalized ICP (also known as plane-to-plane [53]), which utilizes surface normals of both point clouds. Figure 1.10 shows a comparison between these three variants. When working with structured environments, such as indoor scans,

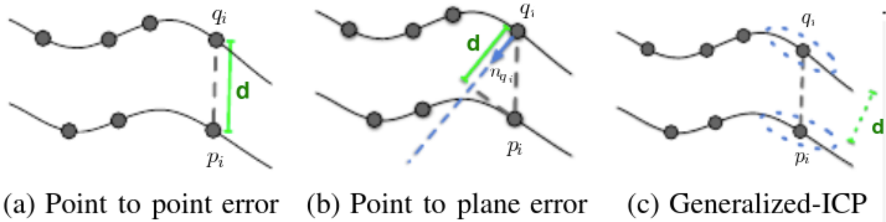


Figure 1.10: Visualization of different error metrics, courtesy of Holz et al [25]

incorporating surface normals into the error metric has been shown to give better results [42].

1.2.7 Pinhole camera model and the photometric loss

Throughout this project, we use the *Pinhole Camera Model* to represent the relationship between 3D points and the corresponding 2D-pixel points in the images. The pipeline for calculating 2D-pixel points from 3D-world coordinates is [57, chapter 2]:

- Transform 3D points from world-frame to the camera-frame,
- project points to the image plane,
- convert to pixel coordinates.

The camera coordinate frame is defined with the z-axis pointing into the image with origo at the camera aperture. The orientation of the x- and y-axis lie along the image axes, i.e. the right-down-front configuration. The important takeaway is that the z-coordinate represents depth. The image plane is defined as the plane at z=1 in the camera frame, and projecting to it consists of dividing a 3D-point by its z-value, giving the *normalized camera coordinate* $\bar{\mathbf{x}}_\mathbf{c}$. The relationship between the normalized camera coordinate and pixel coordinates is defined by the *Intrinsic Camera Matrix* $\mathbf{K}$:

$$\bar{\mathbf{u}} = \mathbf{K}\bar{\mathbf{x}}_\mathbf{c}$$

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Here, u and v are the pixel-coordinates, and x and y are the camera coordinates projected to the image plane.
Another common method of representing the relationship is[23]:

$$\tilde{\mathbf{u}} = \mathbf{K}\mathbf{x_c} = \pi\left(x_c; \mathbf{K}\right)$$

$$\begin{bmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

In this case, $\tilde{\mathbf{u}}$ is the homogeneous pixel coordinate and $\mathbf{x_c}$ is the 3D-coordinate in the camera frame (not normalized). The pixel coordinates u and v can be calculated by dividing the homogeneous pixel coordinate $\tilde{\mathbf{u}}$ by its third component:

$$\bar{\mathbf{u}} = \psi(\tilde{\mathbf{u}}) = \psi(\pi\left(x_c; \mathbf{K}\right))$$

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \frac{1}{\tilde{w}} \begin{bmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{bmatrix}$$

In our case, the **K**-matrix has been given without us needing to perform camera calibration. For futher detail on the intrinsic camera matrix and camera projections, we refer to the book by Szeliski [57].

In the process of projecting to pixel-coordinates, the depth in the image is lost. To be able to return to 3D-points (hereafter called *backprojection*), we need to know the

depth at each pixel. The backprojection is:

$$\mathbf{x_c} = z\mathbf{K}^{-1}\bar{\mathbf{u}} = \pi^{-1}\left(\bar{\mathbf{u}}, z; \mathbf{K}\right) \tag{1.2}$$

where $\mathbf{x_c}$ is the 3D coordinate in camera frame and z is the depth at pixel $\bar{\mathbf{u}}$. Using this equation, we can generate a point cloud from a depth image.

If we have two images A and B of the same scene, it is possible to find the mapping between each pixel in image A and image B by utilizing image-projection and backprojection. The strategy is to backproject the pixels in image A, transform the 3D points from the camera frame of A to the camera frame of B, and project them to pixel-coordinates in image B [19]:

$$\begin{aligned}
\bar{\mathbf{u}}_\mathbf{b} &= \psi\left(\pi\left(\mathbf{T}_{a\to b}\left(\pi^{-1}\left(\bar{\mathbf{u}}_\mathbf{a}, z_a; \mathbf{K}\right)\right); \mathbf{K}\right)\right) \\
\bar{\mathbf{u}}_\mathbf{b} &= \psi\left(\mathbf{K}\mathbf{R}_a^b\mathbf{K}^{-1}\bar{\mathbf{u}}_\mathbf{a} + \frac{\mathbf{K}\mathbf{t}_\mathbf{a}^\mathbf{b}}{z_a}\right) \\
\bar{\mathbf{u}}_\mathbf{b} &= \boldsymbol{w}\left(\bar{\mathbf{u}}_\mathbf{a}, z, \mathbf{R}_a^b, \mathbf{t}_\mathbf{a}^\mathbf{b}, \mathbf{K}\right)
\end{aligned} \tag{1.3}$$

We refer to this mapping as the *warping function*, where $\mathbf{T}_{a\to b}$ and $\left(\mathbf{R}_a^b, \mathbf{t}_a^b\right)$ represent the rigid body transform from camera frame A to B. Here we assume that both images are taken with the same camera, meaning that they have the same intrinsic matrix $\mathbf{K}$. This function is the basis for the *Photometric Loss*, which compares quantities in each pixel in the images to see how similar they are, for example by comparing color [26]. By minimizing the photometric loss, one can recover the relative motion between two images in addition to depth in the scene. It can be formulated as:

$$L_{\text{photo}}^{\text{vis}} = \sum_i \mathcal{L}(I_a\left(\mathbf{u}_i\right), I_b\left(w\left(\mathbf{u}_i, z_i, \mathbf{R}_a^b, \mathbf{t}_\mathbf{a}^\mathbf{b}\right)\right)) \tag{1.4}$$

where $\mathcal{L}$ is a comparison function, such as the L2-norm of the difference. For the photometric loss to work optimally, we have to assume a static scene, no lighting changes,

and that no objects are blocked from view in either image (occlusion). This entails that all corresponding pixels look identical in both images, though these assumptions typically do not fully hold in real-life datasets[54].

Using this loss/error function to estimate motion and 3D geometry is referred to as a *Direct Method*, as opposed to *Indirect Methods* which instead find specific point correspondences and minimize errors between observations and predictions (typically pixel-positions through reprojections)[12]. Direct methods are generally preferred in regions where it is difficult to generate and match keypoints between images, i.e. monotonous, non-distincty textured areas [13].

It should be noted that the photometric loss optimization is an ill-posed problem[7] since it will have many local minimums, as each pixel will have many different pose and depth configurations that fit, similar to scale ambiguity in SfM (figure 1.4). An important aspect of the photometric loss that is very relevant for us, is that if the translation $\mathbf{t}_\mathbf{a}^\mathbf{b}$ is zero, then the depth z_a vanishes from the expression and is irrelevant in the minimization. This means that non-zero translation is vital to be able to infer depth, and larger translations are useful for accurate depth estimation as the depth term becomes more significant in the loss.

1.3 Contributions

The primary contributions of the thesis are:

- Incorporation of advanced AI depth networks into a self-supervised MDE model
- Addition direct supervision to auxiliary networks to improve depth estimates
- Showcasing difficulties of performing depth estimation from drones in challenging non-urban environments and under more demanding drone maneuvering.
- Adding weighting schemes to an off-the-shelf ICP algorithm to test the effectiveness of PCR for localization and navigation

We compare multiple depth networks against each other, all trained on a common self-supervised framework. Seeking to improve the supervision of the self-supervised framework, we apply additional direct supervision of the auxiliary networks with readily available GPS data. Using a large supervised MDE model, we show that leveraging previous training and finetuning it on our own data using self-supervised depth losses can give good results despite suboptimal training data. Overall, we showcase the challenges of MDE from highly maneuvrable UAVs in low-texture environments and contrast our own dataset with a previously successful attempt at depth estimation on the UAVid-dataset[35]. To help localize the drone in our database model as well as find the correct ego-motion, we add some simple weighting schemes to a publically available ICP model, giving an increased likelihood of aligning important landmarks.

1.4 Outline

In chapter 2, we explain the various MDE models we will attempt to use in addition to special considerations we made when implementing them, as well as detailing training methodology.

Then, we motivate several changes we make to the self-supervised MDE-framework we use as the foundation for all the depth models in section 4.1, before the depth estimation results are shown and discussed.

Finally, point cloud matching results are shown and discussed at the end, in chapter 5.

Chapter 2

Model Overview

To be able to accurately localize using PCR, we need the depth estimates to be as accurate as possible. In our attempt to attain good results, we will train and compare four different depth estimation models:

- DynaDepth[65]
- ManyDepth[63]
- MiDaS[44][45]
- MonoViT[68]

The following is an introduction to these four models with a larger focus on the main DynaDepth model, as well as the corresponding training and implementation details. For further information regarding the structure of the respective neural network models, we refer the interested reader to the original articles.

2.1 DynaDepth

2.1.1 Model Information

One of the main challenges of AI computer-vision is that of data gathering. Specifically, data-labeling, which can be a very challenging process depending on what kind of labels are needed. Instead, we turn our gaze toward self-supervised MDE-models so we can gather as much data as possible without worrying about generating depth ground-truths. Ideally, we would want some self-supervised model that can overcome the well-known issue of scale-ambiguity, and this is something that Zhang et al. [65] have strived to achieve. Their work, DynaDepth, is a self-supervised MDE model based on the highly influential Monodepth2 [18], and achieves state-of-the-art accuracy on the KITTI dataset. Improving on Monodepth2, it incorporates IMU-data and an EKF-framework yielding improved results and the ability to resolve the scale-ambiguity of the MDE problem during training. The DynaDepth model consists of several parts:

- A neural network to predict inverse depth from an image at four scales.
- Three neural networks to estimate camera-centric gravity $\mathbf{g_c}$, velocity $\mathbf{v_c}$ and ego-motion $(\tilde{\mathbf{R}}, \tilde{\mathbf{p}})$ from images, denoted $\mathcal{M}_g$, $\mathcal{M}_v$ and $\mathcal{M}_p$ respectively.
- An EKF to fuse the IMU-motion and the camera-based pose estimates to give an updated estimate of the camera ego-motion.

Depth Network

DynaDepth uses a relatively simple neural network for inverse-depth prediction. Using an encoder-decoder framework similar to UNet[46] (figure 1.5) with ResNet[24] as the encoder, it generates predictions at four different scales, with image resolution halved at each scale. This multi-scale depth-network decreases the chance of getting stuck at local minima[18] and means that there will be four sets of depth-related losses.

The estimates are bound between 0 and 1 using a sigmoid function, and in order to convert to depth, a maximum and minimum depth is used. The authors used 0.1m - 100m when training on KITTI, though you will need to supply your own depth-range when training on different data.

Pose, Gravity and Velocity Networks

$\mathcal{M}_g$ takes in a single image and outputs the gravity vector in camera-frame. $\mathcal{M}_v$ takes in two sequential images and predicts the velocity in camera-frame at the first image. $\mathcal{M}_p$ takes in two sequential images and outputs the relative poses between them. Different from other models improving on the Monodepth2 groundwork, the pose network also estimates variances of the 6-DOF motion, which is required for the covariance matrix used in the Kalman filter.

These three networks have a simple structure, using ResNet as the encoder, followed by four stages of convolutions and outputting values of the required dimensions.

IMU and the EKF

The IMU-data consists of acceleration and angular velocity measurements, and are used to estimate the relative drone motion between images. A single training sample consists of three sequential images from the dataset. The random shuffling of the data during training means that it is challenging to continuously carry and update an estimate of the velocity and gravity needed in the IMU motion dynamics. Instead, the gravities and velocities are predicted from the frames in the training sample using $\mathcal{M}_g$ and $\mathcal{M}_v$. The motion between two frames is calculated from the estimated starting velocity and the accumulated IMU data where the estimated gravity is accounted for. Since the acceleration readings also capture the normal-forces counteracting gravity, the gravity vector is needed to correct the IMU acceleration measurements. We will refer to the estimated motion using IMU, $\mathcal{M}_g$, and $\mathcal{M}_v$ as the IMU-motion.

The EKF is used to fuse estimates from the pose-network $\mathcal{M}_p$ and the IMU-motion. The camera-based pose estimate is used as the observation in the Kalman Filter update step, where the network-predicted pose-covariances are also put to use. For information on the camera-centric IMU motion dynamics, as well as the derived EKF-matrices, we refer to the original DynaDepth article [65].

Losses

Intuitively, if you know how much the drone moves between each image, you can

reason about the depth of each pixel by how much it is displaced. This principle is used to create losses that do not require a depth ground truth and is where the photometric loss (1.4) comes in.
The first of the losses presented in the paper, called the *IMU Photometric loss*, is shown in figure 2.1.

$$\begin{aligned} L_{photo}^{IMU} &= \frac{1}{N}\sum_{i=1}^{N} \min_{\delta\in\{-1,1\}} \mathcal{L}\left(\boldsymbol{I}\left(\boldsymbol{y}_i\right), \boldsymbol{I}_\delta\left(w\left(\boldsymbol{y}_i, \tilde{z}_i, \hat{\mathbf{R}}_\delta, \hat{\mathbf{p}}_\delta, \mathbf{K}\right)\right)\right), \\ \mathcal{L}\left(\boldsymbol{I}, \boldsymbol{I}_\delta\right) &= 0.85\frac{1-\mathrm{SSIM}\left(\boldsymbol{I}, \boldsymbol{I}_\delta\right)}{2} + 0.15\left\|\boldsymbol{I}-\boldsymbol{I}_\delta\right\|_1 \end{aligned} \tag{2.1}$$

The δ denotes whether we are comparing to the previous ($\delta = -1$) or next ($\delta = 1$) image in the training sample, which is used for the ”per pixel minimum” trick proposed by Godard et al in [18] which will combat situations where an object might be obscured in either of the two neighboring images. $\mathbf{K}$ is the camera intrinsic matrix, $(\hat{\mathbf{R}}_\delta, \hat{\mathbf{p}}_\delta)$ is the EKF pose estimate between image $\mathbf{I}$ and $\mathbf{I}_\delta$ and $\tilde{z}_i$ is the estimated depth at pixel $\boldsymbol{y}_i$. $\mathcal{L}$ is an image similarity function where SSIM is the structural similarity index[61].

In essence, the IMU photometric loss warps the N pixels in the image to either the previous or next in the sequence depending on which image has the least occlusion for the pixel. This corresponds to $\min_{\delta\in\{-1,1\}}$ in the loss function, because the neighboring image with the lowest photometric error is considered the least occluded. When the correct ego-motion and depth estimate is found, the photometric error will be minimized.

The second loss proposed in the paper is shown in equation 2.2. It is called the *Cross-sensor Photometric Consistency Loss*, and aims to align the EKF and Pose-net estimated ego-motions by equalizing the warpings. As before $(\hat{\mathbf{R}}_\delta, \hat{\mathbf{p}}_\delta)$ is the ego-motion estimate from the EKF, while $(\tilde{\mathbf{R}}_\delta, \tilde{\mathbf{p}}_\delta)$ is the ego-motion estimate from the Pose-net

$\mathcal{M}_p$.

$$L_{\text{photo}}^{\text{IMU-cons}} = \frac{1}{N}\sum_{i=1}^{N}\min_{\delta\in\{-1,1\}}\mathcal{L}\left(\boldsymbol{I}_\delta\left(w\left(\boldsymbol{y}_i,\tilde{z}_i,\tilde{\mathbf{R}}_\delta,\tilde{\mathbf{p}}_\delta,\mathbf{K}\right)\right),\boldsymbol{I}_\delta\left(w\left(\boldsymbol{y}_i,\tilde{z}_i,\hat{\mathbf{R}}_\delta,\hat{\mathbf{p}}_\delta,\mathbf{K}\right)\right)\right) \tag{2.2}$$

This loss gives additional supervisory signals from the IMU to the various networks involved, and together with L_{photo}^{IMU} they serve the purpose of injecting the real scale into the translation and depth estimates.

In addition to these losses, they also employ the disparity smoothness loss L_s as defined by Godard et al. [18], an $L2$-loss for the neural network predicted velocity and gravity L_{vg}, and a vision photometric loss $L_{\text{photo}}^{\text{vis}}$ similar to 2.1 but instead using the Pose-net $\mathcal{M}_p$ predicted ego-motion. The total loss is then:

$$L_{\text{total}} = L_{\text{photo}}^{\text{vis}} + \lambda_1 L_s + \lambda_2 L_{\text{photo}}^{IMU} + \lambda_3 L_{\text{photo}}^{\text{IMU-cons}} + \lambda_4 L_{vg} \tag{2.3}$$

where $\lambda_{1,2,3,4}$ are weights determined empirically to be {0.001, 0.5, 0.01, 0.001}.

While the DynaDepth article is not clear on this, the L_{vg} losses are two separate losses and are computed with the norm of the estimated velocity and gravity, not the vectors. The velocity ground truth is the norm of the GPS-measurement of velocity, and gravity ground truth is simply the gravitational acceleration $g \approx 9.81$.

