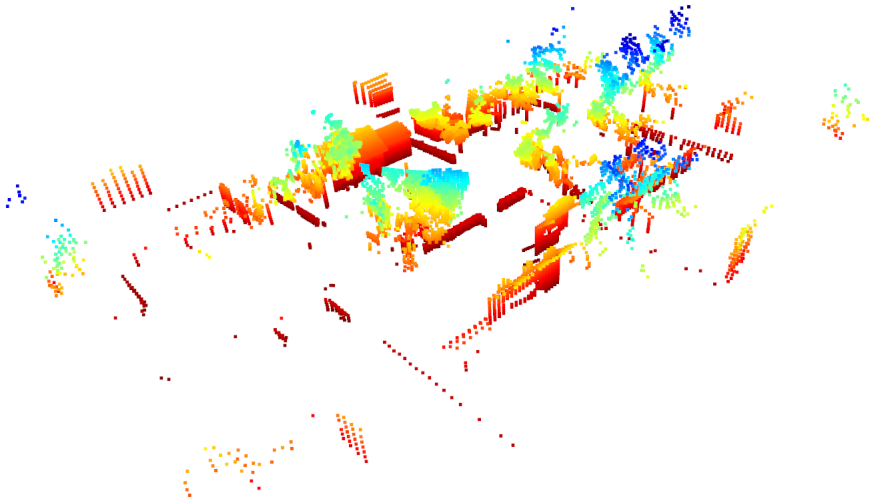


Outdoor global pose estimation from RGB and 3D data

Master's thesis in Systems, Control and Mechatronics

SIMON BASTÅS
ROBERT BRENICK

Department of Mechanics and Maritime Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

Master's thesis 2019:14

Outdoor global pose estimation from RGB and 3D data

SIMON BASTÅS
ROBERT BRENICK

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Mechanics and Maritime Sciences
Division of Vehicle Engineering and Autonomous Systems
Chalmers University of Technology
Gothenburg, Sweden 2019

Outdoor global pose estimation from RGB and 3D data
SIMON BASTÅS
ROBERT BRENICK

Supervisor: Yaowen Xu, CPAC Systems AB
Examiner: Peter Forsberg, Mechanics and Maritime Sciences

Master's Thesis 2019:14
Department of Mechanics and Maritime Sciences
Division of Vehicle Engineering and Autonomous Systems
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Lidar scan generated from the simulation environment CARLA.

Typeset in LaTeX
Gothenburg, Sweden 2019

Outdoor global pose estimation from RGB and 3D data
SIMON BASTÅS
ROBERT BRENICK
Department of Mechanics and Maritime Sciences
Chalmers University of Technology

Abstract

Accurate and robust localization is crucial for safe and reliable autonomous vehicles. In this thesis we present a machine learning approach for localization using RGB images and 3D structure information. More specifically we use two types of neural networks, one that predicts a pose in an given environment with a RGB-D image as input and one that uses a 3D LiDAR scan and a prebuilt 3D map of the environment to refine the prediction. The advantages of using neural networks for localization is the constant runtime and that it use natural navigation, i.e. without the need of infrastructure enhancement.
We show that the RGB-D network gains increased accuracy from the multi-modal RGB-D data, compared to an uni-modal network trained on RGB or depth images when used in outdoor scenes. Additionally, we show the benefit of using 3D LiDAR data for pose refinement.

Keywords: Localization, RGB-D, Neural networks, Machine learning, Pose estimation, Point cloud registration.

Acknowledgements

We would like to take this opportunity to thank our examiner Peter Forsberg and supervisor Yaowen Xu for their guidance and continuous support during the project. We would also like to thank CPAC Systems AB and their employees for hosting us and making us feel very welcomed. We would also like to thank Annika Lundqvist and Sabina Linderoth for sharing their insights about the simulator CARLA.

Simon Bastås and Robert Brenick, Gothenburg, May 2019

Thesis advisor: Yaowen Xu, CPAC Systems AB
Thesis examiner: Peter Forsberg, Chalmers

Abbreviations

NN .....Neural Network
CNN.....Convolutional Neural Network
FFNN.....Feed Forward Neural Network
MLP.....Multi-Layer Perceptron
LK .....Lucas-Kanade
LiDAR .....Light detection and ranging
FOV .....Field of View
RGB .....Red, Green, Blue. Used for color images
VO .....Visual Odometry
ICP .....Iteratvie Closest Point

Contents

List of Figures

List of Tables

1
Introduction

Autonomous robots have attracted a lot of attention from researchers over the past decades. More recently there has also been an increased focus from the commercial sector, both spawning new companies trying to disrupt current markets and old ones investing heavily into autonomy to keep its product offering relevant in the future [1][2][3].

One of the major fields in robotics is localization, i.e. the ability for a robot to position itself in the environment it is operating in. Notable ways of localizing are the use of inertial navigation by dead reckoning [4] and various global navigation satellite systems (GNSS). Inertial measurements are often fast and accurate but since it has no external reference to the world, they are cursed with integration drift. GNSS systems solve this by keeping track of satellites and triangulating the receiver's position with these for external references. This is however slower, more inaccurate and do not work with obstructed line of sight between receiver and satellites.

In more recent work, localization solutions that try to solve the above issues have been introduced. Visual odometry (VO) and a wide set of simultaneous localization and mapping (SLAM) algorithms [5][6] are a few of these. Commonly, visual systems navigate by detecting salient key-features in the environment and does not depend on external beacons sending signals that can be blocked or distorted. VO is the process of estimating the ego-motion of an agent by estimating the relative pose between camera frames. SLAM is the process of localizing with (but not exclusive to) a system such as VO, while simultaneously building a map of the environment. The map is used to maintain global consistency between the estimated camera trajectory and the map, which reduces global drift in the localization.

A great amount of research has recently been done on object detection and feature learning using RGB-D data [8]. This development can be extensively credited to the release of the Microsoft Kinect [7] sensor in 2010, making RGB-D sensors available at consumer prices. RGB-D sensors fuse the information from a ranging sensor with the image from a camera to map a depth value to each pixel in the image. However, these early RGB-D sensors have not been capable of producing accurate depth maps at larger distances, typically not more than 5-6 meters. So the great majority of research with these sensors have been conducted in indoor settings.

Light detection and ranging (LiDAR) sensors sense the time-of-flight of reflected laser pulses, produced and received by the sensor, to calculate the distance to its

surroundings. These types of sensors can achieve centimeter precision at distances over a hundred meters [9], enabling them to accurately sense the scale of 3D structures encountered outdoors. However, they have until recently often been limited in their ability to discriminate smaller features, highly attributed to the design of early LiDARs. Early LiDARs typically scan the surrounding environment by emitting multiple layers of laser beams that spin at a high rate, sharing the design principles of spinning radars. But just like the performance of radars has drastically improved by the advent of solid-state radars, the same is on the verge to happen with LiDARs. Solid-state LiDARs are promised to produce much denser 3D point clouds, at longer distances and at higher frame rates [10]. This increase in density means they can be coupled with cameras to produce RGB-D images that can capture outdoor geometries.

Expecting solid state LiDAR to be available in the near future, it is the intent of this thesis to explore how to improve upon the latest research in visual localization using synthetic RGB-D data.

1.1 Background and Related work

Many traditional localization techniques extract features from an image using feature detectors such as ORB [19], SIFT [11], SURF [20] and match these to features from previous frames to establish correspondences. From the matches, a 6-DoF pose is estimated by triangulation and later optimized using robust fitting algorithms such as RANSAC [13]. Feature-based methods such as SIFT [11] and ORB-SLAM2 [12] are able to localize in real-time with an error of a few centimeters in indoor environments but fail in low-texture environments. To solve this issue, direct approaches uses the whole input and minimizes a combination of the photometric and geometric error to estimate the camera pose between images [22][23]. Both methods tend to drift over time. To overcome the drift, these approaches check for loops by comparing the current image with a database of saved frames. If a loop is encountered and validated, the camera poses are globally optimized to minimize the drift.

Lately, probabilistic methods using random forest [24] and neural networks have emerged as potential techniques to solve the localization problem. While introducing a certain degree of problems on its own, the ability to learn from multiple, high dimensional input sequences and extract meaning from it is unmatched with these trainable systems. One other advantage of neural networks is that the time complexity of a model is constant compared to feature-based methods whose computational time increases as the number of saved frames grows.

The first neural network trained end-to-end for 6-DoF camera pose estimation, PoseNet (2015) [17], was built upon the GoogLeNet [18] architecture. The softmax classification layer from the original GoogLeNet model was replaced by two fully-connected regression layers which were modified to produce a vector with position and orientation estimates separately. The network used transfer learning technique from weights trained on object recognition tasks which drastically re-

duces the amount of training data needed. Several networks have since improved upon PoseNet. The authors of PoseNet proposed a trainable geometric loss function which removes the need for manual tuning of hyperparameters [43]. VLocNet [15] is a similar network that combines a network for global pose estimation together with a siamese network for visual odometry as a mean of improving the global pose estimation with multitask learning [44]. VLocNet++ [14] improves it additionally by adding a semantic network and fusing the extra information to the global pose network.

None of the above mentioned methods uses depth in the input data. Accurate RGB-D data is harder to acquire due to lack of cheap high resolution sensors. However, research has shown that depth data can improve the accuracy of object recognition networks [21]. Eitel et al. [26] argue that a siamese network with RGB and JET-encoded depth input data outperforms other combinations of depth encodings when used in combination with transfer learning.

1.2 Purpose

Encouraged by the success of fusing multi-modal information in VLocNet++, the purpose of the thesis project is to investigate how additional 3D information can improve the accuracy of outdoor localization in a predefined environment using neural networks. To quantify the system, the network will be compared to a baseline without depth information.

1.3 Scope

To prevent the scope of the thesis to grow to proportions unfeasible in the available time some limitations are set on the project.

- No limitation will be set on the computational resources needed to run the network.
- The network will be limited to localize in the area it is trained for. An adaptive algorithm which can learn to localize in new environments online would be preferable but it is outside the scope of the project to investigate such an algorithm.
- The project will primarily focus on how 3D information can improve the localization, not to optimize the whole localization system.

2

Theory

In this chapter, the theory of the relevant topics for the thesis project is briefly described. The first section describes the basics of machine learning and neural networks. The second section covers the theory of rigid motion, rotation transformations and representations. The third section presents the theory on point clouds used in the project. The fourth section covers the optical flow method, Lucas-Kanade, used in the thesis and in the final section, the theory on the point cloud registration network used in the system is presented.

2.1 Neural networks

Neural networks (NN) is a subfield in the area of artificial intelligence. The main idea of NNs is to mimic the neurons and synapses of the human brain. A NN is built up of layers of connected nodes, called neurons. Each neuron is essentially a weighting function which increases or decreases the strength of an input signal before propagating it forward to connected neurons. These weights are said to be trained through a process called backpropagation [37]. One strength with NNs is their ability to derive meaning from and generalize about complex data.

Mathematically the output of a neuron can be derived as

$$y = f(\sum_{i=1}^{m} w_i x_i + b)$$

where b is the bias in the neuron, w_i corresponds to the i:th weight and x_i corresponds to the i:th input. The input x might come from another neuron or it can be the input to the network.

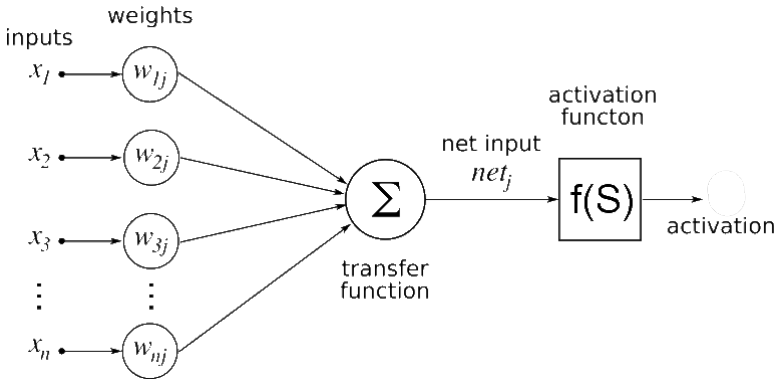


Figure 2.1: Diagram of an artificial neuron, by Chrislb, Wikibooks, Licensed under CC BY-SA 3.0, unaltered.

$f(.)$ is called an activation function. An activation function maps the weighted input to the output of a neuron. The important characteristic of the activation function is that of a normalizer, meaning that it should restrict the favored path in the network of growing without bound. It can also be shown that if the activation function is a non-linear function, a two or more layer NN can be proven to be a universal function approximator [38], which is not the case with linear activation functions. Two common activation functions are the rectified linear unit, ReLU and the sigmoid function. The ReLU function is defined as

$$f(x) = max(0, x)$$

The sigmoid activation function is defined as

$$f(x) = \frac{1}{1 + e^{-x}}$$

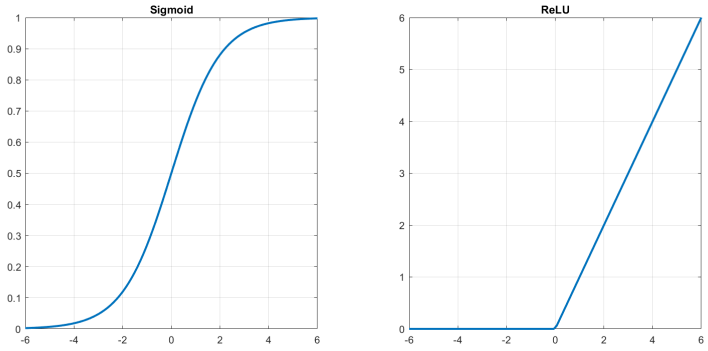


Figure 2.2: Sigmoid and ReLU activation functions.

2.1.1 Layers

Larger neural networks are built-up by multiple layers. In this subsection, commonly used layers and their properties are described.

Fully-connected layer The neurons in fully connected layers input every single output from the previous layer and propagate its activation to every single neuron in the next layer. A big advantage with fully connected layers is that they can learn new features from all combination of inputs, though they are very computationally expensive. Therefore fully connected layers are usually found in the end of convolutional neural networks dedicated to do higher level reasoning.

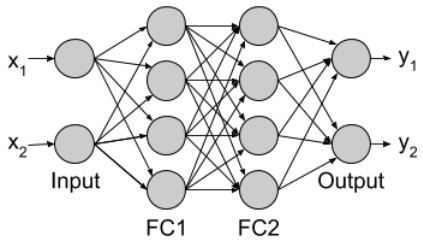


Figure 2.3: Illustration of two fully connected 4x1 layers with an input layer and output layer.

Convolution layers While a fully connected layer propagates the output from each activated neuron to every other neuron in the next layer, a convolutional layer resembles an ensemble of filters that calculates the cross-correlation over a small area of the input and convolving the filters over the entire input. This ensemble of filters produces a map of activations called a feature map, which is the output of the layer. The depth of the feature map is equal to the number of filters in the layer. The vertical and horizontal dimension of the feature map is determined by the kernel size, stride, padding and input size. The kernel size is the size of the filter sliding across the input, typically 1×1, 3×3 or 5×5. Stride determines the step size of the filter and helps in reducing the output dimension of the layer. If a consistent dimension is desirable, padding the input with zeros on the borders make the output dimensions the same after the convolution operation, this is called zero-padding. Weights for each feature map are then trained the same as the fully connected layers with backpropagation. A schematic view of the convolution operation can be seen in Figure 2.4

Pooling layer Pooling layers are also convolution layers, but unlike the convolutional layer explained above, pooling layers have no trainable weights but rather uses a heuristic to determine the action of the kernel. For a max pooling kernel of size 2×2, the heuristic is simply to select the largest value of the four and pass that to the next layer. The result is a sub-sampling of the input. An illustration of this operation can be seen in Figure 2.5.

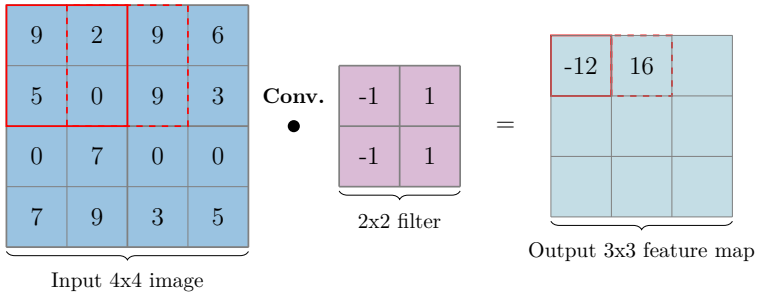


Figure 2.4: Figure illustrating the convolution operation of a 2×2 filter with stride 1. In this example, no zero-padding is used and the dimension of the output will be different from the input.