As mentioned before, achieving the correct depth-scale from monocular images is difficult, since there are an infinite number of 3D-scenes that can create the same 2D-image. However, the authors of DynaDepth have shown that by integrating the IMU-measurements as they have, it is possible to achieve correct scale during training. This is very important for our usage. If the scale of our depth is incorrect, then we would either need to employ non-rigid point cloud registration by including scale as an optimized variable, or manually adjust the scale ourselves. This complicates the process, so we want the scale of the depth estimation to be as accurate as possible. Since the scale is only inferred through the IMU-based losses, the scale is still unknown

during test time, but so long as the test-set is similar to the training set, the scale should hopefully be approximately correct.

DynaDepth inherits an auto-masking feature from Monodepth2 [18], which aims to exclude certain pixels from the loss calculation. It works by setting the weight of pixels that remain sufficiently unchanged between two images to 0 in the loss-functions. This could be the case for the sky, objects moving along with the camera, or situations where the camera is stationary. While the depth-estimation network does not utilize disparity shifts when generating a depth estimate (since it uses single images), the disparity is important for loss-calculation. Even if we will always be able to create a depth estimate, we might say that depth is unobservable for stationary pixels since we cannot generate useful supervisory signals for the depth network with them. This is why masking out such pixels in the loss-function is useful.

2.1.2 Implementation and Training Details

This model has several characteristics we are looking for to be able to use it for our own purposes:

- It is self-supervised.
- It has been shown to give accurate results.
- It incorporates IMU data to infer true scale.

We use this model as the base for our work, as it should hopefully give us scale-aware supervisory signals without needing data-labeling. It will serve as a foundation for testing other depth estimation models in our search for accurate results. During further use, we will maintain the auxiliary networks and the EKF as they are, but we may substitute the depth encoder and/or decoder with alternative models.

For training, we use the Adam optimizer and a batch size of 8, with an initial learning rate of 1e-4 which is decreased by an order of magnitude every 10 epochs. We trained for 30 epochs, as was originally done for the KITTI, but saved the weights regularly

throughout training. Observing the validation losses and depth error metrics, we choose to use the model trained for 16 epochs, and will do the same for the other models. We set the maximum and minimum depth to 5000m and 1m respectively. This ensures the entire depth-range in the training images is able to be represented by the depth network. The only data-augmentation used is color-jittering, where all three images in a single training sample receive the same augmentation.

To reduce VRAM requirements and reduce training-time, we use ResNet18 for the three backbones of $\mathcal{M}_v$, $\mathcal{M}_g$, and $\mathcal{M}_p$, as well as for the depth network. While larger ResNet models might be more capable, the results in the DynaDepth article revealed that the improvement of going from ResNet18 to ResNet50 was negligible. We instead choose the smaller and less complex ResNet18.

2.2 ManyDepth

2.2.1 Model Information

A potential improvement we discussed during the Specialization Project, is finding a way to incorporate multiple images in the depth estimation model. While stereo-methods are not uncommon [4], we only have one camera to work with on our drone, meaning that sequential images from the video-feed are what we can use. This means that the relative pose between each frame has to be estimated in some way, and this is already done for the purpose of calculating photometric losses during training in the DynaDepth model. The authors of ManyDepth, Watson et al. [63], have already developed one such depth estimation model. It builds on the Monodepth2[18] model from 2019, the same as DynaDepth does. Using ManyDepth, we may be able to improve performance by incorporating IMU-data into the depth estimator by fusing the camera-based Pose-net motion with the IMU-motion through the extended Kalman Filter.

ManyDepth shares much of the same structure as DynaDepth, including using photometric losses with the per-pixel minimum trick, predicting depth at four different scales, and more, though no IMU-data is used. This section will instead focus on the depth estimator, which is what we are interested in.

Similar to the photometric loss, the depth estimator module of ManyDepth utilizes the warping function (1.3). However, the warping is performed on the extracted feature maps instead of the images themselves. In addition, the depth is discretized into a set number of layers, and the warping is performed for each discrete depth level. An L_1-error is computed between the warped and non-warped image for each depth level, which are stacked into a cost volume. Intuitively, if a specific depth level d_i is correct for a part of the image, then the error in that part of the volume will be small. Thus, the cost volume says something about the probabilities of which of the depth levels $d_{1,\ldots,n}$ a part of the image belongs to. The cost volume along with the feature map of the image is finally passed into a CNN encoder-decoder which performs per-pixel inverse-depth regression. An overview of this process is shown in figure 2.1.

Before we are able to create the cost volume, the depth levels must be specified. ManyDepth sets the number of discrete depth-levels as a hyperparameter and employs an exponential moving average of the maximum and minimum depths from a single-image depth network (which we will introduce shortly). With a momentum of 0.99, the d_{min} and d_{max} parameters are updated based on the maximum and minimum of the depth estimates averaged over the batch in each training step. The required number of levels are generated linearly over the depth range.

While developing ManyDepth, it was revealed that the cost-volume method of generating dept estimates suffered greatly when observing non-static objects in the scene, much more so than single-image depth estimators do. To remedy this, a single-image depth estimator network is implemented in the model, which should not have such large errors on moving objects. For this use, the original depth estimator from

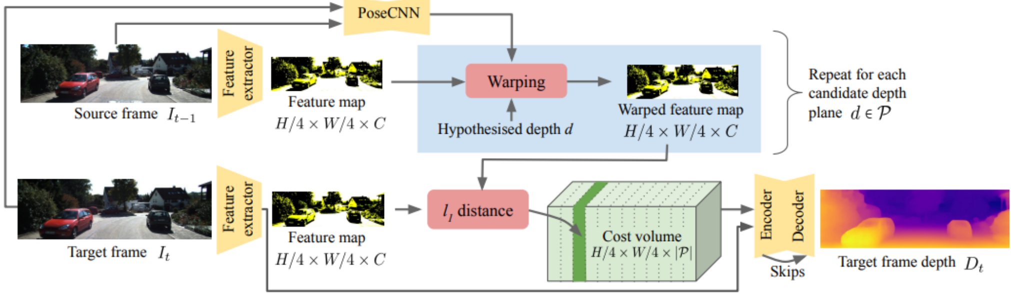


Figure 2.1: Overview of the multi-frame depth estimation model in ManyDepth[63]. The previous and current frames are passed through a CNN feature extractor, and the absolute deviation of the warped and non-warped feature maps is constructed into a cost-volume where each stack represents a depth level in the image. The cost-volume along with the feature map of the current image are passed into an encoder-decoder depth network for per-pixel regression of inverse depth at four different scales.

Monodepth2[18] is utilized, the same as outlined in section 2.1. The purpose of the additional depth map is to give an indication of when this phenomenon is occurring, and to instead make the model learn from the single-image depth estimate in that case. This is done by creating a pixel-mask M, which indicates which regions of the image have sufficiently differing estimates between the two depth estimators. It is defined as:

$$M = \max\left(\frac{D_{\text{cv}} - \hat{D}_t}{\hat{D}_t}, \frac{\hat{D}_t - D_{\text{cv}}}{D_{\text{cv}}}\right) > 1$$

Here, $\hat{D}_t$ is the single-image depth estimate, and D_{cv} is the arg-min of the cost-volume, i.e. the depth value that gives the lowest error when warping the features and not the depth estimate D_t itself. The masked areas are subject to a $L1$-loss, referred to as the *Consistency Loss*:

$$L_{\text{consistency}} = \sum M \left|D_t - \hat{D}_t\right| \tag{2.4}$$

The gradients generated from this loss are blocked from reaching the single-image depth network to ensure that it remains a teacher and the multi-image network is the one learning.

To ensure that the ManyDepth model is able to predict depth both at the start of sequences, and when the vehicle is stationary, the cost volume is with a small probability randomly replaced with all zeroes, forcing it to only rely on information from the current image to estimate depth, and not use the cost volume.

The total loss in ManyDepth is a combination of the photometric loss (1.4), the consistency loss, and the disparity smoothness loss as defined by Godard et al. [18]. As with DynaDepth, the smoothness loss L_s is weighted with $\lambda = 0.001$. Since we have two depth networks, we have two sets of photometric and disparity smoothness losses. The mask M is applied to the photometric loss from the multi-image network such that it only attempts to minimize the photometric loss in pixels that are not covered by the consistency loss $L_{\text{consistency}}$. As such, the full loss of the Manydept model is:

$$L_{\text{total}} = (1 - M)\, L_{\text{photo}}^{\text{vis, multi}} + M L_{\text{consistency}}^{\text{multi}} + L_{\text{photo}}^{\text{vis, mono}} + 0.001 L_s^{\text{mono+multi}}$$

2.2.2 Implementation and Training Details

To train ManyDepth, we embed it into the Dynadepth-framework. Two different models are trained, which differ in the construction of the cost-volume:

- The standard way. Using the $\mathcal{M}_p$-estimated poses to generate the cost volume.
- Incorporating the IMU-data by using the EKF-poses to generate the cost volume.

Since ManyDepth utilizes two separate depth-networks, we apply the IMU-losses twice, once for each of them. The IMU-warp loss L_{photo}^{IMU} (2.1) is also affected by the M-mask the same way as the standard photometric loss, though the IMU-consistency loss $L_{\text{photo}}^{\text{IMU-cons}}$ (2.2) is not. The velocity and gravity losses remain unchanged. The total

combined loss is then:

$$\begin{aligned} L_{\text{total}} &= (1-M)\, L_{\text{photo}}^{\text{vis, multi}} + \lambda_2\, (1-M)\, L_{\text{photo}}^{\text{IMU, multi}} + \lambda_3 L_{\text{photo}}^{\text{IMU-cons, multi}} + M L_{\text{consistency}}^{\text{multi}} \\ &+ L_{\text{photo}}^{\text{vis, mono}} + \lambda_2 L_{\text{photo}}^{\text{IMU, mono}} + \lambda_3 L_{\text{photo}}^{\text{IMU-cons, mono}} \\ &+ \lambda_1 L_s^{\text{mono+multi}} + \lambda_4 L_{vg} \end{aligned} \tag{2.5}$$

where the weighting remains unchanged: $\lambda_{1,2,3,4} = \{0.001, 0.5, 0.01, 0.001\}$.

Training hyperparameters are kept the same as with DynaDepth (2.1.2).

While ManyDepth uses 96 depth levels to generate the cost volume by default, we slightly increase the number to 128, because of the larger depth-ranges on our dataset compared to KITTI. We initialize the depth-range used to generate depth levels equal to the depth range of the single-image network: 1m - 5000m.

2.3 MiDaS and the Dense Prediction Transformer

2.3.1 Model Information

As mentioned in the introduction of this thesis, many depth estimation models are designed for use within the field of autonomous vehicle research. Contrary to the previous two models, MiDaS is more of a universal MDE model, capable of achieving good results on many types of images and having undergone extensive training on large amounts of varied data. Additionally, MiDaS is originally supervised, while the rest of the models we test are self-supervised. MiDaS is trained on twelve different datasets, some simulated and some of them real-world data. The hope of this is that biases that come with learning from a single dataset will be alleviated, but a challenge when doing this is that the datasets have differing ground-truth representations:

- Absolute depth or its inverse, such as from LIDAR,
- depth with unknown scale, such as from SfM,

- or disparity maps from multiple different stereo pairs, i.e. inverse depth with differing and sometimes unknown scales

The main contribution of the authors in the MiDaS-article is the creation of losses compatible with these different ground truth representations, and finding a way to mix the datasets for the best outcome. The losses are computed in disparity space, which generally has preferable numerical qualities as mentioned before. In addition to the unknown disparity scales present in parts of the labeling, the images may have gone through post-processing so the disparity maps have an unknown global shift.

In order to be able to compare the disparities for loss calculation, the prediction and ground truth have their scale (s) and shift (t) aligned. They propose the following estimators:

$$t(\mathbf{d}) = median(\mathbf{d}), \quad s(\mathbf{d}) = \frac{1}{M}\sum_{i=1}^{M} |\mathbf{d} - t(\mathbf{d})|,$$

and then change the prediction ($\hat{\mathbf{d}}$) and ground truth ($\hat{\mathbf{d}}^*$) to have zero translation and unit scale:

$$\hat{\mathbf{d}} = \frac{\mathbf{d} - t(\mathbf{d})}{s(\mathbf{d})}, \quad \hat{\mathbf{d}}^* = \frac{\mathbf{d}^* - t(\mathbf{d}^*)}{s(\mathbf{d}^*)}$$

The losses from the MiDaS paper are not applicable to our data since we have no labeling, and are not presented here. Instead, we will have to embed it into the self-supervised framework of DynaDepth.

The network model itself is referred to as the *Dense Prediction Transformer* (DPT), and is introduced in a separate article [45]. The DPT uses patch embedding of the image following the ViT framework explained in 1.2.5. Using a transformer backbone with multiple layers, transformer-outputs are extracted at four levels and reassembled into an image-like form, which are then incrementally fused together and finally passed into a disparity head which generates the predicted scale- and shift-invariant disparity map. There are multiple different variants of MiDaS, which vary in which transformer model they use. In addition to the image-patch embeddings, a ”readout”-token is included, similar to the class-token in the image-classification usage. While this readout-token

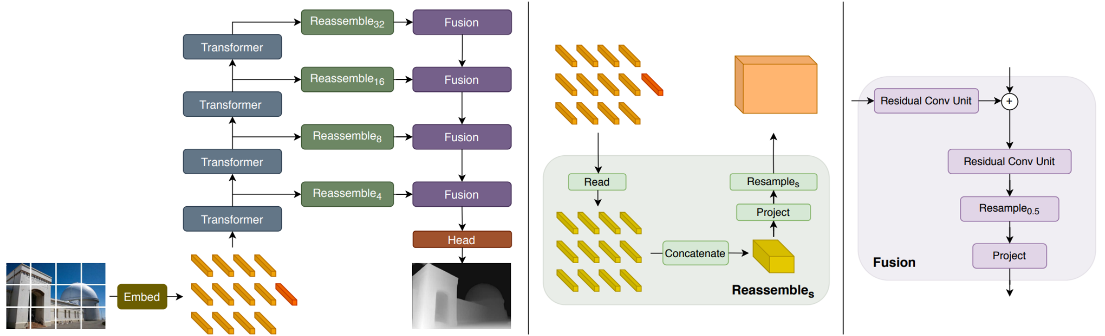


Figure 2.2: Overview of DPT[45]. On the left we see the overall structure model, including the image embeddings (with an additional readout token), transformer-layers, reassembling and fusion blocks. The middle displays the reassembling block where the embeddings are used to recreate the image-like feature maps. The right shows the fusion of these feature maps.

does not have the same intuitive meaning and purpose as the classification token, it improves performance by capturing and distributing global information in the transformer. An overview of the Dense Prediction Transformer is shown in figure 2.2.

Since there is an extra token not grounded in the image, the reassembling block needs to reduce the number of tokens by one. This is done by concatenating the readout-token to each of the image-tokens, and using an MLP to project to the correct dimensions. This is referred to the "Read" function in figure 2.2. These new tokens are concatenated and go through spatial resampling to generate a feature map of varying resolutions, with the feature maps being coarser at the deeper levels of the transformer. The fusion block is based on RefineNet [30], which combines all the outputs from the reassemble-blocks incrementally before the final prediction is generated by a convolutional disparity head.

2.3.2 Implementation and Training Details

As we saw during the specialization project[49], the pretrained MiDaS model is not able to capture fine details in our images, only being able to capture larger features such as different clusters of trees and being able to differentiate between foreground and the sky. Despite originally being a supervised MDE model, we embed MiDaS into the self-supervised framework of DynaDepth and attempt to train using the losses in equation 2.3. MiDaS has several pre-trained variants available, which differ in what kind of backbone they have. We will use the most accurate one as of MiDaS v3.1, which employs a BEiT transformer[2].

To be able to perform the backprojections needed for calculating losses, we need to convert the estimated disparities to depth. Since MiDaS works on scale- and shift-invariant disparities, a simple inversion will not yield useful depths. The evaluation code of MiDaS gives us a way of finding the scale and shift, though this requires a depth ground truth.
After extracting approximate ground truth depth maps from one of the flights using the area-model, we find an approximate scale and shift we apply to all predictions. When training, we use pretrained auxiliary networks to give better supervisory signals early on.

While Dynadepth normally runs with four scales of depth-predictions, MiDaS only outputs one. We attempted to attach disparity-heads to each of the fusion blocks in figure 2.2 to generate four predictions at different scales, though the three additional predictions were unsurprisingly not very good. Only the one full-scale depth prediction is properly pre-trained, and training the three others proved too challenging while using the self-supervised framework of DynaDepth on this model with 345 million parameters. It is well known that large models are more difficult to train, and adding this multi-scale prediction to MiDaS proved to be infeasible for us.
Besides, this would inevitably disrupt the already effective network-weights of MiDaS, which is contrary to our goal of leveraging the extensive training already performed.

Instead, we move forward with only a single full-scale depth prediction.

We use the same hyperparameters as DynaDepth, except with a batch size of 4 because of memory constraints. We also train another "fine-tuning" model, where we use a learning rate of 5e-7, freeze the pre-trained pose-network, and only train for 4 epochs.