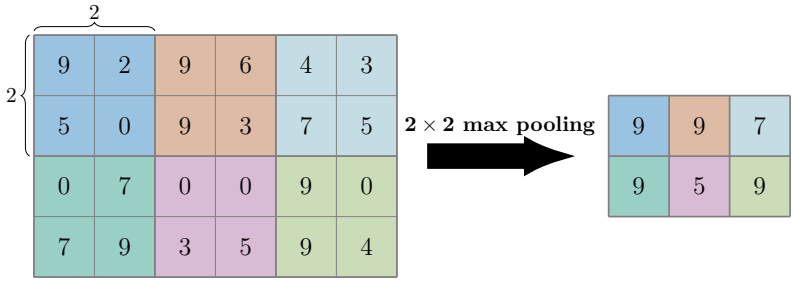


Figure 2.5: Illustration of a max-pooling operation.

2.1.2 Neural network topologies

How a network can be used and its properties are determined by its layers and how they are composed. When network designs share similar structure the topologies are said to be similar and they can usually be grouped under a label. Two common topologies and the ones used in this thesis are presented below.

Multi-Layer Perceptron A Multi-Layer Perceptron (MLP) is a neural network built with only fully connected layers. The entire design consists of an input layer, one or more fully connected layers and an output layer. It is a simple architecture were no connections between nodes form any loop. An example of a MLPs can be seen in Figure 2.3.

Convolutional neural networks A convolutional neural network (CNN) is, similar to the MLP, constructed by an input layer, several intermediate layers and an output layer. The intermediate layers are typically convolutional layers, pooling layers and ReLU activation layers. Before the prediction is done there is usually a few fully connected layers. CNNs are often used when the input has a spatial connection, like images. The filters in the convolution layers learn to detect features and objects in the image. The first layers typically learn filters to detect features like edges, blobs and lines while the middle filters detect combined features like wheels and noses. The last layers then detect more complex shapes like cars and faces. Compared to fully connected networks, CNNs can handle higher dimensional

inputs and build deeper models without exploding the parameter space. CNNs is commonly used in image analysis tasks such as object detection and classification.

2.1.3 Training neural networks

Machine learning can be divided into three main parts. Supervised learning, unsupervised learning and reinforcement learning. In supervised learning, each prediction is evaluated with respect to a known ground truth value and the loss is calculated from the difference of the prediction and the true value. In unsupervised learning, there is no ground truth data. Instead, the network finds patterns in the data and learn how to group the data together. In reinforcement learning an agent learns a task by trial and error where good actions are rewarded and the network learns to maximize the reward over time.

Loss functions A loss is often calculated as the difference between the ground truth and the network's prediction on a training example. To train the network, the weights are shifted by taking a step in the negative gradient direction of the computed loss by use of backpropagation. Commonly used loss functions are the mean squared error for regression tasks and categorical cross entropy for classification tasks.

Multi-loss functions To jointly learn several tasks in a network, a multi-loss function can be used. If the predicted values of these tasks are of different scales or even different domains, a tuning parameter β can be used to scale each part of the loss function such that all parts influence the loss. In the optimal case, the parameter is tuned such that each part contributes equally to the loss function.

$$\mathcal{L}_{tot} = \mathcal{L}_x + \beta\mathcal{L}_q$$

A big drawback with the above mentioned approach is that the tuning parameter has to be manually adjusted for each scene. It's a tedious process which takes a long time and must be repeated for different data sets. To overcome the problem, trainable tuning parameters can be introduced in the loss function. If one assumes there is an uncertainty in the output which can be represented as Gaussian noise, an output $f^W(x)$ where W is the weight on an input, the likelihood can be written as

$$p(y|f^W(x)) = \mathcal{N}(f^W(X), \sigma^2) \tag{2.1}$$

The log likelihood can be written as.

$$log(p(y|f^W(x))) \approx -\frac{1}{2\sigma^2}||y - f^W(x)||^2 - log(\sigma) \tag{2.2}$$

With two model output, the likelihood can be factorized as

$$\begin{aligned} p(y_1, y_2|f^W(x)) &= p(y_1|f^W(x)) \cdot p(y_2|f^W(x)) = \\ &= \mathcal{N}(y_1; f^W(X), \sigma_1^2) \cdot \mathcal{N}(y_2; f^W(X), \sigma_2^2) \end{aligned} \tag{2.3}$$

This leads to the minimization objective, $\mathcal{L}(W, \sigma_1, \sigma_2)$ for the multi-output model.

$$-log(p(y_1, y_2|f^W(x))) \approx \frac{1}{2\sigma_1^2}||y_1 - f^W(x)||^2 \frac{1}{2\sigma_2^2}||y_2 - f^W(x)||^2 + log(\sigma_1\sigma_2) \quad (2.4)$$

With $L_1(\mathcal{W}) = ||y_1 - f^W(x)||^2$ and $L_2(\mathcal{W}) = ||y_2 - f^W(x)||^2$ the final loss function can be written as

$$\mathcal{L}_{tot} = \frac{1}{2\sigma_1^2}\mathcal{L}_1 + \frac{1}{2\sigma_2^2}\mathcal{L}_2 + log(\sigma_1\sigma_2) \quad (2.5)$$

This can be interpret as minimizing the loss with respect to σ_1 and σ_2 as learning the relative weight of the losses $\mathcal{L}_1$ and $\mathcal{L}_2$ adaptively based on the data. A large uncertainty leads to a lower weight on the corresponding loss. The last term acts as a regularization term, avoiding a large increase in the uncertainty which could grow independant of the data to minimize the loss otherwise.

Transfer Learning Transfer learning is a method in machine learning where a model trained on a task is reused as the base for another task. For example, the weights from a network used to classify objects in an image can be reused in a network designed to output a bounding box for the object in the image. The benefits of transfer learning are primarily that it reduces the amount of training data needed. Secondly, it also speeds up the learning process since the weights are in a better range to start with. The similarity of the tasks and the domains between two networks determine how effective transfer learning can be and if it can be used at all.

When adding new layers on top of pre-trained models the new weights will be initialized randomly according to some distribution. The random initialization might lead to exploding gradients during the initial training which may propagate down into the pre-trained layers and severely alter their weights. To prevent this, layers can be frozen, meaning that their weights do not change during training. When the new weights have been more or less set, the earlier layers can be unfrozen and the whole model can be fine-tuned together.

2.2 Rigid Motion and twist

3D rigid body motion can be represented as $m(x) := Rx + t$, where $R \in SO(3)$ is a rotation vector with $SO(3) := \{R \in \mathbb{R}^{3\times3}|det(r) = 1\}$ and $t \in \mathbb{R}^3$ is a translation vector. If homogeneuos coordinates are used, m can be written like

The set of all matrices of this form is called the Lie group $SE(3)$. For every Lie group, there is a corresponding Lie algebra. The Lie algebra for $SO(3)$ is $so(3) := \{A \in \mathbb{R}^{3\times3}|A^T = -A\}$, and the Lie algebra for $SE(3)$ is $se(3) := \{(v, \omega)|v \in \mathbb{R}^3, \omega \in so(3)\}$. The elements in $se(3)$ can be mapped to elements in $SE(3)$, and vice versa, which means that rigid rotations can be represented with the elements in $se(3)$. These elements are usually called twist. The advantage being that we can

represent a six degree of freedom rotation with six parameters instead of twelve. The elements of $so(3)$ and $se(3)$ can be written both as vectors or matrices with the following expressions

$$\hat{\omega} = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix} \in so(3), \quad \xi = \begin{bmatrix} \hat{\omega} & v \\ 0_{3\times 1} & 0 \end{bmatrix} \in se(3). \tag{2.6}$$

A twist se(3) can be converted to an transformation in SE(3) by the exponential function $M = exp(\xi) = \sum_{k=0}^{\infty} \frac{\xi^k}{k!}$. Full derivation can be read in [40].

2.2.1 Rotation formalism

There exist many different ways of representing rotations in 3D. Common representations are Euler angles, quaternions and rotation matrices. Inherent to all these representations is that they can describe rotations with three degrees of freedom, even though they use different numbers of parameters.

Rotation matrix All square matrices are rotation matrices if and only if $R^T = R^{-1}$ and $det(R) = 1$. In the 3D case, they can be represented by three elemental rotation matrices.

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) \\ 0 & sin(\theta) & cos(\theta) \end{bmatrix} \tag{2.7}$$

$$R_y = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix} \tag{2.8}$$

$$R_z = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.9}$$

These three matrices each describe a counterclockwise rotation of θ degrees around the x-, y-, or z-axis in a right-hand rule coordinate system pointing towards the observer. A general rotation can be described by matrix multiplying the three elemental matrices with each other as

$$R = R_x(\theta)R_y(\phi)R_z(\psi). \tag{2.10}$$

Describing rotations as rotation matrices make it easy to apply to vectors, but in turn, makes it a less concise representation than other alternatives.

Euler Angles Using Euler angles is one way of describing a rigid body rotation with respect to an extrinsic (fixed) or intrinsic (moving) frame with three angles, Roll(θ), Pitch(ϕ) and Yaw(ψ). Each angle corresponds to a rotation around each

axis in a 3-dimensional coordinate system, making it a compact representation which is easy to comprehend. Depending on if an intrinsic or extrinsic frame is used, an orientation can be calculated as the rotation over the axes in multiple orders. In the extrinsic case, this results in $3! = 6$ combinations and in the intrinsic case this yields $3 \cdot 2 \cdot 2 = 12$ combinations.

Euler Angels are restricted by the Gimbal lock phenomenon. Gimbal lock is the name for the situation when one of the rotation axes align itself with another and now any rotation around these two axes represent the same rotation, effectively removing one degree of freedom from the system.

Another problem with the Euler angle representation for regression learning of poses is the discontinuous nature of the defined set of allowed rotations, i.e. if the allowed rotations are in the set $[0, 360°]$, and the object points in a direction close to 360° in one axis, adding any more rotation on that axis makes the orientation angle jump down close to 0.

Quaternions A Quaternion can be described with the equation

$$q = q_w + bq_i + cq_j + dq_k \tag{2.11}$$

where q_w is described as the scalar part of q and $[q_i, q_j, q_k]$ is described as the vector part of q. The representation of a rotation as a quaternion (4 numbers) is more compact than the representation as an orthogonal matrix (9 numbers) but one larger than the Euler representation. However, this extra number brings with it several key-properties that makes the quaternion representation common in engineering.

For a given axis and angle, one can easily construct the corresponding quaternion, and conversely, for a given quaternion one can get the axis and the angle. Both of these are much harder to easily extract from rotation matrices or Euler angles.

The expression qpq^{-1} rotates any vector quaternion p around an axis given by the vector a and the angle θ where a and θ depends on the quaternion $q = q_r + q_i i + q_j j + q_k k$. a and θ can be found from the following equations:

$$(a_x, a_y, a_z) = \frac{(q_i, q_j, q_k)}{\sqrt{q_i^2 + q_j^2 + q_k^2}} \tag{2.12}$$

$$\theta = 2\,\mathrm{atan2}\left(\sqrt{q_i^2 + q_j^2 + q_k^2}, q_r\right) \tag{2.13}$$

Euler angle to quaternion conversion One problem when converting from an Euler angle representation to a quaternion q, is that both q and $-q$ represent the same rotation. Given a vector represented by the quaternion p, and a rotation represented by the quaternion q, the resulting quaternion rotated by q is $p' = qpq^{-1}$. On the other hand, rotating p by $-q$ yields

$$-qp(-q)^{-1} = (-1)qpq^{-1}(-1) = (-1)^2qpq^{-1} = qpq^{-1}, \tag{2.14}$$

which is an identical rotation.

One way of restricting this duality is to only allow the scalar part of the quaternion to be positive and if a situation arises when the resulting quaternion gets a negative scalar value, setting the quaternion to its inverse instead as $q = -q$.

2.3 Point clouds

Point clouds are sets of 3D data points usually sampled from a continuous geometric shape, either from the real world with sensors or from mathematical models in a computer.

2.3.1 Iterative Closest Point

One of the most common methods for point cloud registration is the iterative closest point (ICP) algorithm, introduced by Besl and McKay in 1992 [30]. The algorithm is used for aligning two similar point clouds

$$\mathcal{S} = s_{ii=1\ldots N} \tag{2.15}$$
$$\mathcal{T} = t_{ii=1\ldots N} \tag{2.16}$$

where the source cloud is denoted as $\mathcal{S}$ and the target point cloud is denoted as the $\mathcal{T}$. The algorithm computes the transformation matrix $\mathbf{W}$ which transform the source cloud to the target cloud. The main part of the algorithm is the iterative search for closest match point pairs and finding the transformation that best aligns these by least squares. Point pairs are found by comparing the distance from one point in $\mathcal{S}$ to every other point in $\mathcal{T}$ and choosing the one with the smallest distance, as long as it is below some bound $d > 0$. The transformation is then calculated as

$$\mathbf{W} = \underset{\mathbf{W}}{\operatorname{argmin}} \frac{1}{N_p} \sum_{i=1}^{N_p} ||t_i - \mathbf{W}s_i||^2 \tag{2.17}$$

The algorithm terminates when the resulting mean square error of the registration falls below some bound $\tau > 0$.

There are however two main downsides of the ICP algorithm. The explicit estimation of closest point correspondences results in the complexity scaling quadratically and it easily gets stuck in bad local minimum if the initial alignment is not very accurate. The algorithm is also nontrivial to integrate which leads to issues in using it with deep learning frameworks.

2.3.2 Surface Normals

In 3D, the surface normal of a point $p = (x,\ y,\ z)$ is a vector perpendicular to the surface spanned by the neighboring points to p. The problem of determining the normal to a point on the surface can be approximated by the problem of estimating the normal of a plane tangent to the surface, which in turn can be solved by a least-square plane fitting algorithm.

One way of finding the tangent normal for a point is by Principal Component Analysis (PCA). PCA analyzes the eigenvector and eigenvalue of the covariance matrices created from k number of points in the neighborhood of the point of interest. For each point p_i the covariance matrix C_i is created as

$$C = \frac{1}{k}\sum_{i=1}^{k}(p_i - \bar{p}) \cdot (p_i - \bar{p})^T, \quad C \cdot \overrightarrow{v_j} = \lambda_j \cdot \overrightarrow{v_j},\ \ j \in \{1, 2, 3\} \tag{2.18}$$

where $\overline{p}$ represents the 3D centroid of the nearest neighbors, λ_j is the j-th eigenvalue of the covariance matrix, and $\vec{\mathbf{v}_j}$ the j-th eigenvector.

As C is a symmetric and positive semi-definite matrix and its eigenvalues are real numbers $\lambda_j \in \mathcal{R}$, the eigenvectors $\vec{v}_j$ form an orthogonal frame, corresponding to the principal components of P_k. If $0 \leq \lambda_0 \leq \lambda_1 \leq \lambda_2$, the eigenvector $\vec{v}_0$ corresponding to the smallest eigenvalue λ_0 is the approximation of $\vec{n} = (n_x, n_y, n_z)$ or $-\vec{n}$.

2.3.3 PointNet

Until recently, most work on 3D data analysis with neural networks has been done with CNNs. There have been many proposed ways of representing 3D data in a way that makes it compatible with the architecture of CNNs. Some methods that have been used to do this is to represent the point set in a 3D occupancy grid or as 3D voxels. However, this data transformation often render the resulting data unnecessarily voluminous. The primary concern with this approach is that the computations needed to process high resolution volumes grows unfeasibly fast. Other concerns are that 3D point clouds typically are very sparse, which yields unnecessarily many computations on empty data cells.

PointNet [28] is a method of processing raw point cloud data that has greatly improved performance in classification and segmentation tasks. PointNet has also recently been shown to be suitable for point cloud registration, which will be presented in Section 3.4.1. The structure of the PointNet architecture originates from insights that one key property of point clouds is that they are, unlike image arrays or voxel grids, inherently unordered. This insight the design of the network to one that is invariant to input permutations. PointNet does this by independently and identically process each point with a shared Multi-Layer Perceptron network. After the MLP network, the output propagates through a symmetric operation which in most circumstances is a max-pooling layer. These symmetric operations encode sets

of optimization functions/criteria that select interesting or informative points of the point cloud. A general function of this can be expressed as

$$f(x_1, ..., x_n) \approx g(h(x_1), ..., h(x_n)), \tag{2.19}$$

where $f : 2^{\mathbb{R}^N} \longrightarrow \mathbb{R}$, $h : \mathbb{R}^N \longrightarrow \mathbb{R}^K$ and $g : \underbrace{\mathbb{R}^K \times ... \times \mathbb{R}^K}_{n} \longrightarrow \mathbb{R}$.