2.4 MonoViT

2.4.1 Model Information

While MiDaS/DPT uses a purely transformer based encoder, we mentioned earlier that there are models which combine the usage of CNNs and transformers in their encoder. MonoViT [68] is one such model. Instead of using the image patch method as seen in ViT[10], the start of the encoder consists of passing the image through a convolutional block generating feature maps of half the original height and width, which is then passed through several encoder blocks consisting of both transformers and convolutional layers. A figure of the model is shown in figure 2.3, and follows a common encoder-decoder model which generates predictions at four different scales with the image-size halved at each. As mentioned in earlier sections, CNNs are generally good at capturing local information, while transformers can do well at capturing global information [70]. A combination of CNNs and transformers should, in theory, be capable of capturing both global and local features of an image, potentially yielding better depth predictions.

The joint transformer layer was first intruduced by Lee et al. [27] in their article on *MPViT*, and works as follows:
Instead of a single fixed size patch embedding, convolutions of different kernel-sizes (3x3, 5x5, 7x7) are used to create three different sets of embeddings, each able to capture information at different receptive fields. These embeddings are passed to three transformer-encoders and a convolutional block in parallel, where the embeddings generated from the smallest kernel are shared by one of the transformer layers and the

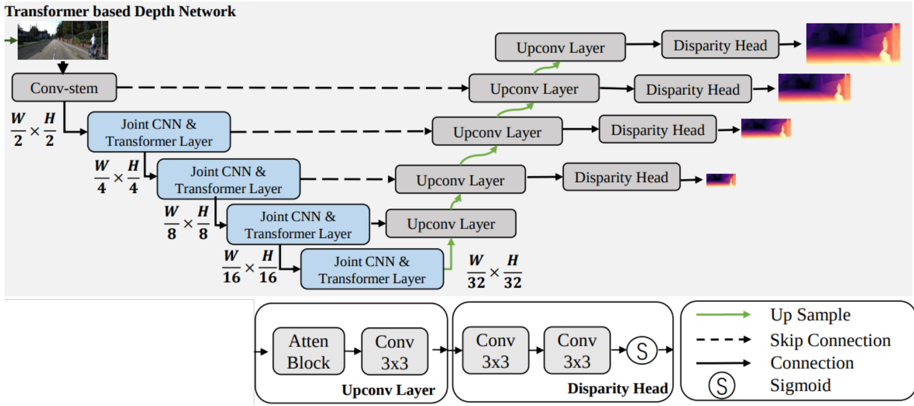


Figure 2.3: Overview of the multi-scale depth network of MonoViT[68]. Following an encoder-decoder structure, the encoder sequentially generates feature maps of gradually decreasing resolution, and the decoder combines up-sampling, skip-connections and disparity heads to generate inverse-depth predictions at four scales.

convolutional block. The outputs from these four blocks are concatenated and passed through a 1x1 convolutional layer, allowing local and long-range features to interact and fuse. This is then passed on to the next layer, which repeats the process. Figure 2.4 shows the structure of the joint CNN and transformer layer.

Like DynaDepth[65] and ManyDepth[63], MonoViT also builds on the self-supervised framework of Monodepth2[18], meaning that a photometric loss with the per-pixel minimum trick is used, similar to 2.1, but with a simple Pose-network instead of the IMU-motion, as well as a disparity smoothness loss.

Contrary to the other transformer-based model we are testing, this one is significantly smaller in terms of the number of parameters (345 million vs 28 million). The self-supervised framework limits the complexity of the models we can utilize, meaning we need to find a middle ground between strong network-capabilities and what we can effectively train.

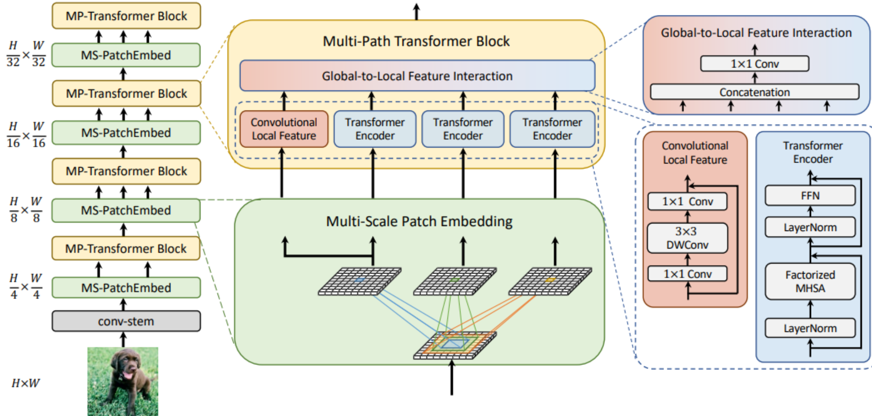


Figure 2.4: The Joint CNN & Transformer layer as visualized by the original authors, Lee et al. [27]. Three different sets of embeddings are created using convolutional layers, and they are passed in parallel into the three transformer layers and the single convolutional layer. The outputs are then concatenated and fused through a 1x1 convolution, outputting a feature map with half the original height and width.

2.4.2 Implementation and Training Details

Following the implementation of the MonoViT-authors, we use the AdamW optimizer and apply a lower learning rate of 5e-5 to the MonoViT-encoder, and 1e-4 to all other parameters. The learning rate scheduling-scheme is also changed: The learning rate decays by a factor of 0.9 following each epoch. All other parameters are the same as with DynaDepth (2.1.2).

Chapter 3

UAV-dataset

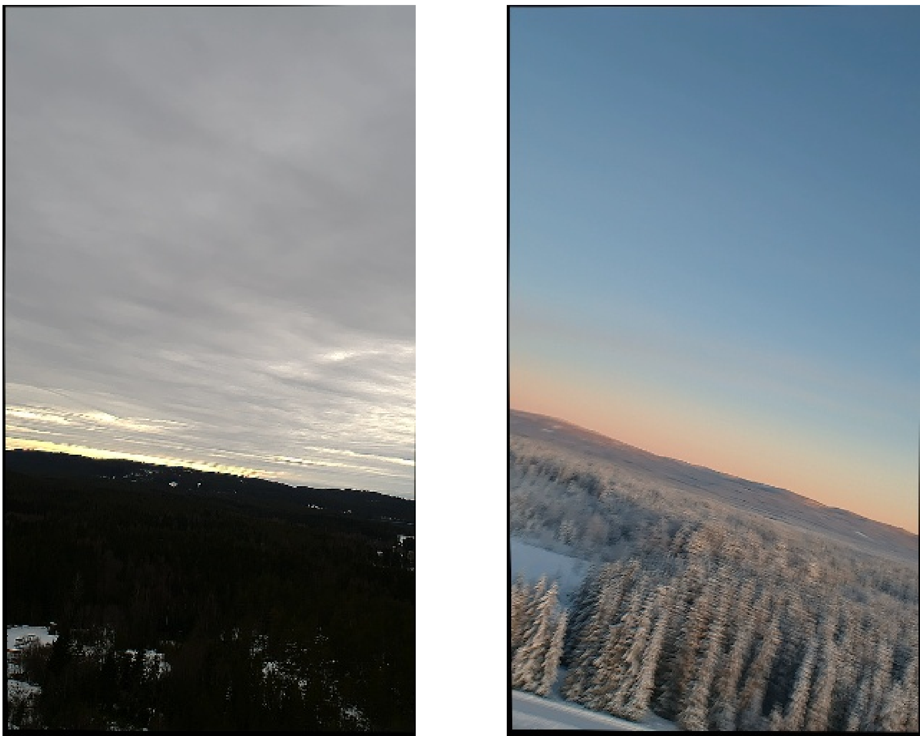
Our dataset consists of five separate flights. Because of the small number of flights, we use one of them as both the validation and test set, while the four others are used for training. We trim the training set to remove sections that are unlikely to generate good supervisory signals. This includes parts with rapid maneuvering and incidents of poor and varying lighting conditions. Rapid maneuvers are too challenging to accurately estimate poses for and cause motion blurring, which is problematic for the photometric loss. Poor lighting conditions also raise issues:

- If compared frames have different lighting-conditions, this violates the assumptions of the photometric loss.
- If the foreground is too dark because of sun-glare, the details of the terrain become very challenging to make out. This makes the depth estimation task much more difficult and might make the photometric loss ineffective because large sections of the image are very similar.

Examples of two images removed from the training are shown in figure 3.1. The first one is removed because of the lighting conditions, and the other because of rapid motions as we can infer from the motion blur. While the worst sections of maneuvering and lighting were removed, parts of the remaining dataset have somewhat poor

lighting and contain jerking motion adjustments and turns, because we do not want to remove too much data.

As mentioned in section 1.2, non-zero translation is required to infer depth, and



(a) Lighting example

(b) Motion example

Figure 3.1: Two examples of removed images from the dataset. a) is removed because of poor lightning, and b) is removed because of the rapid drone rotation.

sections where the drone is stationary cannot give good supervisory signals to the depth network. While DynaDepth incorporates an automasking-feature that blocks out pixels that remain unchanged between images to remedy this problem, this is

much less effective on our dataset. Our drone experiences some constant vibration from the rotors, and we saw during the specialization project[49] that this hampers the job of the automasker, which is not an issue for the car in the KITTI dataset. Because of these problems with the automasker on our dataset, we cut out sections where the drone is stationary, often during the start of a flight. After all the cuts, our dataset is reduced from 33.747 to 19.817 images. For comparison, the commonly used training section of the KITTI dataset consists of 45.200 images.

The test set is not trimmed in any way. Using the area model and the localization provided by the drone's GPS-based navigation system, we extract a pseudo ground-truth for the depth estimates on the test set, which we will use for validation during training as well as a final evaluation of the depth estimates.

The datasets were captured during winter, which may cause issues since the snow-covered ground is very low-texture and indistinct, making it hard for the models to find the correct warping of the pixels as the photometric error will not change much in those parts of the image. The same can be said for the sky. Still, the trees in the images can be used for the alignment of the main image and the warped image. At the same time, the forest is somewhat monotonous as well, though this is more of an issue at a distance where the image is less detailed, and not as much up close. We have made sure not to fly very high, though it might have been better to pitch the camera down to not include as many objects in the distance as well as have less of the sky present in the image.

There were some issues when capturing the data, with the sampling-rate of the IMU and camera FPS varying throughout the flights. We experienced this during the specialization project[49] and assumed this had to do with the software being overloaded and discarding some images and measurements. Because of this, we chose to discard the old dataset. To address these problems in this Master's Thesis, we captured new data with reduced video-framerate, hoping that this would fix the issue. While this did improve the situation, it did not completely fix it. We will later discuss the issues

this will cause during training. An added benefit of the reduced FPS of the video is the increased length of the translation between images, which is beneficial for training since this makes depth easier to estimate using the photometric loss.

Before training, we undistort the images and resize them from 1080×1920 to 288×512. The resizing maintains the original aspect ratio of the images to prevent stretching or squeezing of the images:

$$\frac{1920}{1080} = \frac{512}{288} = 1.777...$$

The image downsampling is done to speed up training and reduce memory requirements.

Chapter 4

Monocular Depth Estimation Results and Discussion

In this chapter we will first outline some problems we experienced with the baseline DynaDepth model, and explain what we did to address them. Then, we report on the results of all the trained models from chapter 2 with these improvements included. Following the results, we discuss various points related to the results, before we finally decide which MDE model to use in the final part of the thesis regarding PCR.

4.1 Problems and Improvements to DynaDepth

During the specialization project[49] we found it more challenging to use our UAV-dataset to train DynaDepth than KITTI was for the original authors. We hypothesized that the increased freedom of movement of a drone compared to a car made the estimates generated by the auxiliary networks (relative pose, gravity, velocity) much less accurate than was desired.

Indeed, inspecting the estimated gravity vectors, we see that they are far off. For

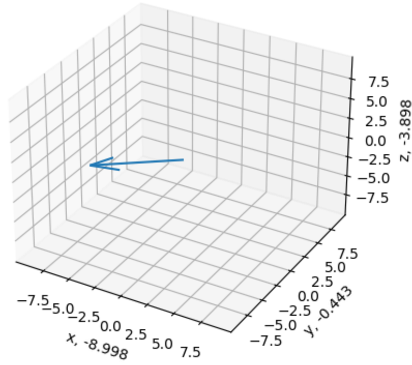
example, in figure 4.1, the gravity network yields an estimate sideways, which is obviously incorrect. We observe that the gravity network $\mathcal{M}_g$ converges to a seemingly random fixed gravity estimate early on in training and outputs this gravity-vector regardless of the image it receives. As mentioned in section 2.1, the gravity loss only takes into account the magnitude, and not direction, of estimated gravity. The direction of gravity would need to be inferred through the EKF-motion in the photometric loss 2.1. This is not sufficient for our data, and we instead propose using GPS orientations during training to generate a good estimate of gravity for supervision. Doing this, the gravity L2-loss will compare the vectors instead of only the magnitude.
Note: When we refer to GPS, we are using the corrected GPS-measurements that were generated by the navigation system of the drone, which fuses GPS, IMU, altimeter, and motor inputs.

The predicted velocity has some issues as well. While it is not possible for us to know exactly what the velocity should be by looking at the images, we can intuit that the velocities at consecutive frames should be similar as long as there are no abrupt maneuvers. The following are two consecutive velocity estimates where the drone is flying straight ahead, transformed to front-left-up orientation for ease of interpretability:

$$\mathbf{V}_0 = \begin{bmatrix} 4.933 \\ -0.499 \\ -1.147 \end{bmatrix}, \mathbf{V}_1 = \begin{bmatrix} 3.729 \\ 0.039 \\ 3.269 \end{bmatrix}$$

While $\mathcal{M}_v$ has been able to learn that the drone moves forward in the x-direction, we see some clear discrepancies in the other dimensions.

(a) Image



(b) Gravity vector

Figure 4.1: Example of erroneous gravity estimation on the test set. While the gravity vector should point down, we see that the estimate points to the side.

In general, both the velocity and pose-networks, $\mathcal{M}_v$ and $\mathcal{M}_p$, are biased in favor of moving straight ahead since this is the case most of the time in the data. This is much less of a problem if you are training on the KITTI-dataset which DynaDepth was originally created for, since the car almost always moves forwards and does not need to perform pitching and rolling maneuvers to change direction. Figure 4.2 shows an extreme example of a drone maneuver where the drone turns while moving sideways,

and the EKF and $\mathcal{M}_p$ estimated motions struggle to follow the turn and movement. This figure also shows the benefit of using IMU-measurements and not only relying on deep learning, as the EKF-motion has an easier time following the rotation of the drone.

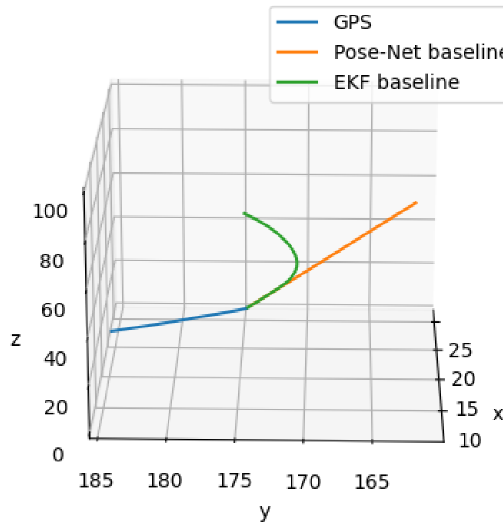


Figure 4.2: Example of failed pose estimation because of a challenging drone maneuver. This section was subsequently removed from training, as it was deemed to not generate useful supervisory signals.

Since accurate motion-estimates are crucial to generate good supervisory signals for the neural networks, we propose to add direct supervision on $\mathcal{M}_p$ and EKF-motion with GPS-based poses. Using measured GPS pose at image k and $k+1$, the relative motion in camera frame is calculated as:

$$\mathbf{T}_{gps} = \mathbf{T}_{cg}\mathbf{T}_k^{-1}\mathbf{T}_{k+1}\mathbf{T}_{gc}$$

The GPS uses the common front-left-up axis convention, while camera-frame is right-

down-front. $\mathbf{T}_{gc}$ represents the transform between these coordinate frames, such that the relative motion is represented in camera-frame. We create two MSE-losses with $\mathbf{T}_{gps}$, one comparing with the EKF, and the other comparing with $\mathcal{M}_p$, and add them to the total loss of the various models with a weight of $\lambda_{gps} = 0.5$:

$$L_{total} \mathrel{+}= 0.5\mathbf{MSE}\left(\mathbf{T}_{gps}, \mathbf{T}_{posenet}\right) + 0.5\mathbf{MSE}\left(\mathbf{T}_{gps}, \mathbf{T}_{EKF}\right)$$

We hope that this additional loss can improve both $\mathcal{M}_p$ and $\mathcal{M}_v$ to generate better motion estimates, which in turn might yield better depth predictions. Adding direct supervision on the translation also has the added effect of resolving the scale ambiguity, though IMU-data plays an important role in that as well.

4.2 Depth evaluation on the test-set

When evaluating the depth accuracy, we limit ourselves to error computations for depths within 200m in the ground truth, since we know that large depths are more prone to be inaccurate. Additionally, we apply cropping of 5 pixels on the sides and 15 pixels on the bottom to avoid depth artifacts on the edges caused by the undistortion performed on the images. We report RMSE along with some other commonly used metrics as introduced by Eigen et al. [11]:

- **AbsRel**: Mean of absolute relative depth error, $\frac{|\hat{D}-D|}{D}$
- **SqRel**: Mean of square relative depth error, $\frac{\left(\hat{D}-D\right)^2}{D}$
- **RMSLE**: Root Mean Square Logarithmic (depth) Error
- δ_i: Proportion of pixels where $max\left(\frac{\hat{D}}{D}, \frac{D}{\hat{D}}\right) < 1.25^i$. Close to 1 is best.