In PointNet, h is approximated by the MLP network and g by a set of single-parameter functions and a max-pooling function. From this operation, a global feature vector $[f_1, ..., f_n]$ is acquired from which additional processing can be made for various tasks. In the original paper, methods for object classification and pointwise segmentation is presented. A simplified visualization of the network architecture as it is described in the original PointNet paper [28] is shown in Figure 2.6

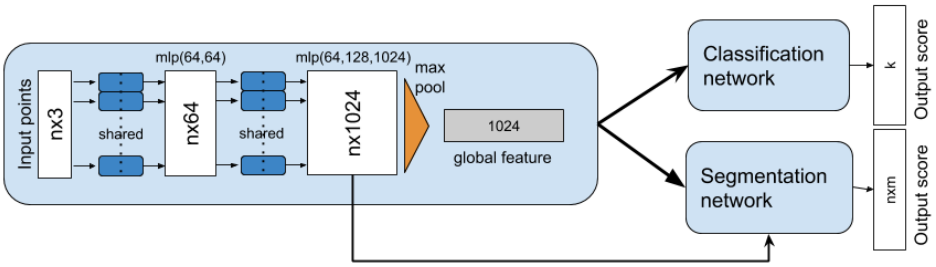


Figure 2.6: A simplified schematic view of the PointNet architecture as it is presented in the original PointNet paper [28].

2.4 The Lucas–Kanade algorithm

In 1981, Lucas and Kanade introduced an algorithm to estimate the optical flow in an image [31]. By assuming that corresponding pixels in two pictures taken close together in time shifts only a small amount, one can assume that the flow in the local region around a pixel is constant. The goal of the algorithm is to minimize the sum of squared error between the template image T and the source image S by finding the transformation that best aligns the two images.

$$\sum_x [S(W(x;p)) - T(x)]^2 \tag{2.20}$$

where $W(x;p)$ is the set of allowed warps of the image and p a set of parameters $p = [p_1...p_n]^T$ for the warp function. The sum is minimized with respect to p and the sum is over all pixels in the image T.

The algorithm assumes there exists some initial guess of the parameters p and then iteratively update the guess p by minimizing the sum

$$\sum_x [S(W(x;p+\Delta p)) - T(x)]^2 \tag{2.21}$$

with respect to Δp and then update p as $p \leftarrow \Delta p + p$.

This is then repeated until a threshold ϵ is reached on the norm of the updated p; $\epsilon \leq ||\Delta p||$.

The derivation of this algorithm starts by linearizing 2.20

$$\sum_x [S(W(x;p)) + \nabla S \frac{\partial W}{\partial p} \Delta p - T(x)]^2 \tag{2.22}$$

where $\nabla S = (\frac{\partial S}{\partial x}, \frac{\partial S}{\partial y})$ is the gradient of S evaluated at $W(x;p)$ and $\frac{\partial W}{\partial p}$ is the Jacobian of the warp with the form

$$\frac{\partial W}{\partial p} = \begin{bmatrix} \frac{\partial W_x}{\partial p_1} \frac{\partial W_x}{\partial p_2} \cdots \frac{\partial W_x}{\partial p_n} \\ \frac{\partial W_y}{\partial p_1} \frac{\partial W_y}{\partial p_2} \cdots \frac{\partial W_y}{\partial p_n} \end{bmatrix} \tag{2.23}$$

minimizing 2.21 is a min-mean square error problem with a closed form solution that is derived as

$$2 \sum_x [\nabla S \frac{\partial W}{\partial p}]^T [S(W(x;p)) + \nabla S \frac{\partial W}{\partial p} \Delta p - T(x)] \tag{2.24}$$

which is the partial derivative of 2.21 with respect to Δp. Setting this expression to zero and solving for Δp yields

$$\Delta p = H^{-1} \sum_x [\nabla S \frac{\partial W}{\partial p}]^T [T(x) - S(W(x;p))] \tag{2.25}$$

where H^{-1} is the Hessian matrix

$$H = \sum_x [\nabla S \frac{\partial W}{\partial p}]^T [\nabla S \frac{\partial W}{\partial p}]. \tag{2.26}$$

In the end, the LK algorithm iteratively applies 2.25 and updates p as $p \leftarrow \Delta p + p$. When referring to different versions of the Lucas-Kanade (LK) algorithm, this is often called the **additive** approach.

Another approach that produces the same result, solves for an update of the inverse warp W^{-1} instead of an update of the warp parameters p. In this approach, the roles

of the source and template images are inverted which will be shown can decrease the cost of the computations tremendously. This version of the LK algorithm is often called the **inverse compositional** (IC) approach.

The goal objective in the IC case is to minimize

$$\sum_x [T(W(x;\Delta p)) - S(W(x;p))]^2 \tag{2.27}$$

and update the warp function W by

$$W(x;p) \leftarrow W(x;p) \cdot W(x;\Delta p)^{-1}. \tag{2.28}$$

The derivation of the IC approach starts with

$$\sum_x [T(W(x;0)) + \nabla T \frac{\partial W}{\partial p} \Delta p - S(W(x;p))]^2. \tag{2.29}$$

By assuming that the identity warp ($W(x;0) = x$) is in the set of warps being considered the expression can be simplified to

$$\Delta p = H^{-1} \sum_x [\nabla T \frac{\partial W}{\partial p}]^T [S(W(x;p)) - T(x)] \tag{2.30}$$

where H^{-1} is the Hessian matrix

$$H = \sum_x [\nabla T \frac{\partial W}{\partial p}]^T [\nabla T \frac{\partial W}{\partial p}] \tag{2.31}$$

where the Jacobian $\frac{\partial W}{\partial p}$ is evaluated at $W(x;0)$. In 2.31, there is nothing that depends on p across iterations, therefore this Hessian can be kept constant and pre-computed. Since this is by far the costliest computation in the algorithm, this version is most often found in systems that need to run in real-time.
proof that the two presented versions of the Lucas-Kanade algorithm are equal can be found in the paper[32].

2.5 PointNetLK

In 2019, Y. Aoki et al. presented a neural network for point cloud registration called PointNetLK [29]. This network was based on PointNet and a modified version of the IC Lucas-Kanade algorithm. They show that this network outperforms the current best ICP based methods both in its ability to match severely miss-aligned point clouds while at the same time converge to a solution several orders of magnitude faster.

The PointNetLK architecture uses a PointNet MLP network as a function $\phi : R^{Nx3} \rightarrow R^K$. With this function any N-dimensional point cloud P will produce a K-dimensional global feature vector $\phi(P)$. The optimization is set up as finding the rigid transformation $G \in SE(3)$ that best aligns the source and template point clouds $P_{\mathcal{S}}$ and $P_{\mathcal{T}}$. The objective is first formulated as finding the inverse transformation G^{-1} with the Inverse Compositional LK method as

$$\phi(P) = \phi(G^{-1} \cdot P_{\mathcal{T}}) \tag{2.32}$$

this is then linearized as

$$\phi(P) = \phi(P_{\mathcal{T}}) + \frac{\partial}{\partial \xi}[\phi(G^{-1} \cdot P_{\mathcal{T}})]\xi \tag{2.33}$$

where G^{-1} is defined as the inverse exponential map from $se(3)$ to $SE(3)$ $G^{-1} = exp(-\sum_i \xi_i \mathrm{T}_i)$ where ξ is the twist parameters and T is the generators for the exponential map. Just as in the original IC approach this prevents us from doing costly Jacobian calculations every iteration.

In the classical LK algorithm, the Jacobian $J = \phi(P_{\mathcal{T}}) + \frac{\partial}{\partial \xi}[\phi(G^{-1} \cdot P_{\mathcal{T}})]$ is calculated by splitting the expression in two and using the chain rule to calculate an image gradient in the ND image directions and an analytical warp Jacobian. But, as mentioned before, this does not work in this case since it does not exists any structure which allows taking gradients in the x, y or z directions. The solution presented to this issue is to calculate J in a stochastic gradient approach. This is done by calculating each column i in J as a finite difference gradient as

$$J_i = \frac{\phi(exp(-\xi_i T_i) \cdot P_{\mathcal{T}}) - \phi(P_{\mathcal{T}})}{t_i} \tag{2.34}$$

where t_i are infinitesimal small perturbations of the twist parameters ξ. It is now possible to solve for ξ as

$$\xi = J^{+}[\phi(P_{\mathcal{T}}) - \phi(P_{\mathcal{S}})] \tag{2.35}$$

where J^{+} is the pseudoinverse of J. The iterative algorithm consists of calculating the twist parameter with 2.35 and updating the source point cloud as

$$P_{\mathcal{S}} \leftarrow \Delta G \cdot P_{\mathcal{S}} \quad \Delta G = exp \sum_i \xi_i T_i. \tag{2.36}$$

This loop is repeated until a stopping criterion is reached which is either reaching a max number of iterations or updating less than a minimum threshold for ΔG. The final estimate is then calculated as the composition of the iterative updates ΔG as

$$G_{est} = \Delta G_n \cdot ... \cdot \Delta G_1 \cdot \Delta G_0. \tag{2.37}$$

A visual representation of the algorithm is shown in Figure 2.7.

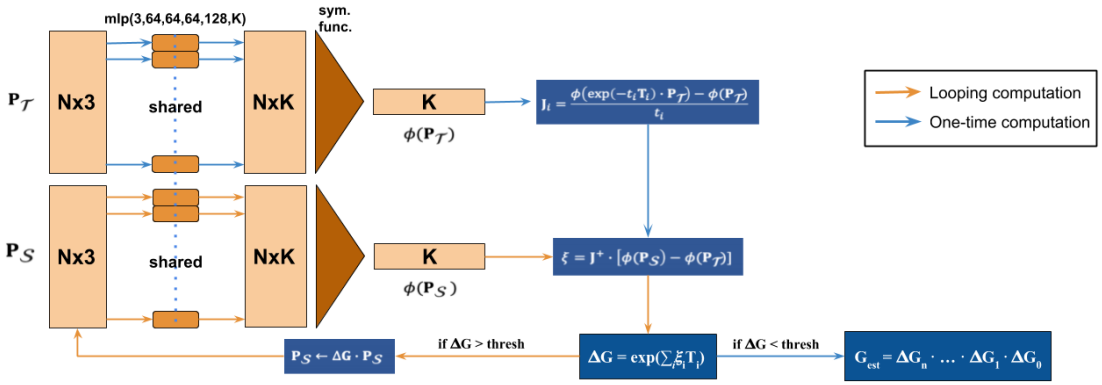


Figure 2.7: The PointNetLK network used for registration.

3
Method

In this chapter, the workflow and methods used to construct the various sub-components of the localization system are presented. The main focus of this chapter is the presentation of the two neural networks used and how the data sets were acquired in the project.

Most of the developed code was written in Python, using Keras [33] with TensorFlow [34] as backend for the implementation of the convolutional RGB-D network. For the PointNetLK code, Pytorch was used to implement the network. All training was done on a GeForce RTX 2080 Ti, with 11GB of GDDR6 memory and 4352 CUDA cores.

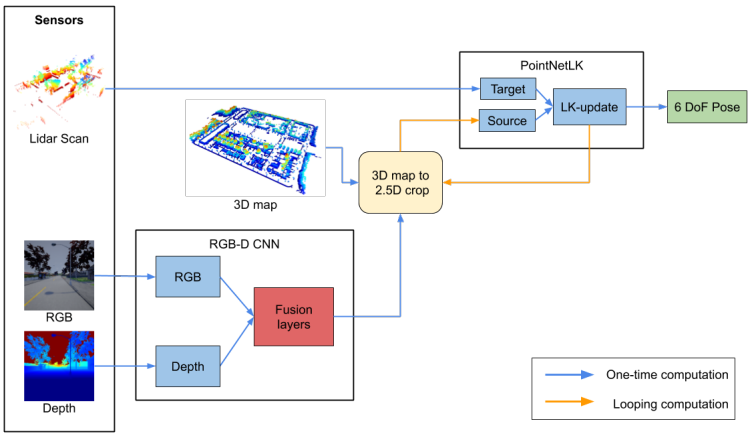


Figure 3.1: Flowchart of the fully combined system.

3.1 Full system design

The purpose of the thesis was to test if camera and depth sensor data can be combined and fused in a way that increased the performance and robustness of a system for localization. The final system estimates the 6 DoF pose in a known environment with a RGB image, a depth image and a LiDAR scan taken at the same position as input. Figure 3.1 shows the dataflow of the entire system which was constructed in two parts.

Image based localization prediction network The first part was an image based prediction network used to produce an initial guess of the pose where a RGB and a depth image was taken. The image pair was fed through a convolutional neural network which estimates a 3D position and a quaternion, together representing the pose of the camera. The idea was to utilize the ability of neural networks to generalize about complex input data to learn a suitable way of fusing the information from the color and depth image, making the predictions more robust to changes in the environment.

Point cloud position refinement network The position refinement network was used to correct the guess from the prediction network by matching the current LiDAR scan to a previously created map of the surrounding area. The refinement network that was used is a state-of-the-art 3D point cloud registration network called PointNetLK. However, one limitation of this network is its inability to match partial segments of a 3D model to the full model. This shortcoming guided our implementation to one that uses the predicted position from the image network to extract areas around the predicted position in the full 3D map. This set of candidate areas was then matched to the LiDAR scan. The goal being that these cropped areas contain the right amount of similar 3D data for PointNetLK to estimate a correct transformation. The output transformation G was then used to refine the initial position estimate.

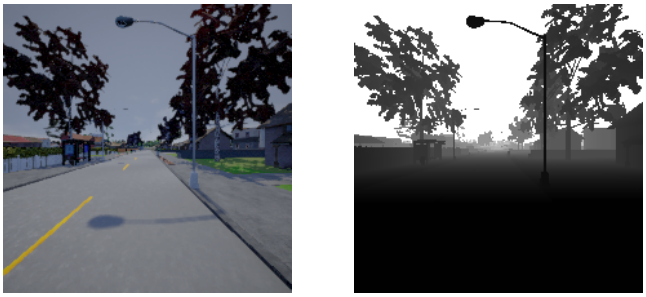
3.2 Data sets

To train a neural network with supervised learning, a large amount of data is typically needed. To train and evaluate the system design, a data set that contains RGB images, depth images and high resolution LiDAR scans with corresponding pose labels would be needed. This was however not found in the public domain and it was decided that a simulator should be used to create the required data.

3.2.1 CARLA Simulator

To generate simulated data, the CARLA driving simulator [27] was used. CARLA is an open source simulator developed for autonomous car research. With CARLA, any amount of high quality data with matching pose labels can easily be created.

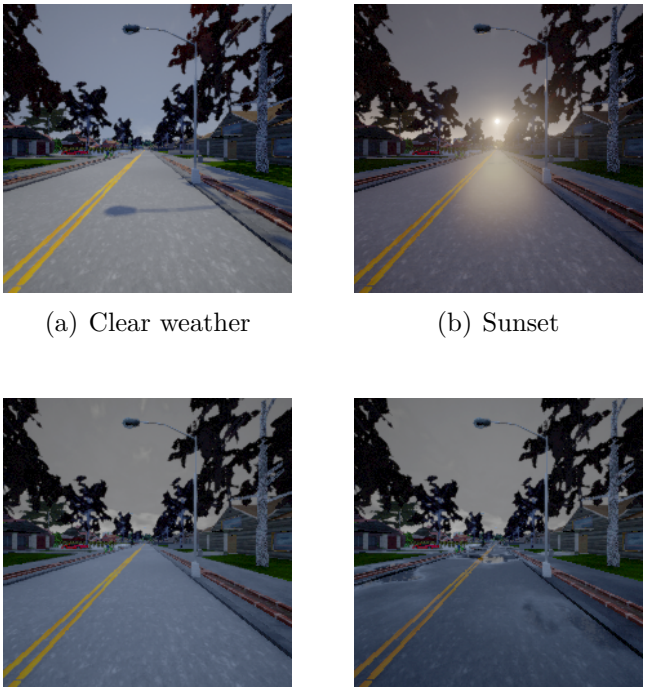
A camera sensor can be attached to a vehicle in CARLA, recording images with a preset frame rate. The image size and field-of-view can be set to create images with the correct input size for a certain network architecture. The camera sensor can produce both RGB and depth images, as shown in 3.2



(a) RGB Image (b) Depth Image

Figure 3.2: Example RGB image and corresponding depth map.

The simulator has the capability to simulate several weather conditions, as depicted in 3.3 For example, rainy weather with puddles reflecting light, cloudy weather which darkens the environment and a simulated sunset which create artificial lens flares when the sun is in frame. With a data set consisting of varying weather, the network can hopefully learn more general information in the environment and become more robust.



(a) Clear weather (b) Sunset

(c) Cloudy weather (d) Rainy weather

Figure 3.3: Images visualizing images in different weather conditions.

Emulated LiDAR sensors are available in CARLA as well. All relevant parameters such as upper and lower field of view, number of channels, max range and number of points per channel can be configured in CARLA. During a scan capture, the simulation environment can be frozen resulting in a 360° scan without any need for velocity adjustments. A scan is shown in Figure 3.4.