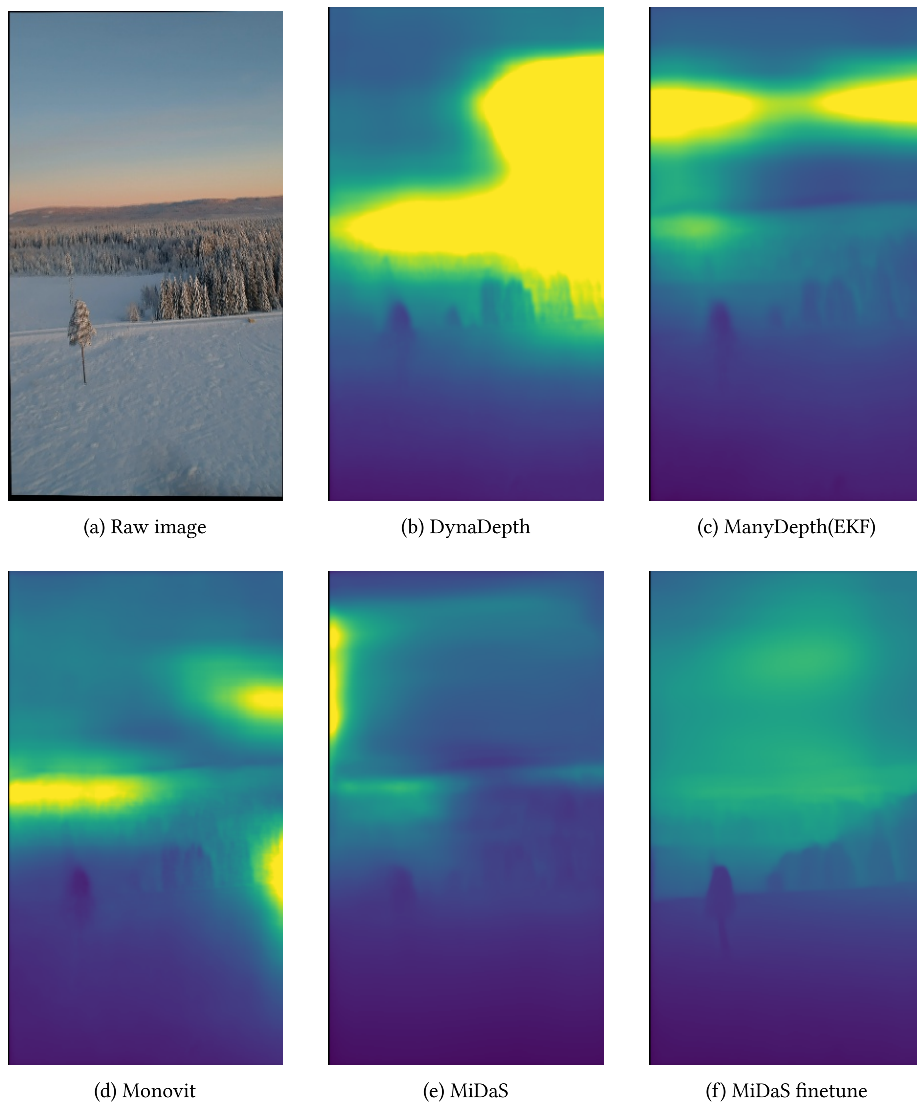
To allow for depth evaluation, we use the pseudo ground-truth generated from area model along with an estimated pose from the drone's GPS-based navigation system. This generated depth map will not be fully accurate, both because of limited detail in the 3D-model, as well as some pose uncertainty. Similar to the original training

images, the pseudo ground-truth is resized from 1080×1920 down to 288×512 to make it compatible with the depth estimates.

Before evaluating an MDE model, it is necessary to resolve the scale-ambiguity of the task. Despite incorporating both IMU and GPS data during training, we did not manage to achieve the correct scale on the test-set. Following Godard et al. [18], we apply median scaling, which consists of scaling the depth predictions such that the median depth aligns with the median of the ground truth. We report the scale along with the scaled error metrics averaged over the test-set in table 4.1, constrained to within 200m. In the table, Dynadepth(w/o GPS) represents the baseline DynaDepth model without our additions. Figure 4.3 shows a depth-prediction comparison between the trained models, and an error map of the same scene is shown in figure 4.4.

Table 4.1: Scaled depth error metrics on the test-set, along with mean scale

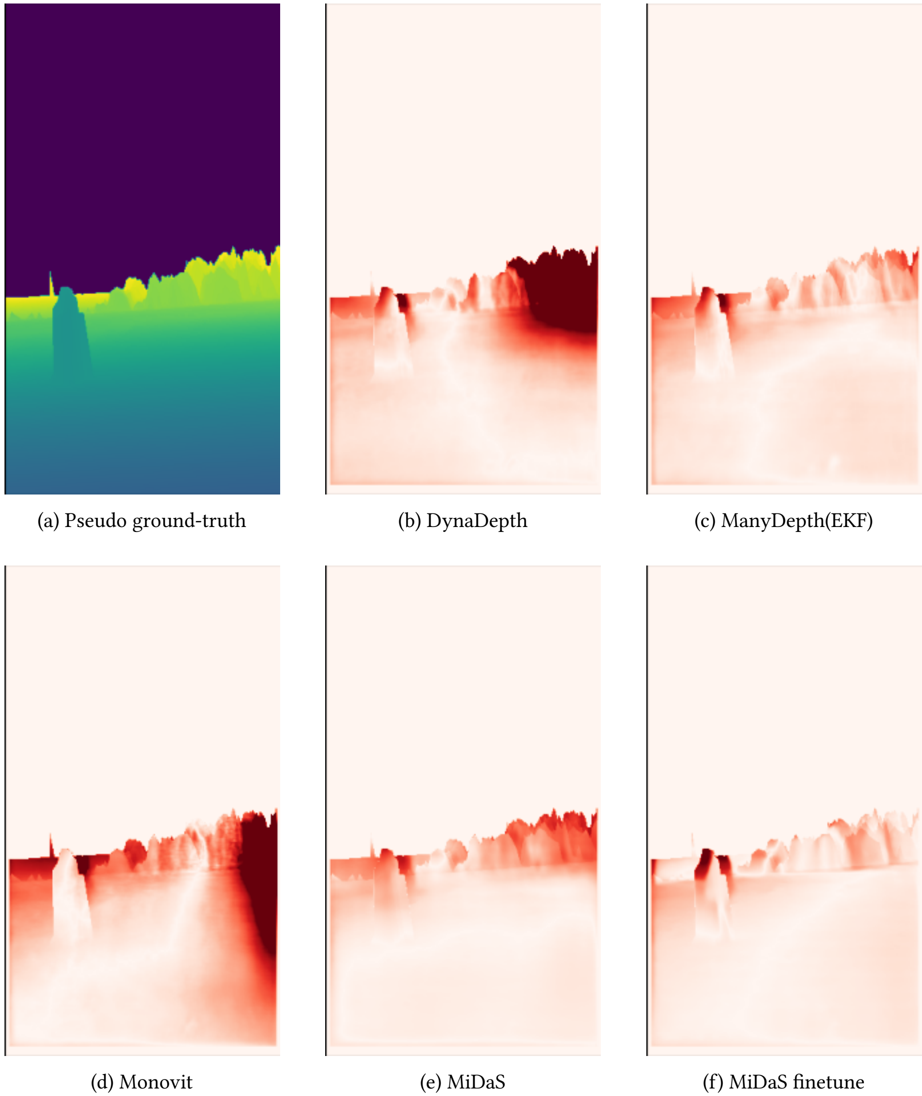
MODEL	**Mean Scale** $\pm \sigma$	**AbsRel**	**SqRel**	**RMSE**	**RMSLE**	δ_1	δ_2	δ_3
DynaDepth(w/o GPS)	0.890 ± 0.162	0.169	10.196	32.003	0.208	0.776	0.950	0.986
DynaDepth	0.870 ± 0.157	0.154	9.159	30.805	0.195	0.804	0.960	0.989
ManyDepth	0.871 ± 0.153	0.135	3.558	20.099	0.176	0.812	0.973	0.998
ManyDepth(EKF)	0.824 ± 0.150	0.130	3.041	19.047	0.170	0.812	0.981	0.999
MonoViT	0.808 ± 0.133	0.156	9.136	30.450	0.208	0.775	0.954	0.989
MiDaS	0.693 ± 0.155	0.128	3.208	20.124	0.176	0.814	0.970	0.996
MiDaS finetune	0.858 ± 0.179	0.107	2.265	16.319	0.144	0.881	0.986	0.998



(a) Raw image (b) DynaDepth (c) ManyDepth(EKF)

(d) Monovit (e) MiDaS (f) MiDaS finetune

Figure 4.3: A comparison of five models evaluated on an image from the test set. For all depth-images, we limit the color intensity of the image to 300m for ease of comparison and to prevent the foreground from becoming too dark in some of them.



(a) Pseudo ground-truth

(b) DynaDepth

(c) ManyDepth(EKF)

(d) Monovit

(e) MiDaS

(f) MiDaS finetune

Figure 4.4: Comparison of error maps after performing median scaling. Error computation is limited to within 200m. The area model used to generate the pseudo ground-truth is not completely accurate, nor is the estimated drone-position which the depth image is generated from. As such, the error metrics reported in table 4.1 should be taken with a pinch of salt.

4.3 Observations and Discussion

4.3.1 Depth Errors

A common erroneous result we observed for almost all trained models was the depth estimated in the sky. Generally, the upper sections of the image are estimated as being as close as the ground, though some parts of the sky have large estimated depths. Since this affects all models, this indicates that this error is an effect of our supervisory signals or the data, not of the depth networks themselves. It is not immediately clear what could cause this. A possible explanation is that it might have to do with the fact that no depth-estimate managed to change the photometric losses in those sections significantly, so the initial estimates of depth from the start of training remained, and those are generally low. This hypothesis is contradicted by the MiDaS training results. The initial estimates of sky-depth at the start of our MiDaS-training is the maximum depth possible, 5000m, since it is good at detecting the difference between foreground and background. As training continues, this quickly degrades to the results we see in our other trained models.

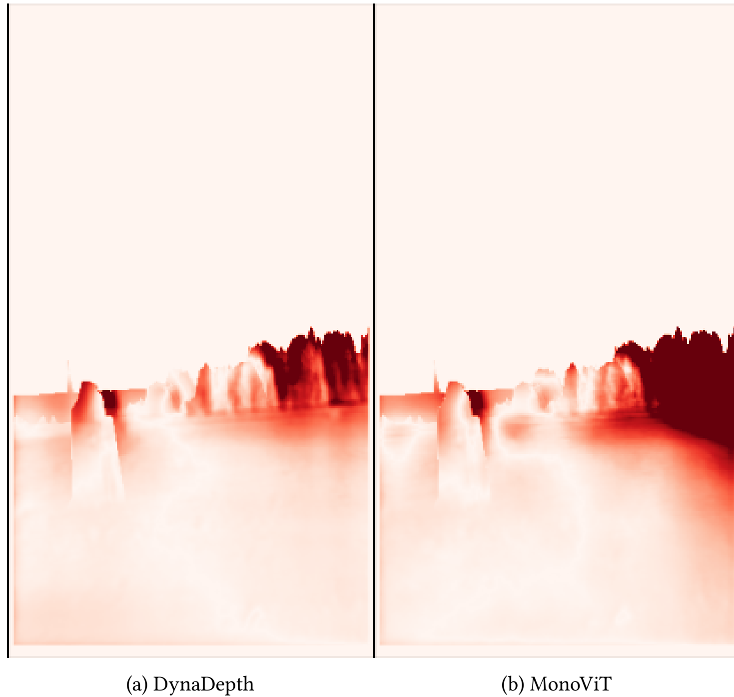
Inspecting the sky in some parts of our dataset might give an indication of the reason. For example, in figure 4.5, the color of the snow-covered ground is somewhat similar to the sky in the image. This can be a contributing cause of the problem seen in figure 4.3, where the large parts of the sky have similar depths as the ground. A compounding factor is the aforementioned difficulties of finding the correct warpings in these sections, as the photometric errors are likely to be small since neighboring pixels are similar, though some changes in the moving clouds could confuse the situation. The combination of similarities with the ground as well as poor supervisory signals can explain these depth errors.

Figure 4.5: Example of an image from the training set where we see that the snow-covered ground has a very similar color to the clouded sky.

As mentioned in the previous section, we constrain the depth evaluation to within 200m. Depth estimation is inherently more difficult at longer ranges because of the limited resolution and sensitivity of the camera, which causes degradation of details in faraway sections of the image, though this degradation can be a depth-cue in and of itself. As touched on in previous sections, disparity shifts between images are required to infer depth. This means that objects that are far away are hard to calculate depth for. This is represented in our models by the depth appearing inversely in the photometric

loss. This means the impact depth has on the loss decreases as the depth increases. For instance, the change when adjusting from $z = 3$ and $z = 4$ is much more significant than from $z = 300$ to $z = 400$, making the exact depth harder for the photometric loss to decide for faraway objects. This is exacerbated by the level of detail in the image decreasing with increased depth, meaning that a slight pixel-warping error is less significant. Since we are using gradient descent, the rotational part of the ego-motion is much more impactful to adjust at these ranges, possibly making the depth converge to lower values than it should. Figure 4.4 shows the trend of larger errors further away from the camera, though as explained, the figure only shows results within 200m.

The depth error maps in figure 4.4 reveal some depth-artifacts for both DynaDepth and MonoViT, where there are large errors on the section of trees to the right in the image. It is not clear what could cause this error, though it may be tempting to blame overfitting. As mentioned in chapter 3, our dataset is much smaller than KITTI. While DynaDepth was originally trained for 30 epochs on KITTI, we reduced it to 16 on our own data, since a smaller dataset reduces the number of epochs before we can expect to see overfitting. These errors also appear at checkpoints early in training, indicating that it may not be from overfitting, but rather having to do with poor supervision. Figure 4.6 shows a similar error being present for DynaDepth and MonoViT at epoch 8. The aforementioned depth-sensitivity problems mean that the model overshooting the depth severely in faraway sections is not heavily punished by the photometric loss, as both the observed disparity shifts and the pixel warpings will be tiny regardless. This means that the depth networks are not properly supervised to fix these depth artifacts once they appear during training, and similar artifacts are also observed on parts of the training set. This is even more likely to occur in monotonous low-texture regions, where the photometric loss generally struggles since the disparity shift is less apparent. In general, the large depths in our images mean that large parts of the image do not provide good depth-network supervision through the photometric loss. Tilting the camera further down when recording the data may have been a better choice, as this would decrease the depths of the images.



(a) DynaDepth (b) MonoViT

Figure 4.6: DynaDepth and MonoViT error maps at epoch 8, where we still observe similar depth artifacts on the right as seen in figure 4.4.

Looking at some of the inverse-depth predictions during training, we can easily see problems. The poor lighting and maneuvering make the self-supervised depth estimation problem very challenging, as figure 4.7 reveals. While the image is shown for DynaDepth, poor results are seen for the other MDE models as well, even for MiDaS early in training. This means that the problem is not just from not being able to generate useful supervision, but also because the image is hard to infer depth from.

(a) Image (b) Inverse Depth

Figure 4.7: Example of poor results during DynaDepth training, indicating inadequate supervision and a challenging estimation problem caused by the lighting and movement.

4.3.2 Auxiliary Network Estimates

In addition to poor lighting and large depths, we assume a large problem of our training is the difficulty in properly estimating ego-motion for use in the photometric loss. The self-supervised framework was originally designed for use within the domain of autonomous vehicles, where we have cars driving on roads. A drone allows for full 6-DOF maneuvers instead of the 3-DOF of the car, which are forward motion along with sideways motion and yawing during turns. This means we have a much more challenging estimation problem than what the Pose-net $\mathcal{M}_p$ and velocity network $\mathcal{M}_v$

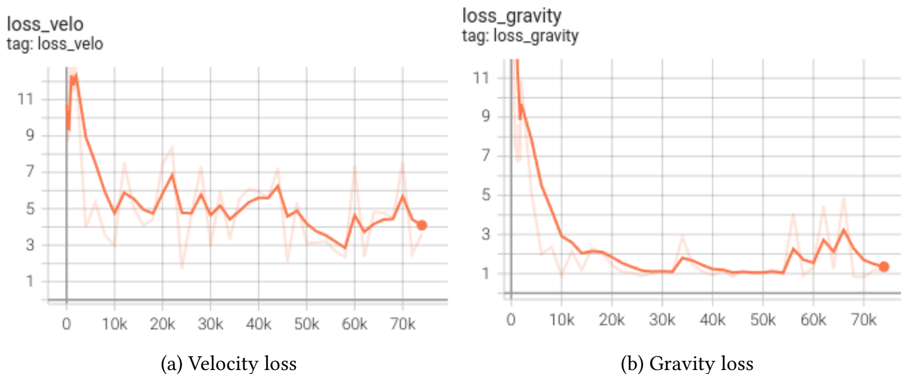
were created for. Since DynaDepth implements a variant of the Pose-net that also attempts to estimate pose-variances to use in the EKF, we can inspect the estimated uncertainty of the pose predictions. While the authors of DynaDepth managed to achieve low variance on the forward-component of the translation, we see high uncertainty in all six degrees of freedom. The effect of this is that the EKF-motion is entirely based on the IMU-motion, with little inclusion from the Pose-net $\mathcal{M}_p$.

Inspecting a comparison of the estimated drone motion between IMU and Pose-net reveals that this may be the correct choice. Figure 4.10a shows a part of the estimated motion on the training dataset. As we only estimate relative motion, we must manually accumulate them together, which also means we drift over time. That is why we only show a small section without significant maneuvers for this comparison. IMU-motion is closer to the GPS, though it is a bit noisier, which makes sense as the IMU experiences noise as well as significant vibrations from the drone rotors. There are also the synchronization problems between the images and the IMU and GPS, which means the first and last IMU-measurements between training images are not perfectly in synch with the timestamps of the images. Though this ought to be a minor inconvenience most of the time since there are usually many more measurements included such that the overall cumulative motion is very close to correct. However, because of the varying FPS of the camera and sampling rate of the IMU, there are some sections where there are only two or three measurements recorded between the images, giving a less accurate relative motion estimate.

Noisy IMU-motion can also be caused by difficulties in estimating the starting velocities with $\mathcal{M}_v$. Since we have slightly varying FPS of our camera and $\mathcal{M}_v$ has no information about the timesteps, it is technically impossible for it to observe the velocity. The synchronization problem is likely much more significant in regard to GPS-measurements than the IMU. The GPS-measurements affect the velocity network $\mathcal{M}_v$ both through velocity-magnitude supervision (part of L_{vg} in equation 2.3) as well as the EKF-motion supervision, and the synchronization issues will make the supervisory signals less accurate. Figure 4.8a show the velocity loss on the validation set during training. The

losses still remain high and have some spiking, showing the difficulty in estimating velocity.

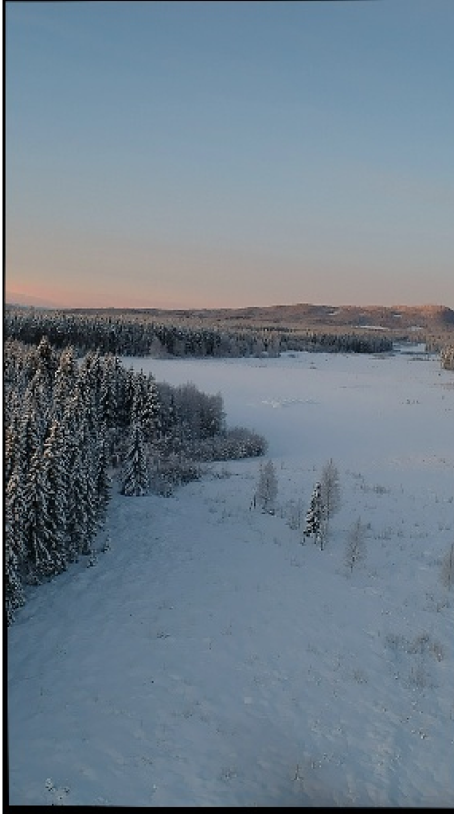
While the velocity estimation struggles, the gravity estimates are a bit better as the loss in figure 4.8b shows, even though it suffers the same synchronization issues. Still, the improvement of adding supervision of the direction of gravity as outlined in section 4.1 quickly makes the gravity estimates better. Even if the generated ground truth gravity is slightly wrong because of the delay between the image and GPS, as well as some inevitable inaccuracies in the GPS-based orientation, it will still point in the approximately correct direction, as opposed to what we saw in figure 4.1. Predicted gravity after the change to the supervision is shown in figure 4.9.



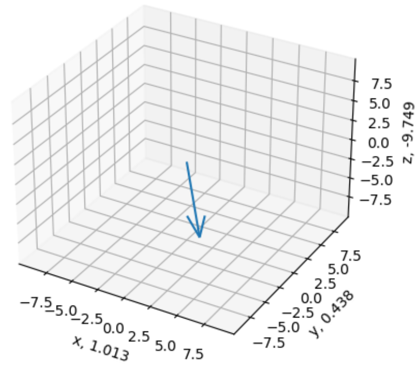
(a) Velocity loss

(b) Gravity loss

Figure 4.8: Gravity and velocity loss on the validation set during training of DynaDepth with the added GPS supervision. As said in chapter 2, we originally trained for 30 epochs, but are using checkpoints at epoch 16 for evaluation.



(a) Image

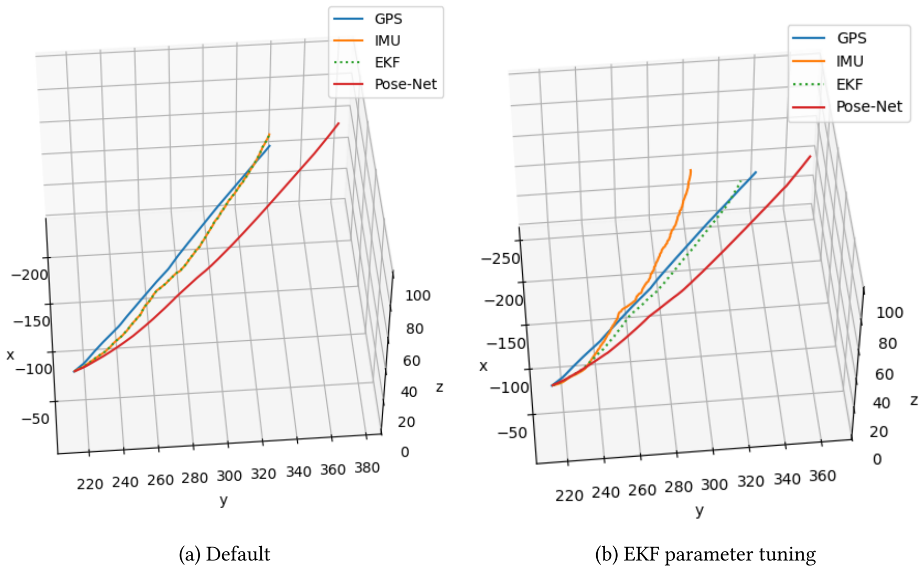


(b) Gravity vector

Figure 4.9: Example of good gravity estimation on the test set after changing the gravity loss. The drone is slightly pitched forward as it flies, which makes part of the gravity vector point along the x-axis of the body, giving the positive x-value of the gravity estimate.

It would be ideal if the visual information could smooth out the IMU motion through the Kalman filter, though this does not happen because of the aforementioned Pose-net estimated variances. It is possible to manually adjust the IMU noise covariances to be closer to the Pose-net as we did in figure 4.10b, but this did not show any clear improvement of the depth estimates. The resulting motion estimates after our tuning

make it seem like the IMU-motion is made to drift in the opposite direction of the Pose-net, such that the EKF-estimate stays in between them and remains somewhat similar to before our tuning. Since we saw no improvement, we stuck to the default EKF-tuning when training the models in table 4.1.



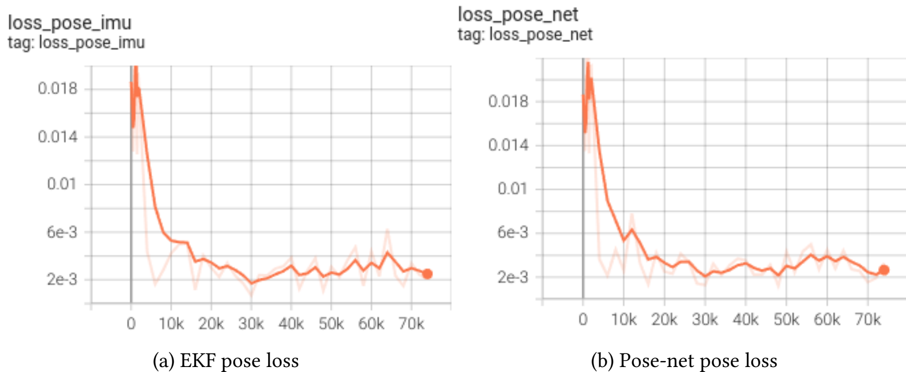
(a) Default

(b) EKF parameter tuning

Figure 4.10: Comparison of the Pose-net, IMU, and EKF motion trajectories in a small section of the training data. While the Pose-net seems to underestimate the turning of the drone and drifts early, the EKF motion is noisier though with less overall drift, though the EKF drift in height is not as apparent from this viewing angle. a) shows results using the default EKF parameters by Zhang et al. [65], where the EKF ignores Pose-net. b) is of our own tuning by increasing the IMU noise covariance to be more in line with the Pose-net covariance, such that the EKF incorporates visual data.

The motion estimates also raise the question of whether increasing the weighting of the IMU photometric loss 2.1 in the total loss (equation 2.3) to $\lambda_2 = 1$ would be better, though again, this gives negligible changes in the results, causing marginally worse error metrics and no noticeable improvement when inspecting the depth images. Figure 4.10 may be giving the false impression that just because EKF-motion appears to

follow the GPS better, it is better than the Pose-net for the purposes of depth estimation through the photometric loss. It is important to remember that the relative motion is all we need, not the overall trajectory, and the noise we see in the EKF-motion could mean that the relative motion error is larger than we assume. Inspecting the GPS-based pose loss in figure 4.11 we see that they are both approximately the same. From the motion comparison, it looks like while Pose-net consistently underestimates the motion, the EKF/IMU-motion has consistent noise errors, yielding relative-pose estimates that have similar levels of precision. The angle of the image also makes it hard to see the drift in altitude for the EKF compared to GPS, and any drift in rotation is not revealed from the trajectory, so the EKF-motion is less accurate than it looks. The same holds true even with the EKF-tuning mentioned earlier.



(a) EKF pose loss

(b) Pose-net pose loss

Figure 4.11: GPS-based pose MSE during training of DynaDepth

4.3.3 ManyDepth

While training ManyDepth we observed that the consistency loss (2.4) increased during the course of training, seemingly struggling to decrease and converge. It seems that the multi-image depth network, in large part, is pushed towards replicating the results of the single-image network, though it still achieves superior results as seen in table 4.1. Figure 4.12 shows the logged consistency loss during training of the ManyDepth(EKF)

model.

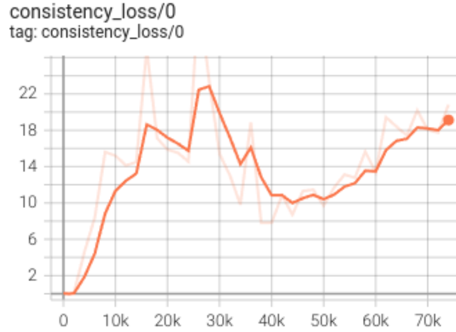


Figure 4.12: ManyDepth consistency loss (equation 2.4) on the training set. This loss grows much larger than all other losses, and does not have a downward trend.

The inclusion of the single-image depth network is supposed to combat the problems caused by moving objects in the scene, where the multi-image depth network yielded very poor results, much more so than the standard mono-image network of monodepth2[18]. This is not an issue in our dataset since we have completely static scenes with no people or cars moving around. One might assume that disabling the consistency loss might then be better, though in our testing, this caused deterioration of the results. It is likely that the large depth deviations causing the large consistency losses stem from poor single-image predictions in parts of the images (as seen in the DynaDepth error-maps in figure 4.3b), with some addition from inaccuracies in pose estimates used to generate the cost-volume. Still, we see an overall improvement in the depth predictions, with more consistent estimates without as many artifacts as we see with Dynadepth.

There is also a slight improvement when using the EKF-poses instead of the Pose-net $\mathcal{M}_p$, though the scale is further off. As we see in table 4.1, the scale is more inaccurate for the model using the EKF to generate the cost-volume. This is quite unexpected, as we would assume adding scale-aware data directly into the depth estimator, and not just through the supervisory signals (IMU-losses, 2.1, 2.2), would increase the

scale-accuracy. The scale-awareness is bottlenecked by the velocity network since the IMU only helps to find the change in velocity between the images, and not directly resolving the scale-ambiguity of the translation, meaning the depths will also not be scale-aware. Overall, the decreased scale-accuracy when using EKF-motion in the ManyDepth depth network is likely related to the EKF-motion problems mentioned earlier: IMU-noise, synchronization issues, and the velocity not being observable for the velocity-network $\mathcal{M}_v$.

4.3.4 MiDaS

As we can see in table 4.1, MiDaS achieved the least accurate scale when trained normally, i.e. not the finetuning way. It is possible that this is related to us having to give MiDaS an approximate scale to start with. The pretrained pose network $\mathcal{M}_p$ had its own learned scale that likely was somewhat conflicting with MiDaS at the start, and overall they ended up with a worse scale-results than the others. On the other hand, when we froze the pose network in the finetune-training, MiDaS was forced to learn a scale that was compatible with the Pose-net. As such, we see that "MiDaS finetune" has a more accurate scale in table 4.1.

An important metric not seen in table 4.1 is the distinctness and consistency of the depth predictions. For instance, only the MiDaS-finetuning model manages to generate a distinct outline of the lone tree and generally has less blurry predictions. This is not the case for the other trained MiDaS model, which has much more blurry and indistinct depth estimates, despite having comparatively good error metrics. As exemplified in the error map (4.4e), the good error metrics are caused in large part by the indistinct depth estimates fitting well with the ground, as well as not having any extreme depth artifacts as we see with DynaDepth and MonoViT.

4.3.5 Incorporation of GPS pose supervision

While we see some small improvement when using GPS-supervision, one might expect the motion estimation to be significantly better. We investigate the improvement

by evaluating the GPS-pose losses on the two DynaDepth models trained with and without the additional GPS supervision we described in section 4.1. As table 4.2 reveals, the improvement for the Pose-net is small, and the EKF-motion is worse. However, there is slightly less deviation of the pose-losses, which indicates that we managed to improve the consistency of the estimates as there were fewer large motion errors. The pose supervision on the EKF-motion was mostly intended to improve the velocity network $\mathcal{M}_v$, though the EKF MSE-loss on the test set indicates our failure. Though as mentioned before, the varying FPS of the camera makes the velocity unobservable, something that cannot be remedied no matter how strong and precise the supervision is.

A possible explanation for the unimpressive motion improvement is that the motion between images is very small, meaning that the MSE-loss will be very small so long as the pose estimates are not wildly inaccurate. A result of this is that the other losses dominate the optimization steps. Figure 4.13 reveals that the trajectories drift quite early, even with the very low pose-losses. We attempted to increase the weight of the MSE-loss to address this, but this gave worse depth-results. This supervision is not completely accurate, both because of the uncertainty of the GPS pose-estimates, and also because of the aforementioned synchronization problems between the frames and GPS-measurements. Images are matched with the most recent GPS-measurement, and this delay is a source of inaccuracy, especially during fast maneuvers. This means that the optimal motion for the GPS-losses is slightly different from the photometric losses, and necessitates a somewhat low weighting of the GPS-pose loss so that these errors don't contribute to worse depth-estimates. The photometric losses are the only ones related to depth, and ought to be the main losses to minimize.

We also hoped to see more accurate scales when adding GPS-supervision. IMU-data plays a role in finding the correct scale, though less directly, as supervision of the estimated translation of the drone ought to yield an even more accurate scale. However, looking at table 4.1, this is not the case as the Dynadepth-model trained without our proposed modifications has a slightly more accurate scale on the test set. However, it should be said that the scale will be unobservable at test-time, since the scale is only

Table 4.2: Average Pose-net and EKF-pose losses on the test set with and without the additional GPS supervision in section 4.1.

MODEL	Pose-net loss	EKF-pose loss
No GPS	0.00596± 0.00714	0.00680± 0.00799
With GPS	0.00468± 0.00485	0.00721± 0.00700

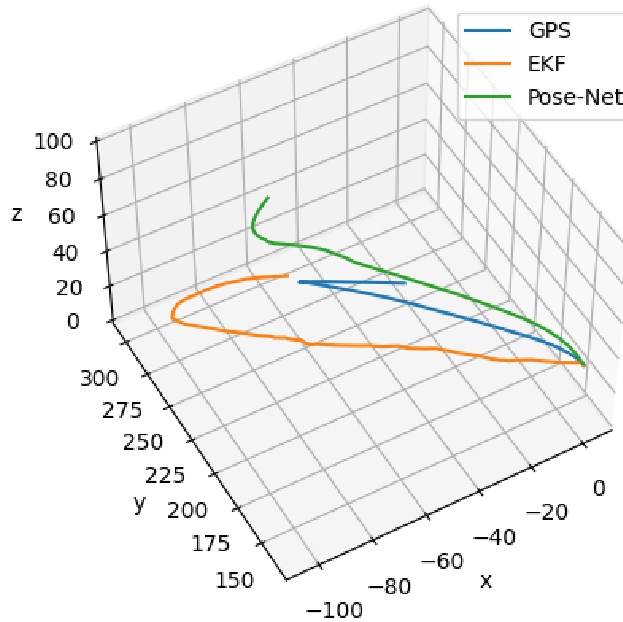


Figure 4.13: Pose-net and EKF trajectories on the test set, with GPS supervision included. As we see, the EKF-motion drifts a bit faster than the Pose-net. Since the flight contains some stationary hovering, the forward motion bias of the velocity network and the pose network makes it seem like the scale of the estimated trajectories is much larger than the GPS, though it is not.

inferred during training. If we instead inspect the scale on one of the training flights, table 4.3, we see results more in line with what we expected. As DynaDepth already does a good job of finding the scale, the improvement is not massive in absolute terms, though the relative improvement is quite large. The standard deviation of the scale is much larger than DynaDepth achieved on KITTI[65], which hints at the difficulty of monocular depth estimation on our dataset.