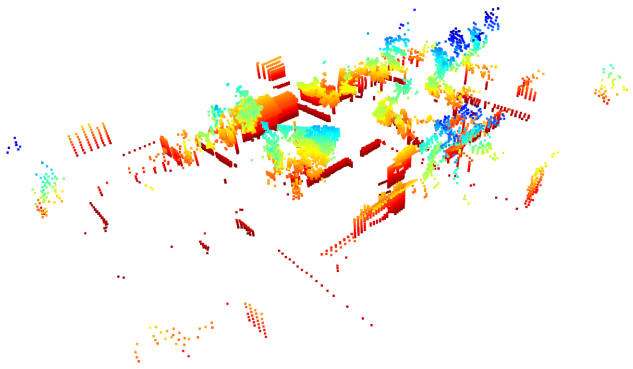


Figure 3.4: LiDAR scan from CARLA.

3.2.2 Data collection

To generate the data sets used in the project, a vehicle was set to drive around a map on autopilot with a RGB and a depth camera sensor attached to it. The map used to generate the data set was approximately 120 000m^2 containing a downtown area, suburban areas and wooded areas. Figure 3.5 shows a top view of the environment. Two training data sets were acquired, one with 7600 image pairs in clear weather and one with 10400 image pair with varying weather conditions. The varying weather data set had an equal distribution of images taken in clear, cloudy, rainy and sunset conditions. 10% of the training images were used for validation during training.
Two smaller corresponding test data sets were recorded to be used to evaluate the performance of the networks. The test data sets also contained LiDAR scans that are used in the refinement process.
To generate data equally distributed over the entire city, a delay of approximately 5 seconds were introduced between each captured image. The vehicle was also randomly re-spawned at regular intervals into different parts of the city during the data acquisition.

3D map creation To create a point cloud map, the same autopilot mode was used as with the image data collection. All individual scans were then added together to build a map. Every scan was captured in a local coordinate system with the sensor position as origin and thus each local scan had to be converted into a global coordinate system to create the full map. Given the vehicles global position and heading, a scan can be transformed from a local to a global coordinate system.
The whole scan was rotated with a yaw rotation matrix 3.1. A yaw rotation, expressed in degrees, was used instead of a full rotation matrix since the map is flat and the roll and pitch angle are almost constant in the data set.

$$\begin{bmatrix} cos(\phi) & -sin(\phi) & 0 \\ sin(\phi) & cos(\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.1}$$

Figure 3.5: Top view of the used environment in CARLA simulator.

After the rotation, the ground truth position of the vehicle was added to all points. During this process the road geometry is cropped by removing all points with a height lower than a certain threshold. Since the road height was identical around the map no extra global position information was gained from it. The road surface also created problems in the point cloud registration system, both by decreasing pose accuracy but also by increasing the computational speed as the number of points in the map and the scan increased. Figure 3.6 shows the final 3D map.

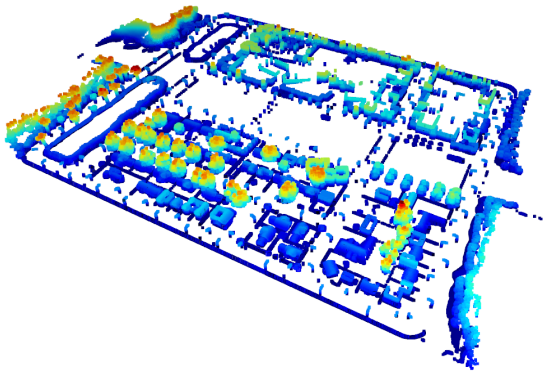


Figure 3.6: Generated 3D map from scans.

3.2.3 Angle representation

The ground truth rotation output from CARLA was in Euler angles, where Roll(θ), Pitch(ϕ) and Yaw(ψ) was in the range -180° to 180°. With this representation, there is a non-continuous transition between -180° and 180°. This is an issue when

training neural networks on a regression task since a -180° prediction where the ground truth is 179.9° is accurate but in terms of non-cyclic loss functions it would produce close to a maximal loss. To overcome this problem the angels were converted to a constrained quaternion using the following equations:

$$q_w = cos(\frac{\psi}{2})cos(\frac{\phi}{2})cos(\frac{\theta}{2}) + sin(\frac{\psi}{2})sin(\frac{\phi}{2})sin(\frac{\theta}{2}) \tag{3.2}$$

$$q_x = cos(\frac{\psi}{2})cos(\frac{\phi}{2})sin(\frac{\theta}{2}) - sin(\frac{\psi}{2})sin(\frac{\phi}{2})cos(\frac{\theta}{2}) \tag{3.3}$$

$$q_y = sin(\frac{\psi}{2})cos(\frac{\phi}{2})sin(\frac{\theta}{2}) + cos(\frac{\psi}{2})sin(\frac{\phi}{2})cos(\frac{\theta}{2}) \tag{3.4}$$

$$q_z = sin(\frac{\psi}{2})cos(\frac{\phi}{2})cos(\frac{\theta}{2}) - cos(\frac{\psi}{2})sin(\frac{\phi}{2})sin(\frac{\theta}{2}), \tag{3.5}$$

$$q = \begin{bmatrix} q_w \\ q_x \\ q_y \\ q_z \end{bmatrix}. \tag{3.6}$$

To restrict the quaternion to one hemisphere each element was inverted if q_w was negative. Constraining a quaternion to one hemisphere results in a smooth rotation since the duality property, described in Section 2.2.1, is removed.

3.3 RGB-D Neural network

The aim was to design a network that takes two images, one RGB and one depth image, as input and produce a 3D translation vector and a quaternion representing the orientation. The idea was that the network learns to connect objects in the scene to an orientation, effectively building a virtual map encoded in the network's weights.

3.3.1 Network design

The network design was inspired by the work of Kendall et al. [17]. It uses the GoogLeNet [18] architecture with inception modules. The inception module, pictured in Figure 3.7, consists of several convolutions filters with different kernel size. The varying filter size increases the probability that a network finds similar features, even if the size of them differs between images. To reduce the computational cost of the bigger convolutions, 1×1 convolutional filters are added prior to reduce the dimensionality. The module allows a network to learn similar features with fewer layers and weights compared to a network were layers are just stacked on top of each other.

Using this existing network architecture enables the use of pretrained weights to speed up the learning process and reduce the need of a large data set. The chosen pre-trained weights are initially trained on a scene recognition task on a data set called Places[41]. Intuitively, this task should be similar to that of scene localization as both depend on scene understanding more than the typical object recognition task.

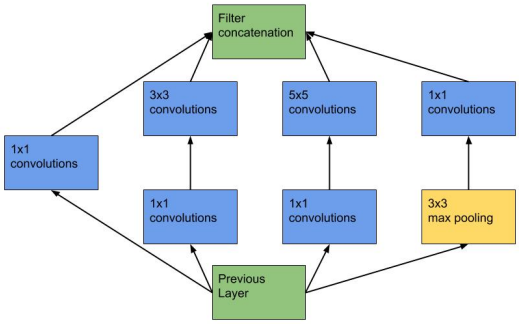


Figure 3.7: Illustration of an inception module.

The whole GoogLeNet architecture is shown in Figure 3.14. The network begins with a few max pooling, batch normalization and convolution layers before 9 inception modules are stacked on top of each other. After the final inception modules a large 7×7 max pooling layer decreases the input size to the final fully connected layer. Two small auxiliary networks are added after the third and sixth inception module. The idea of them is to avoid vanishing gradients while backpropagating through many layers.

Complete network architecture The complete RGB-D network, depicted in Figure 3.8, uses two identical GoogLeNet models. One that is trained on RGB images as input and one with JET encoded depth images as input. The final three layers fuse the information from both networks before the rotation and translation predictions are made. The fusing was done by stacking the output from the last inception module of both networks depth wise and inputting this to a 7×7 average pooling layer. A fully connected layer with 1024 neurons then encode the information and learns how to extract the pose from the combined feature map. In total there were 14 million trainable weights in the network.

3.3.2 Network training & preprocessing

The network was trained in several stages. First the identical RGB and depth models were trained individually for 800 epochs each, saving the weights from the best performing epoch. Afterwards, the models were fused into one network as described in Section 3.3.1 and finetuned while the previously trained layers were frozen. Training the networks separately took around 5 hours each and finetuning took approximately 60 hours, leading to a total training time of 70 hours.