Table 4.3: Average and standard deviation of the scale within 200m in one of the training flights.

MODEL	**Scale**		**Std**
DynaDepth(w/o GPS)	1.0259	±	0.191
DynaDepth	1.0038	±	0.183

4.3.6 Effects of self-supervision

It is well known that self-supervision is a more challenging problem than supervised learning[41], and this came to light when training MiDaS. While we have included direct supervision of the auxiliary networks, the depth networks are still self-supervised. As we explained in chapter 2, we had to perform some extra steps to be able to train MiDaS on our own data using the self-supervised framework of DynaDepth. This was to load pretrained weights and only perform a single scale depth prediction to ensure we could leverage the training already performed on other datasets. Since the neural network of MiDaS contains 345 million parameters, using self-supervision was a tall order, though the extensive supervised training already performed let us achieve our best results when finetuning on our own data.

It is not a coincidence that the largest neural network models gave us the most trouble, since these are more demanding in terms of good supervision, which is harder to generate using self-supervision [41] [38]. While a supervised MDE model can directly apply supervision to the depth network, we use the ill-posed photometric loss[7] to

generate supervision for multiple networks at the same time, and at the same time trying to resolve the scale-ambiguity by including the IMU (equation 2.1 and 2.2). A comparison of the parameters in the four different depth networks we have used is shown in table 4.4. The larger number of parameters of MonoViT may have been its downfall. While MiDaS was pretrained, MonoVit was not and did not significantly surpass the relatively simple depth estimation network of DynaDepth.

The smallest depth network we trained was ManyDepth, which turned out to be the second best w.r.t the error metrics. It also did not experience as many of the seemingly random depth errors as DynaDepth and MonoViT did, as exemplified in figure 4.4. This may be because of the reduced model complexity, but the parameter difference with DynaDepth is small, so it is likely caused by the increased capabilities gained when including more than a single image for depth estimation.

We could have generated the pseudo ground-truth for all of the training data, but since we know it is somewhat inaccurate as exemplified by figure 4.4a, we decided to use self-supervision despite the challenges it poses.

Table 4.4: Number of parameters in the respective depth networks

MODEL	**Parameters**
DynaDepth	17.791.564
ManyDepth	17.388.532
MonoViT	27.870.108
MiDaS	345.014.441

4.3.7 Training Data

We have seen success in applying self-supervised MDE to UAV-datasets, such as the work of Madhuanand et al. [36], who used the UAVid dataset [35]. However, inspecting this dataset we see some quite significant changes which make it more ideal than ours

for the purpose of self-supervised depth estimation:

- Mostly forward flying motion, with smooth maneuvering and mostly constant elevation
- More consistent velocity
- Fixed FPS
- Urban environment with good lighting conditions
- Camera tilted down, yielding depth ranges in the images far smaller than ours

On the other hand, our dataset has:

- frequent motion adjustments, consisting of quick pitching and tilting jerks, as well as altitude changes.
- Varying speed
- Slightly varying FPS of the camera and also varying sampling rate of the IMU
- The environment is of forested winter landscapes, with varying lighting conditions.
- Depths up to thousands of meters.

It is clear that the UAVid dataset makes it much easier for a self-supervised model to generate good supervisory signals since the job of the Pose-net and velocity network is made much simpler. While our dataset can make the auxiliary networks better because of the varying movement, this is not our goal, as we only need good depth estimates. ManyDepth is an outlier in this sense, as the motion between frames is used in the depth prediction, meaning that our dataset has an advantage in this case. As touched on earlier, urban environments make the depth prediction task easier, since there are more distinct colors and lines instead of the monotonous winter landscape of our dataset. This is both a benefit for the depth networks themselves, as well as for the supervisory signals since the warped image (1.3) is easier to align. It is also

intuitively easier to predict depths in urban environments as there are more features and objects, where, for example, observed sizes of things such as cars give strong depth cues, whereas trees in rural environments can have largely varying heights making for a more challenging MDE problem.

Even though the UAVid dataset is of a different environment than we are focusing on, it would still be interesting to test our models with it, though we would need access to more than just the images. We made an attempt to reach out to the curators of the UAVid dataset to inquire about the camera intrinsics as well as IMU and GPS measurements, though there was no response.

4.4 Chosen Model for PCR

Out of the trained models shown in table 4.1, we choose to use ”MiDaS finetune” for attempting point cloud registration in the next chapter. This is because it has the lowest depth errors, as well as less noisy depth estimates than the other models. In addition, it does a relatively good job of keeping the ground flat (as seen in the error-map 4.4f), which will be useful for registration.

Chapter 5

Point Cloud Registration

In this section, we perform point cloud registration for the purposes of estimating the motion of the drone, as well as attempting to localize within the 3D-model. We will use an off-the-shelf ICP algorithm and add our own simple weighting schemes, though an entire Master's Thesis could probably be written regarding finding the optimal algorithm for this task.

5.1 Setup

The generated point clouds from the depth images are expected to have significant errors caused by erroneous depth estimates. The 3D-area model is six years old, meaning we have errors with some shrubbery and trees that have since disappeared still being depicted in the model, while also having limited details since it was captured from above by airplane without including multiple viewing angles.

A simple ICP implementation as outlined in section 1.2.6 is unlikely to achieve great success, since there is no method of handling outliers and rejecting matches, which is important when working with inaccurate clouds. There are implementations of the ICP algorithms available from multiple libraries online, such as Pytorch3D or

Open3D, though these can be quite simplistic. Instead, we will use a more advanced ICP algorithm created by Birdal [5]:

- **Sampling**: Performs the ICP algorithm at $i = 10$ hierarchy levels, with an increasing number of samples from the source cloud following the pattern $n_i = \frac{n_{total}}{2^{i-1}}$ (counting i backwards). The resulting alignment at a hierarchy level is used as initial alignment on the next level. Sampling of the source cloud follows the geometrically stable sampling method by Gelfand et al. [17], while all points of the target cloud are always used.

- **Matching**: A KD-tree search is performed to find a source-point's nearest neighbor in the target cloud based on euclidean distance.

- **Weighting**: No weighing is used.

- **Rejecting**: Duplicated matches are removed, and the lowest distance one is retained. The standard deviation of the distances of the matched points is robustly estimated using the median absolute deviation[47], and matches are considered outliers and subsequently rejected if their distance is outside three standard deviations of the median.

- **Error metric**: The Point-to-Plane error metric is used. To generate normals of the clouds, we use the ten nearest neighbors for plane-fitting.

- **Minimizing the error metric**: A linear approximation as outlined by Low [32] is used to minimize the point-to-plane error. This approximation allows the problem to be solved in a single step as a simple linear least-squares problem. The ICP iterations terminate when the change between iterations is sufficiently small.

This ICP algorithm does not include a scale factor. We showed in chapter 4 that we have some scale errors in our depth estimates. When generating point clouds, we apply the median scaling with the pseudo ground-truth. While the clouds are originally represented in camera-frame (right-down-front), we transform the registration results to the front-left-up configuration for ease of interpretability.

We choose two scenes to focus on, the first in figure 5.1 and the second in figure 5.2. In these figures, we also include the registration according to the GPS-based relative motion between the scenes to act as a ground truth for comparison with our results. The red cloud is the source cloud and represents the first image. The blue cloud is the target cloud and represents the second image. This holds for all other visualized clouds as well. For both scenes, there is a gap of four images in-between to see more of a significant shift in the environment, though not far enough away to make registration too challenging.

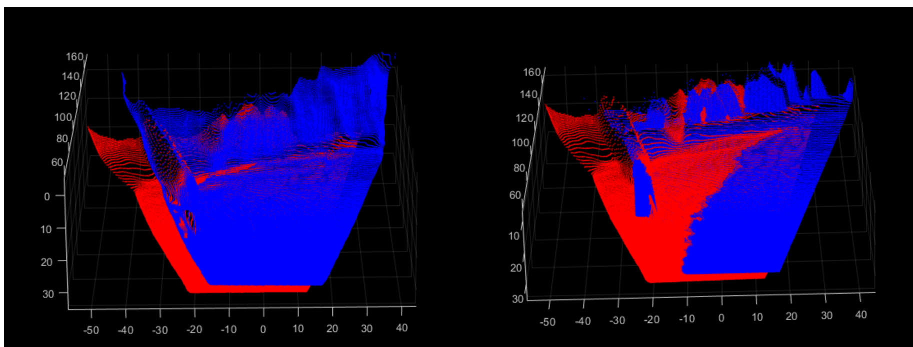
The first scene is intended to be relatively easy, as the lone tree on the left ought to be a good object for precise motion estimation and localization. The second scene is more challenging, consisting of the drone flying over the treetops. The generated point clouds will have much more bumpy surfaces and are prone to getting the ICP algorithms stuck at a local minimum.

For the first scene, we use a 175 meter depth limit for projecting the pixels to point clouds, so some of the trees in the distance can be included. For the second scene, we used a depth limit of 150 meters because we saw some obvious depth errors beyond this limit. This makes the proposed method less universally usable, though we wish to give the ICP algorithm the best chance possible to succeed for investigating the potential of our approach. In addition, we ignore the top third of the depth image to avoid erroneously backprojecting sky-pixels and crop the sides by 15 pixels. As observed in the depth-error maps in figure 4.4f, there are some increased errors on the sides of the images, and we wish to exclude them.

All point clouds are generated from images of resolution 288×512, i.e. the same as the depth estimates. This means that we downsample the pseudo ground-truth depth images from 1080×1920 to 288×512 before backprojecting (eq. 1.2). While some information will be lost as a result of this downsampling, the database is already six years out of date and has limited detail regardless, and the MDE-based point clouds will

also lack fine details of the environment. The reduced number of points in the cloud as a result of this downsampling has the beneficial effect of reducing the computational complexity of the registration.

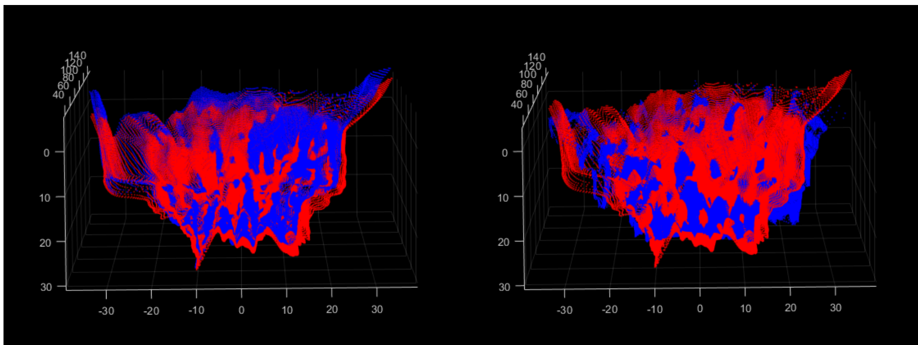
(a) First Image (b) Second Image



(c) GPS Registration between predictions (d) GPS Registration with database model

Figure 5.1: Scene 1 focuses on attempting to get accurate registrations with the use of a single tree as an easy object for matching. The red cloud is the source cloud, and the blue cloud is the target cloud.

(a) First Image (b) Second Image



(c) GPS Registration between predictions (d) GPS Registration with database model

Figure 5.2: Scene 2 is a more challenging situation where many trees are involved. The red cloud is the source cloud, and the blue cloud is the target cloud.

Adding to the ICP algorithm above, we will also test two weighting schemes:

- Normal-compatibility weighing, $w_i = n_{p,i} \cdot n_{q,i}$
- Cross-product with the averaged normal of the target cloud, $w_i = n_{q,i} \times n_{q,avg}$

The normal compatibility reduces the weight of point matches with differing normals, meaning that matches where the surfaces around the points differ are assumed to be more likely to be erroneous.

Inspired by the normal-compatibility weighting, which uses the dot-product, we implement a simple cross-product scheme for some specific parts of our dataset. It consists of finding an average normal vector of the target cloud and setting the weight of a matched pair as the cross-product between this average and the normal of the matched point of the target. The reasoning is that sometimes, large parts of a cloud are flat ground with few trees included. In this case, it is crucial to correctly align the trees to find the true registration. However, the majority of points are on the ground, such that the ICP algorithm prioritizes aligning the flat surfaces instead of the points consisting of the trees. While using the point-to-plane error metric alleviates this problem somewhat since it allows the algorithm to translate and rotate the surfaces along each other without increasing the error [17], this proposed weighting scheme can further address this issue. We only apply this to the first scene, as the second scene does not consist of flat surfaces. This selective application of the weighting scheme makes the algorithm unfitting for general use, but our goal is to investigate the best-case potential of ICP for this task.

An important aspect to consider is that even if the estimated pose differs from the GPS readings, this does not mean that the ICP algorithm failed to converge to the global optimum. The ICP implementation might do the job perfectly, but if the generated point clouds are inaccurate, the results may not align with the GPS. In addition, the GPS pseudo ground-truth will be somewhat inaccurate. So while high-precision results are unlikely, both because of MDE errors and the limited database details, we still hope we can see some degree of success within localization and ego-motion estimation.

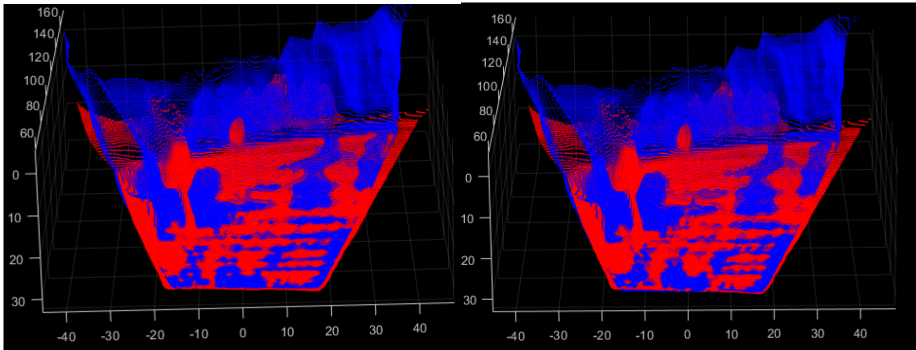
5.2 Estimating ego-motion

Our first test consists of estimating drone ego-motion using the generated point clouds from two images. Figure 5.3 shows registration results of the first scene using the three weighting variants. As we see, only the cross-product weighting scheme manages to align the lone tree to an extent. Unfortunately, the overall result is not at all close to the GPS readings. With the other two registration attempts we barely see any movement. We assume a large reason for this is that the ”ground” of the point clouds is slightly curved and not as flat as they are in real life. While the point-to-plane error metric allows for flat surfaces to glide along each other since only the point-distances from the plane are minimized, the curved surfaces incentivize the algorithm to keep the clouds right on top of each other such that the curve fits well. This systemic problem of our depth estimator makes accurate motion estimation through ICP very challenging. As we see in figure 5.3c, this can be alleviated with the proper weighing scheme. The cross-product weighing scheme helps offset this issue, as matches to points on the ground of the target cloud are weighted down. Unfortunately, the source cloud moved too far, and we see a massive error in the y-axis translation. On the other hand, some of the artifacts of the depth estimation can be useful for registration. For instance, the trails of points around the lone tree in scene 1 give a good marker for finding the rotational component of the pose, as we see them properly aligned with the GPS-registration in figure 5.1c, though our registration failed in this regardless.

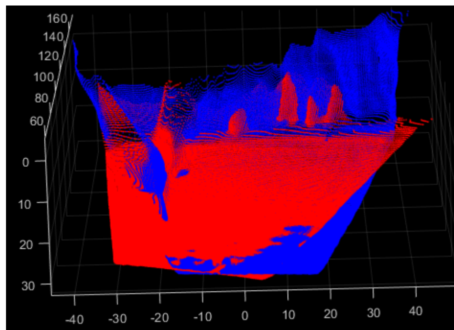
The trees in the rear of the cloud appear to cause issues for the registration, as they are most visible in the target cloud where the drone has moved further ahead. This is a common problem we will inevitably encounter when attempting this kind of task, and should ideally be handled using rejection schemes since points not present in both clouds are considered outliers. However, when both of the clouds are inaccurate and noisy, it is much more challenging to detect these specific points as outliers and subsequently ignore them. The cross-weighting registration (5.3c) shows that the algorithm converged to aligning the rearmost trees quite erroneously as compared to the GPS-registration.

Overall, the estimated ego-motion is quite poor for scene 1, which is quite unfortunate, as we had hoped this would be one of the easier situations.

We show the results of ego-motion estimation of scene 2 in figure 5.4. As mentioned before, we do not attempt cross-product weighting since these clouds do not consist of large flat surfaces. This is a much more challenging estimation problem because the bumpy depth-surfaces of numerous treetops generate several local minima for the optimizer to converge towards. The point-to-plane error metric is generally superior to point-to-point, though the advantage is diminished in unstructured environments[42]. Point-to-plane works better for the first example when large parts of the cloud consists of flat ground, and it is likely not much better than a point-to-point error metric in this case. Somewhat surprisingly, the estimated motion compared to the GPS are not clearly worse than in scene 1. The forward motion is captured quite well, and the elevation to a lesser extent, though the other variables are much less accurate. It is hard to say how good the registration is simply by looking at the point clouds since the scene is very cluttered. The results of scene 1 look clearly wrong because of the failure to align the tree, whereas it is difficult to say in scene 2.



(a) No weighting (b) Normal-compatibility weighting

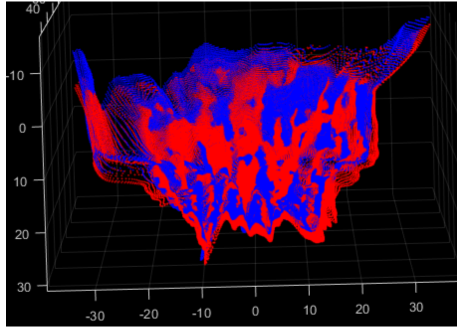


(c) Cross-normal weighting

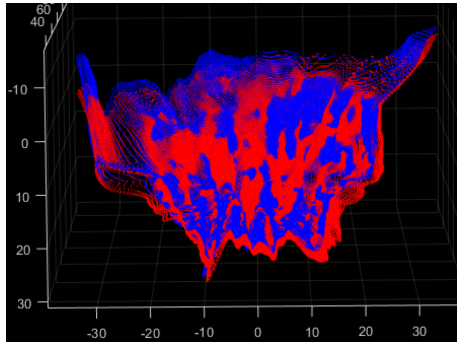
DOF	GPS	(a)	(b)	(c)
$\phi(^\circ)$	-0.390	0.145	0.043	-2.813
$\theta(^\circ)$	-0.556	-2.321	-2.461	-1.255
$\psi(^\circ)$	-3.389	1.893	3.192	9.865
x(m)	2.533	2.507	2.818	5.424
y(m)	-1.241	-3.026	-4.998	-23.293
z(m)	0.215	-3.909	-4.085	-2.546

(d) Motion comparison

Figure 5.3: Resulting ego-motion registration of the first scene. Observing the alignment in the table compared to GPS, it looks like ICP managed to correctly capture the forward motion of the drone for a) and b) with only a slight inaccuracy, though the five other degrees of freedom are incorrect. While c) has attempted to align the trees, the overall registration is incorrect.



(a) No weighting



(b) Normal-compatibility weighting

DOF	**GPS**	**(a)**	**(b)**
$\phi(^\circ)$	-0.070	1.912	2.056
$\theta(^\circ)$	0.138	-0.018	0.064
$\psi(^\circ)$	0.088	0.371	0.359
x(m)	2.722	3.108	3.034
y(m)	0.059	-0.762	-0.783
z(m)	0.526	0.362	0.474

(c) Motion comparison

Figure 5.4: Ego-motion registration results of the second scene. Only the forward motion and elevation change seem somewhat accurate, though it is very challenging to make out the details of the registration. Results are also mostly unchanged when using the normal-compatibility weighting.

5.3 Localization

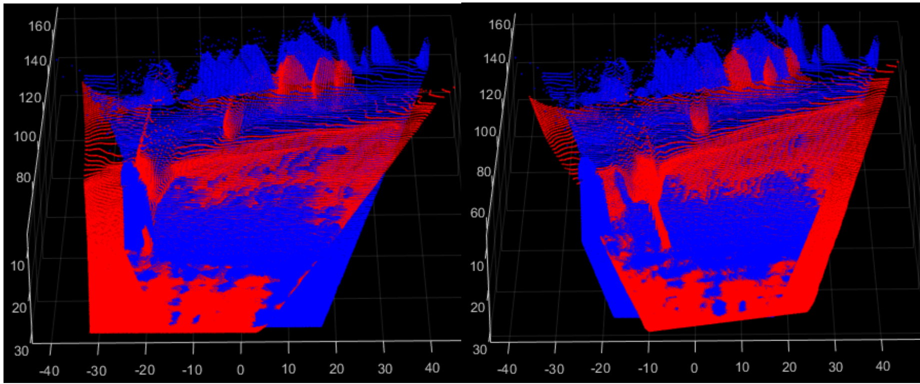
We make several attempts of matching with the area model. We perform the matching the same way as in the previous section, but instead use the pseudo ground-truth depth images to generate the target cloud. Should the registration succeed, we will have found the drone position relative to a known position from which the target cloud was generated. The first scene is shown in figure 5.5, and the second in figure 5.6.

When matching with the area model in scene 1, we see that the ICP algorithm has an easier time aligning the lone tree. Unlike the predicted depth, the pseudo ground-truth depth does not experience curvature in the ground. This lets the point-to-plane error metric show its benefit of sliding the surfaces to allow for an attempted alignment of the tree. Unfortunately, figures 5.5a and 5.5c show that the source cloud moved too far, with a similar result as we saw in figure 5.3c. The normal-compatibility weighting scheme showed some of the worst results yet, actively moving the trees away from each other. A reason for this can be inferred in figure 5.7. The tree will have very different normals in the target cloud compared to the source cloud, and this means that the weighting will deprioritize aligning potential matched points of the tree.

As in the previous section on ego-motion, the two clouds of scene 2 are not easily aligned. In fact, while most other registrations have correctly captured the forward motion of the drone, this registration does not do well in any of the six degrees of freedom.

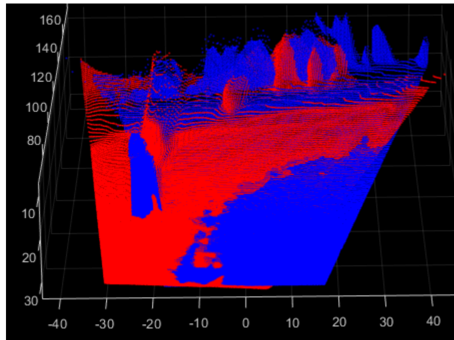
None of the weighting schemes in the two scenes achieved great success compared to the GPS-motion. As we have seen in the previous chapter, the pseudo ground-truth has limited detail and is somewhat inaccurate. Additionally, none of the predicted depths managed to generate clouds as jagged as the area model, and the trees are instead very rounded and not straight. Inspecting figure 5.7, there is a clear difference in how a tree is represented in the two point clouds. In light of this, it is unsurprising that the ICP algorithm failed to find relative motion similar to the GPS. The skewed tree of the red cloud is from the MiDaS depth prediction and can be caused by the fact that the drone

looks downward at the tree. In figure 4.3f it looks like the tree is estimated as having the same depth in the image at both the top and bottom, but it ought to be closer at the top. There could also be a bias of the depth model generally assuming sections closer to the bottom of the image to be closer.



(a) No weighting

(b) Normal-compatibility weighting

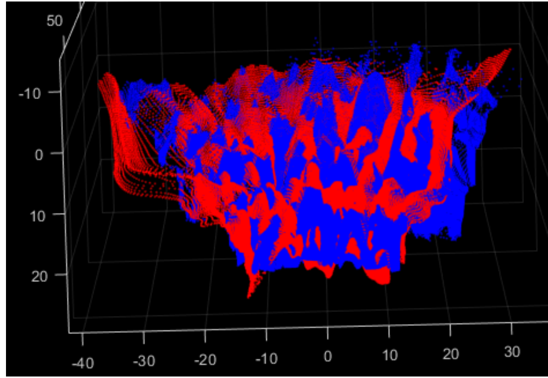


(c) Cross-normal weighting

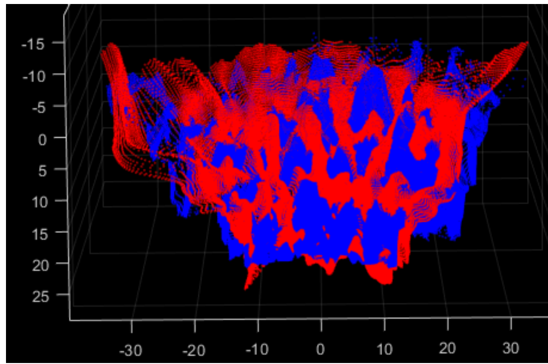
DOF	**GPS**	**(a)**	**(b)**	**(c)**
$\phi(^\circ)$	-0.390	6.175	7.464	2.551
$\theta(^\circ)$	-0.556	-3.106	-3.201	-1.251
$\psi(^\circ)$	-3.389	11.838	-0.063	9.558
x(m)	2.533	5.797	2.512	2.773
y(m)	-1.241	-30.583	2.611	-25.840
z(m)	0.215	-6.778	-3.940	-2.504

(d) Motion comparison

Figure 5.5: Attempted localization of the first scene. As we have seen with other ICP runs, only the x-axis translation is mostly correct, with the exception of a). While it is easier to align the tree when using the 3D model, the overall registration result is poor.



(a) No weighting



(b) Normal-compatibility weighting

DOF	GPS	(a)	(b)
$\phi(^\circ)$	-0.070	-3.697	-5.888
$\theta(^\circ)$	0.138	-2.467	-0.757
$\psi(^\circ)$	0.088	-1.452	-1.246
x(m)	2.722	-1.692	-1.489
y(m)	0.059	0.545	2.386
z(m)	0.526	-2.717	-1.057

(c) Registration comparison

Figure 5.6: Attempted localization of the second scene. The database model and the point cloud generated by the depth estimate are significantly different, giving poor localization results.

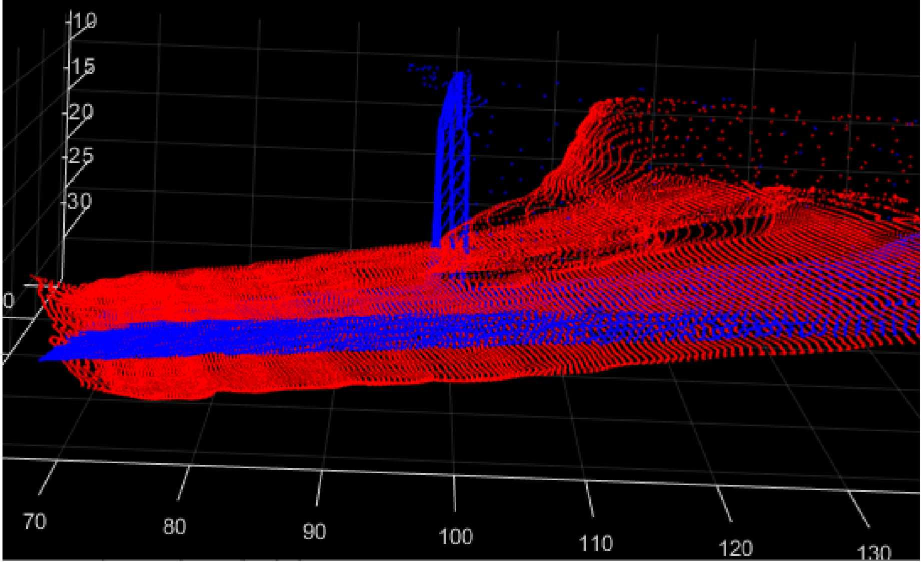


Figure 5.7: Comparison of how the tree is modeled in the 3D database and how the MDE model captures it. They are very dissimilar, causing the failure to use it as a distinct landmark for localization, especially when using normal-compatibility weighting.

5.4 Overall results

Neither motion estimation nor localization saw much success, not even for the first scene that was cherry-picked to hopefully be simple. Additionally, we made specific choices to aid in improving chances of good alignments: manually selecting depth levels for the two scenes instead of having a fixed one, and adding a simple additional weighting scheme to only the first scene. These decisions were taken to find the best-case scenario for our proposed method and to compensate for the off-the-shelf ICP algorithm that was not specifically designed for the task, though it makes our method less generalized.

As shown, neither the depth networks nor the area model has properly captured

the structure of the trees correctly, and these discrepancies make it nearly impossible to estimate the precise relative pose between the clouds, though some rough alignment is possible, as seen in figures 5.3c, 5.5c and 5.5a. If a state-of-the-art visual localization method were attempted with a similarly old and low-detailed database like ours, it would likely have done poorly, perhaps even worse than ours.

Chapter 6

Conclusion

As we have seen in this thesis, the task of self-supervised depth estimation from UAVs is a challenging one. Using several state-of-the-art MDE models, we achieved varying results, though none of them good enough to use for precise localization and navigation. While the results not sufficient to compete with GPS, we still see some promise in being able to roughly align ourselves with a prominent landmark for localization.

All fault cannot be placed on the depth estimates, as the 3D-model had limited detail, such that even a perfect depth prediction might not yield a precise estimate. So while the results we achieved here with our approach were not very good, other visual localization methods would likely have struggled if such an old and low-detail database was used. In addition, the environment we were working with is known to be difficult, which was why we wanted to investigate this unusual approach to visual localization without feature descriptors and point matching.

Future work should consist of gathering more data that is of higher quality, i.e., good lighting, smooth and simple movement, consistent FPS and sampling. Parameter-tuning of the depth estimation models(loss-weighting, learning-rate, EKF-tuning, etc.) may improve performance, though replacing the simple pose-estimation network with

a better and pre-trained one could be a significant boon. The ICP algorithm has much room for improvement, as optimizing ICP for large clouds of forested environments with the presence of outliers and noise could be an entire Master's Thesis by itself, and we did not have much time to focus on it here. An alternate solution could be to use an AI-based PCR model instead of only focusing on the ICP algorithm.

Despite the unimpressive results presented here, it is too early to conclude if this approach is infeasible. While the field of visual localization has had decades of research and optimization, our method is new and has much room for further improvement. As deep learning continues to improve in this period of booming AI research, our attempted localization pipeline can be revisited, as it seems current self-supervised depth estimation networks are insufficient for the task.

References

[1] Nicolas Andreff, Radu Horaud, and Bernard Espiau. Robot hand-eye calibration using structure-from-motion. *The International Journal of Robotics Research*, 20 (3):228–248, 2001. URL https://doi.org/10.1177/02783640122067372.

[2] Hangbo Bao, Li Dong, Songhao Piao, and Furu Wei. Beit: Bert pre-training of image transformers, 2022. URL https://arxiv.org/abs/2106.08254.

[3] Paul J. Besl and Neil D. McKay. A method for registration of 3-d shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, 1992. URL https://ieeexplore.ieee.org/document/121791.

[4] Amlaan Bhoi. Monocular depth estimation: A survey, 2019. URL https://arxiv.org/abs/1901.09402.

[5] Tolga Birdal. Icp registration using efficient variants and multi-resolution scheme, 2014. URL https://www.mathworks.com/matlabcentral/fileexchange/47152-icp-registration-using-efficient-variants-and-multi-resolution-scheme.

[6] Eric Brachmann and Carsten Rother. Visual camera re-localization from rgb and rgb-d images using dsac, 2020. URL https://ieeexplore.ieee.org/document/9394752.

[7] Shu Chen, Zhengdong Pu, Xiang Fan, and Beiji Zou. Fixing defect of photometric loss for self-supervised monocular depth estimation. *IEEE Transac-*

tions on Circuits and Systems for Video Technology, 32(3):1328–1338, 2022. URL https://ieeexplore.ieee.org/document/9386100.

[8] Yang Chen and Gérard Medioni. Object modeling by registration of multiple range images. In *Proceedings. 1991 IEEE International Conference on Robotics and Automation*, pages 2724–2729 vol.3, 1991. URL https://ieeexplore.ieee.org/document/132043.

[9] Javier Civera, Andrew J. Davison, and J. M. MartÍnez Montiel. Inverse depth parametrization for monocular slam. *IEEE Transactions on Robotics*, 24(5):932–945, 2008. URL https://ieeexplore.ieee.org/document/4637878.

[10] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2020. URL https://arxiv.org/abs/2010.11929.

[11] David Eigen, Christian Puhrsch, and Rob Fergus. Depth map prediction from a single image using a multi-scale deep network, 2014. URL https://arxiv.org/abs/1406.2283.

[12] Jacob Engel, Vladlen Koltun, and Daniel Cremers. Direct sparse odometry. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2018. URL https://ieeexplore.ieee.org/document/7898369.

[13] Jakob Engel, Thomas Schöps, and Daniel Cremers. Lsd-slam: Large-scale direct monocular slam. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 834–849, Cham, 2014. Springer International Publishing. ISBN 978-3-319-10605-2. URL https://link.springer.com/chapter/10.1007/978-3-319-10605-2_54.

[14] Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography.

Commun. ACM, 24:381–395, 1981. URL https://dl.acm.org/doi/10.1145/358669.358692.

[15] Michaël Fonder, Damien Ernst, and Marc Van Droogenbroeck. M4depth: Monocular depth estimation for autonomous vehicles in unseen environments, 2021. URL https://arxiv.org/abs/2105.09847.

[16] Huan Fu, Mingming Gong, Chaohui Wang, Kayhan Batmanghelich, and Dacheng Tao. Deep ordinal regression network for monocular depth estimation, 2018. URL https://arxiv.org/abs/1806.02446.

[17] N. Gelfand, L. Ikemoto, S. Rusinkiewicz, and M. Levoy. Geometrically stable sampling for the icp algorithm. In *Fourth International Conference on 3-D Digital Imaging and Modeling, 2003. 3DIM 2003. Proceedings.*, pages 260–267, 2003. doi: 10.1109/IM.2003.1240258. URL https://ieeexplore.ieee.org/document/1240258.

[18] Clément Godard, Oisin Mac Aodha, Michael Firman, and Gabriel Brostow. Digging into self-supervised monocular depth estimation, 2018. URL https://arxiv.org/abs/1806.01260.

[19] Trym Vegard Haavardsholm. Ttk21 - introduction to visual slam; lecture 2: 3d-geometry, 07 09 2022.

[20] Rostam Affendi Hamzah and Haidi Ibrahim. Literature survey on stereo vision disparity map algorithms. *Journal of Sensors*, 2016, 2016. URL https://www.hindawi.com/journals/js/2016/8742920/.

[21] Kai Han, Yunhe Wang, Hanting Chen, Xinghao Chen, Jianyuan Guo, Zhenhua Liu, Yehui Tang, An Xiao, Chunjing Xu, Yixing Xu, Zhaohui Yang, Yiman Zhang, and Dacheng Tao. A survey on vision transformer. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(1):87–110, 2023. URL https://ieeexplore.ieee.org/abstract/document/9716741.

[22] Hassan Hassan, Abdelazim Negm, Mohamed Zahran, and Oliver Saavedra. Assessment of artificial neural network for bathymetry estimation using high resolution satellite imagery in shallow lakes: Case study el burullus lake. *International Water Technology Journal*, 5, 12 2015. URL https://www.researchgate.net/figure/A-hypothetical-example-of-Multilayer-Perceptron-Network_fig4_303875065.

[23] Simen Haugo. Ttk4255: Robotic vision, homework 3: Geometric image formation, 2022.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. URL https://arxiv.org/abs/1512.03385.

[25] Dirk Holz, Alexandru Ichim, Federico Tombari, Radu Rusu, and Sven Behnke. Registration with the point cloud library - a modular framework for aligning in 3-d. *IEEE Robotics & Automation Magazine*, 22:110–124, 2015. URL https://sci-hub.se/10.1109/mra.2015.2432331.

[26] Michal Irani and Prabu Anandan. About direct methods. In Bill Triggs, Andrew Zisserman, and Richard Szeliski, editors, *Vision Algorithms: Theory and Practice*, pages 267–277, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-44480-0. URL https://link.springer.com/chapter/10.1007/3-540-44480-7_18.

[27] Youngwan Lee, Jonghee Kim, Jeff Willette, and Sung Ju Hwang. Mpvit: Multi-path vision transformer for dense prediction, 2021. URL https://arxiv.org/abs/2112.11010.

[28] Yawei Li, Kai Zhang, Jiezhang Cao, Radu Timofte, and Luc Van Gool. Localvit: Bringing locality to vision transformers, 2021. URL https://arxiv.org/abs/2104.05707.

[29] Zhenyu Li, Xuyang Wang, Xianming Liu, and Junjun Jiang. Binsformer: Revisiting adaptive bins for monocular depth estimation, 2022. URL https://arxiv.org/abs/2204.00987.

[30] Guosheng Lin, Anton Milan, Chunhua Shen, and Ian Reid. Refinenet: Multi-path refinement networks for high-resolution semantic segmentation, 2016. URL https://arxiv.org/abs/1611.06612.

[31] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows, 2021. URL https://arxiv.org/abs/2103.14030.

[32] Kok-Lim Low. Linear least-squares optimization for point-to-plane icp surface registration. 2004. URL https://www.researchgate.net/publication/228571031_Linear_Least-Squares_Optimization_for_Point-to-Plane_ICP_Surface_Registration.

[33] David Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–, 11 2004. URL https://link.springer.com/article/10.1023/B:VISI.0000029664.99615.94.

[34] Wenjie Luo, Yujia Li, Raquel Urtasun, and Richard Zemel. Understanding the effective receptive field in deep convolutional neural networks. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL https://proceedings.neurips.cc/paper_files/paper/2016/file/c8067ad1937f728f51288b3eb986afaa-Paper.pdf.

[35] Ye Lyu, George Vosselman, Gui-Song Xia, Alper Yilmaz, and Michael Ying Yang. Uavid: A semantic segmentation dataset for uav imagery. *ISPRS Journal of Photogrammetry and Remote Sensing*, 165:108–119, 2020. ISSN 0924-2716. doi: https://doi.org/10.1016/j.isprsjprs.2020.05.009. URL https://www.sciencedirect.com/science/article/pii/S0924271620301295.

[36] Logambal Madhuanand, Francesco Nex, and Michael Ying Yang. Self-supervised monocular depth estimation from oblique uav videos. *ISPRS Journal of Photogrammetry and Remote Sensing*, 176:1–14, 2021. ISSN 0924-2716. doi: https:

//doi.org/10.1016/j.isprsjprs.2021.03.024. URL https://www.sciencedirect.com/science/article/pii/S0924271621000952.

[37] Armin Masoumian, Hatem A. Rashwan, Julián Cristiano, M. Salman Asif, and Domenec Puig. Monocular depth estimation using deep learning: A review. *Sensors*, 22(14), 2022. ISSN 1424-8220. URL https://www.mdpi.com/1424-8220/22/14/5353.

[38] Yue Ming, Xuyang Meng, Chunxiao Fan, and Hui Yu. Deep learning for monocular depth estimation: A review. *Neurocomputing*, 438:14–33, 2021. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2020.12.089. URL https://www.sciencedirect.com/science/article/pii/S0925231220320014.

[39] Will Nash, Tom Drummond, and Nick Birbilis. A review of deep learning in the study of materials degradation. *npj Materials Degradation*, 2, 2018. URL https://www.nature.com/articles/s41529-018-0058-x.

[40] Semih Orhan and Yalın Baştanlar. Efficient search in a panoramic image database for long-term visual localization. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*, pages 1727–1734, 2021. URL https://openaccess.thecvf.com/content/ICCV2021W/ACVR/html/Orhan_Efficient_Search_in_a_Panoramic_Image_Database_for_Long-Term_Visual_ICCVW_2021_paper.html.

[41] Rui Peng, Ronggang Wang, Yawen Lai, Luyang Tang, and Yangang Cai. Excavating the potential capacity of self-supervised monocular depth estimation, 2021. URL https://arxiv.org/abs/2109.12484.

[42] François Pomerleau, Francis Colas, Roland Siegwart, and Stéphane Magnenat. Comparing icp variants on real-world data sets. *Autonomous Robots*, 2013. URL https://www.researchgate.net/publication/235631701_Comparing_ICP_variants_on_real-world_data_sets.

[43] François Pomerleau, Francis Colas, and Roland Siegwart. A review of point cloud registration algorithms for mobile robotics. *Foundations and Trends® in*

Robotics, 4:1–104, 2015. URL https://www.researchgate.net/publication/277558596_A_Review_of_Point_Cloud_Registration_Algorithms_for_Mobile_Robotics.

[44] René Ranftl, Katrin Lasinger, David Hafner, Konrad Schindler, and Vladlen Koltun. Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer, 2019. URL https://arxiv.org/abs/1907.01341.

[45] René Ranftl, Alexey Bochkovskiy, and Vladlen Koltun. Vision transformers for dense prediction, 2021. URL https://arxiv.org/abs/2103.13413.

[46] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015. URL https://arxiv.org/abs/1505.04597.

[47] Peter J. Rousseeuw and Christophe Croux. Alternatives to the median absolute deviation. *Journal of the American Statistical Association*, 88(424): 1273–1283, 1993. URL https://www.tandfonline.com/doi/abs/10.1080/01621459.1993.10476408.

[48] Szymon Rusinkiewicz and Marc Levoy. Efficient variants of the icp algorithm. In *Proceedings Third International Conference on 3-D Digital Imaging and Modeling*, pages 145–152, 2001. URL https://ieeexplore.ieee.org/document/924423.

[49] Espen A. Sande. Ttk4550: Specization project. 2022. URL https://www.dropbox.com/s/3uwfrzwxtpk4c5b/specialization.pdf?dl=0.

[50] Torsten Sattler, Michal Havlena, Konrad Schindler, and Marc Pollefeys. Large-scale location recognition and the geometric burstiness problem. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1582–1590, 2016. URL https://ieeexplore.ieee.org/document/7780544.

[51] Ashutosh Saxena, Sung H. Chung, and Andrew Y. Ng. Learning depth from single monocular images. In *Proceedings of the 18th International Conference on Neural Information Processing Systems*, NIPS'05, page 1161–1168, Cambridge, MA, USA, 2005. MIT Press. URL https://dl.acm.org/doi/10.5555/2976248.2976394.

[52] Ashutosh Saxena, Jamie Schulte, Andrew Y Ng, et al. Depth estimation using monocular and stereo cues. In *IJCAI*, volume 7, pages 2197–2203, 2007. URL https://ai.stanford.edu/~ang/papers/DepthEstimation_Saxena_IJCAI.pdf.

[53] Aleksandr Segal, Dirk Hähnel, and Sebastian Thrun. Generalized-icp. 2009. URL https://www.semanticscholar.org/paper/Generalized-ICP-Segal-H%C3%A4hnel/b352b3a7f1068b2d562ba12a446628397dfe8a77.

[54] Tianwei Shen, Lei Zhou, Zixin Luo, Yao Yao, Shiwei Li, Jiahui Zhang, Tian Fang, and Long Quan. Self-supervised learning of depth and motion under photometric inconsistency, 2019. URL https://arxiv.org/abs/1909.09115.

[55] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014. URL https://arxiv.org/abs/1409.1556.

[56] Olga Sorkine-Hornung and Michael Rabinovich. Least-squares rigid motion using svd. *[Online]*, 2017. URL https://igl.ethz.ch/projects/ARAP/svd_rot.pdf.

[57] Richard Szeliski. *Computer Vision: Algorithms and Applications, 2nd ed.* Springer, 2022.

[58] Marko Teittinen. Depth cues in the human visual system. *The encyclopedia of virtual environments*, 1, 1993. URL http://www.hitl.washington.edu/projects/knowledge_base/virtual-worlds/EVE/III.A.1.c.DepthCues.html.

[59] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017. URL https://arxiv.org/abs/1706.03762.

[60] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017. URL https://arxiv.org/abs/1706.03762.

[61] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions*

on Image Processing, 13(4):600–612, 2004. URL https://ieeexplore.ieee.org/document/1284395.

[62] Jamie Watson, Michael Firman, Gabriel J. Brostow, and Daniyar Turmukhambetov. Self-supervised monocular depth hints. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019. URL https://arxiv.org/abs/1909.09051.

[63] Jamie Watson, Oisin Mac Aodha, Victor Prisacariu, Gabriel Brostow, and Michael Firman. The temporal opportunist: Self-supervised multi-frame monocular depth, 2021. URL https://arxiv.org/abs/2104.14540.

[64] Weihao Yuan, Xiaodong Gu, Zuozhuo Dai, Siyu Zhu, and Ping Tan. New crfs: Neural window fully-connected crfs for monocular depth estimation, 2022. URL https://arxiv.org/abs/2203.01502.

[65] Sen Zhang, Jing Zhang, and Dacheng Tao. Towards scale-aware, robust, and generalizable unsupervised monocular depth estimation by integrating imu motion dynamics, 2022. URL https://arxiv.org/abs/2207.04680.

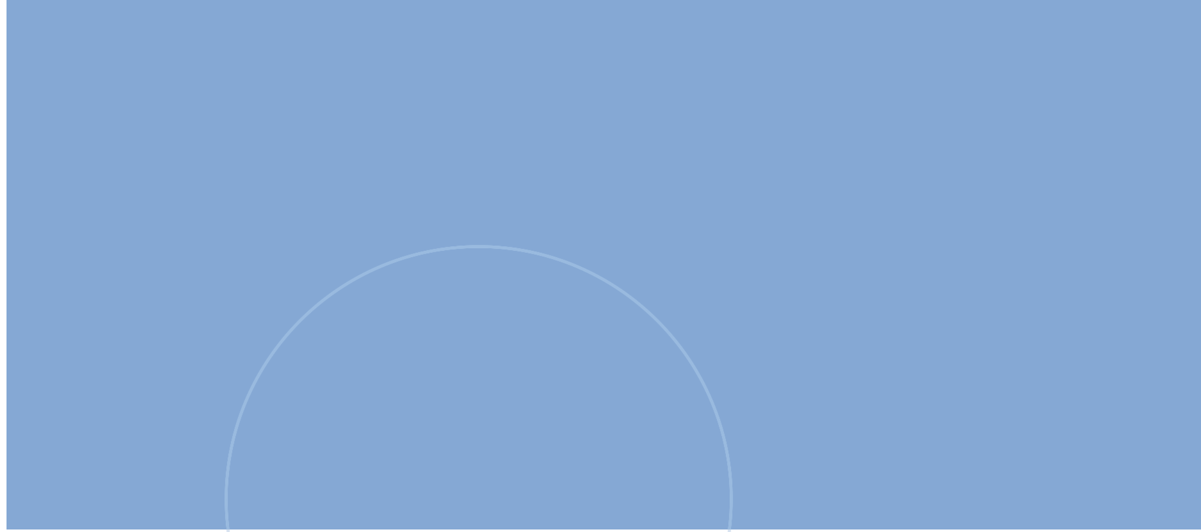
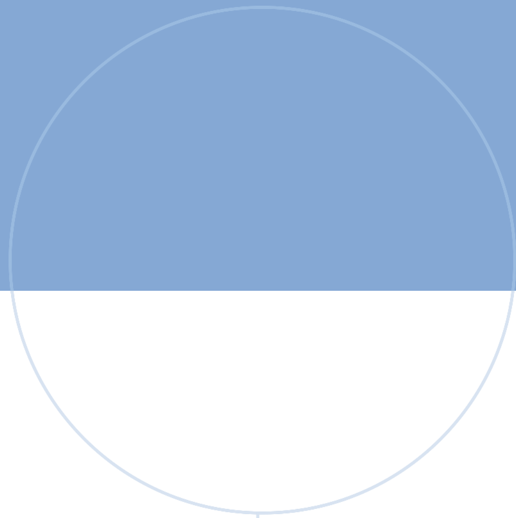
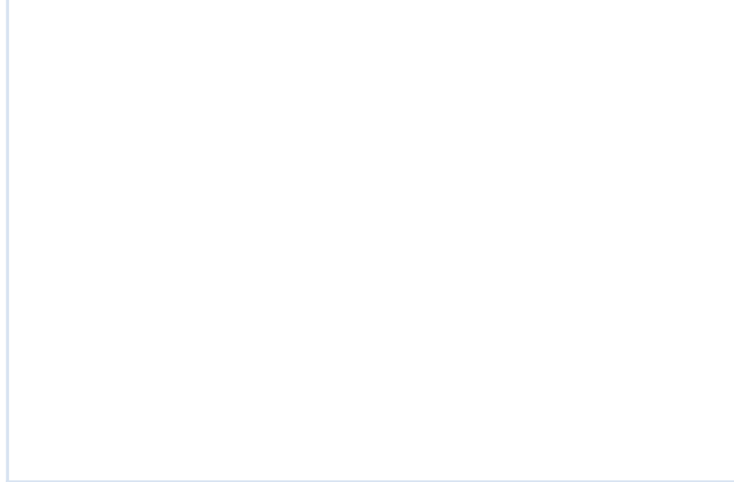
[66] Zhenghong Zhang, Mingkang Xiong, and Huilin Xiong. Monocular depth estimation for uav obstacle avoidance. In *2019 4th International Conference on Cloud Computing and Internet of Things (CCIOT)*, pages 43–47, 2019. URL https://ieeexplore.ieee.org/document/8980350.

[67] Zichao Zhang, Torsten Sattler, and Davide Scaramuzza. Reference pose generation for long-term visual localization via learned features and view synthesis. *International Journal of Computer Vision*, 129(4):821–844, 12 2020. URL https://doi.org/10.1007%2Fs11263-020-01399-8.

[68] Chaoqiang Zhao, Youmin Zhang, Matteo Poggi, Fabio Tosi, Xianda Guo, Zheng Zhu, Guan Huang, Yang Tang, and Stefano Mattoccia. MonoViT: Self-supervised monocular depth estimation with a vision transformer. In *2022 International Conference on 3D Vision (3DV)*. IEEE, sep 2022. URL https://doi.org/10.1109%2F3dv57658.2022.00077.

[69] Hao Zhu. Image retrieval-based localization under seasonal changes. In *2022 IEEE 2nd International Conference on Computer Communication and Artificial Intelligence (CCAI)*, pages 142–148, 2022. URL https://ieeexplore.ieee.org/abstract/document/9807825?casa_token=S1v2MoODQicAAAAA:1r9dOnpQtYuv42t0g_S3CxkFMUgggzfWzpojJWwJmJyqLld2qyOcKQny-O8jp3oqaJDbiXZvuA.

[70] Shuangquan Zuo, Yun Xiao, Xiaojun Chang, and Xuanhong Wang. Vision transformers for dense prediction: A survey. *Knowledge-Based Systems*, 253:109552, 2022. ISSN 0950-7051. URL https://www.sciencedirect.com/science/article/pii/S0950705122007821.

Norwegian University of
Science and Technology