For training, all images were center-cropped to the size of 224x224 pixels. The Adam [35] optimizer was used with a clip value on the gradients of 1.5 and a learning

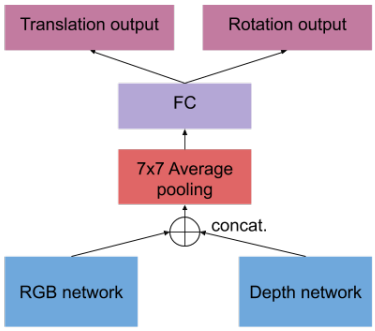


Figure 3.8: Overview of the top layers of the full network architecture. The output size of the concatenated RGB and depth network is 7x7x2048 and the fully connected layer has 1024 neurons.

rate of 0.001. To be able to effectively use pretrained weights, the inputs on the new task needs to be in the same range as they were in the original training, in other words the same preprocessing techniques need to be applied. For this task, the mean of the training data set was subtracted from all images. The same mean was used for the training, validation and test data set. The mean was calculated as

$$\mu_c = \sum_i X_{i,c} \tag{3.7}$$

where the sum was taken over all pixel in the three available channels c_i. The idea behind the normalization technique is to center the data into a similar range. This will prohibit the gradients to explode and a common learning rate can be applied to the whole training process.

Colorized depth encoding The depth sensor produces a one channel depth map where each pixel value corresponds to the distance from the sensor to the pixel. Encoding the one channel depth value to a three channel color encoding is useful since it enables the use of pre-trained RGB network architectures for depth images as well.

To calculate the colorized image, the depth map was first normalized to values between 0 and 255 and then passed through a function which transforms the map to a 3 channel encoding. In the project the JET colormap function was used.

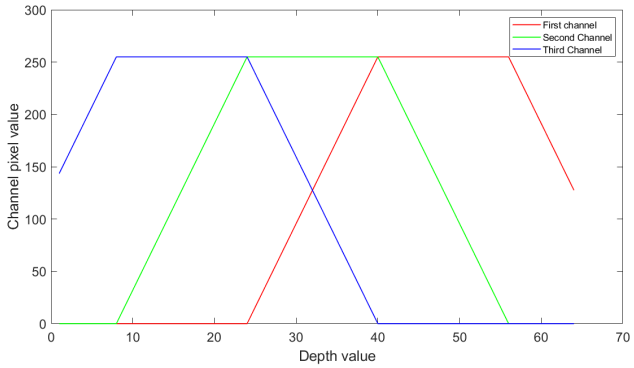
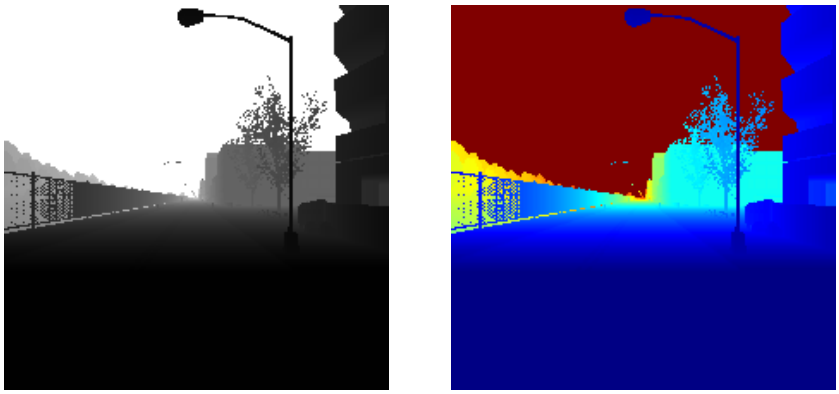


Figure 3.9: JET colormap encoding for a 5 bit depthmap.

Using the JET colormap, pixels near the sensor were colored blue and pixels far away were colored red. A depth image encoded this way is shown in Figure 3.10



(a) Depth Image (b) JET colorized Image

Figure 3.10: Example depth image and its corresponding JET encoded representation.

Trainable loss function The network has two tasks, to predict a translation and a rotation. During training the angle error and the translation error will have different magnitudes, meaning they will impact the magnitude of the gradients during backpropagation differently. To balance the importance of both tasks, a trainable loss function described in Section 2.1.3 was implemented with a minor modification. To create a more numerically stable loss function and avoid division with zero, the loss was rewritten as

$$\mathcal{L}_{tot} = \mathrm{e}^{-\hat{s}_x}\mathcal{L}_x + \hat{s}_x + \mathrm{e}^{-\hat{s}_q}\mathcal{L}_q + \hat{s}_q \tag{3.8}$$

With $\hat{s}_x = log(\sigma_x^2)$ and $\hat{s}_q = log(\sigma_q^2)$. In addition to being more stable, the $e^{-\hat{s}}$ term is positive thus representing a valid variance value. In this loss function $\hat{s}_x$ and $\hat{s}_q$ are trainable parameters which shift during training so that the losses calculated for each task is balanced.

As the lowest loss from the trainable loss function doesn't necessarily correspond to the best accuracy and best weights, a custom metric was implemented. This metric calculates the mean translation error and angle error for the whole validation data set at the end of each epoch. The weights from the epoch with the lowest combined error were saved as the best weights.

3.4 Point Cloud registration network

The second portion of the localization system was a point cloud registration network that aligns a 3D LiDAR scan taken at the vehicles current position with a highly detailed map of the operational area. For the registration network to be able to match a scan to the map, it is necessary that the scan and the crop of the map

contain approximately the same points. This was achieved by using the predicted position from the RGB-D image network to extract portions of the map close to the scan and pre-process the map to make it more similar to a LiDAR scan. Additional accuracy gains were achieved by applying a sliding-window type approach that crop multiple $n \times n$ $[m^2]$ shaped map segments around the initial prediction and try to match the crops with the scan.

3.4.1 Network design

PointNetLK [29] is a neural network for point cloud registration that promises to be both robust to bad initial alignment of the point clouds and converge much faster than traditional ICP methods. It is based on PointNet together with a modified version of the Lucas-Kanade method. The main idea behind PointNetLK is to utilize a PointNet network as an imaging function, encoding the 3D geometry in an "image" vector and then use the LK algorithm to align two of these vector "images". The novel idea of this approach is the modification of the LK algorithms so that it can be used without image gradients calculated with 2D or 3D convolutions. A visualization of the network structure can be seen in Figure 3.11. The theoretical motivation behind the design can be found in Section 2.5.

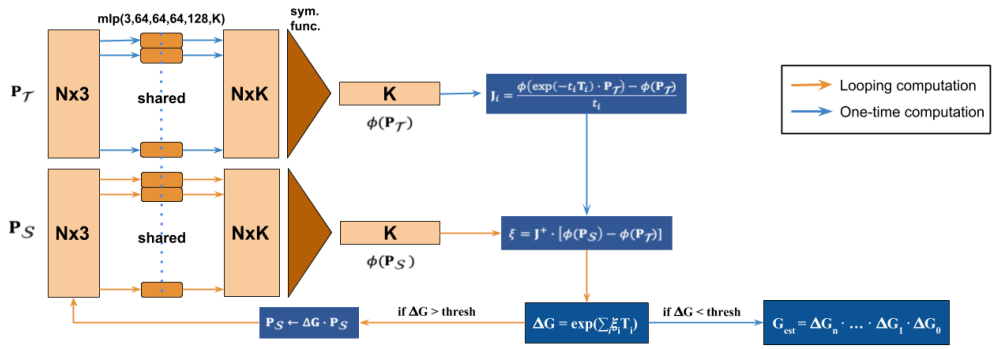


Figure 3.11: The PointNetLK network used for registration.

3.4.2 Training PointNetLK

As with all neural network, the registration network PointNetLK needs training before it will be able to do any predictions. The initial training of the MLP weights was done on a classification task. For this training, the ModelNet40 [36] data set was used which consists of point cloud models of 40 different classes. The models range from guitars to airplanes and each class has around 100 different models split into a train and a test set.

All models passed to PointNet were first normalized and centered to fit within a unit cube. The training ran until convergence which took around 16 hours. The weights of the first MLP portion of PointNet, after the classification training, was then transferred to PointNetLK and used as initialization weights for the registration training. PointNetLK was also trained on ModelNet40 models using 20 of the classes

for training and 20 for testing. The training was done by applying a varying degree of translation and rotation perturbations to each source point cloud, creating a very large data set of source and template point cloud pairs with known ground truth transformation G_{gt}. During training, PointNetLK uses a loss function that minimizes the error between the estimated transformation and the ground truth

$$||G_{est}^{-1} \cdot G_{gt} I_4||_2. \tag{3.9}$$

3.4.3 Map segment matching

To evaluate the performance after the PointNetLK training process, a map segment from the 3D map was used to test the networks ability to match point clouds with markedly different geometries than trained on. A map segment cropped from the full 3D map was used as the target and the same point cloud, transformed with a random translation and rotation, was used as the source. Images from before and after the transformation are shown in Figure 3.12.

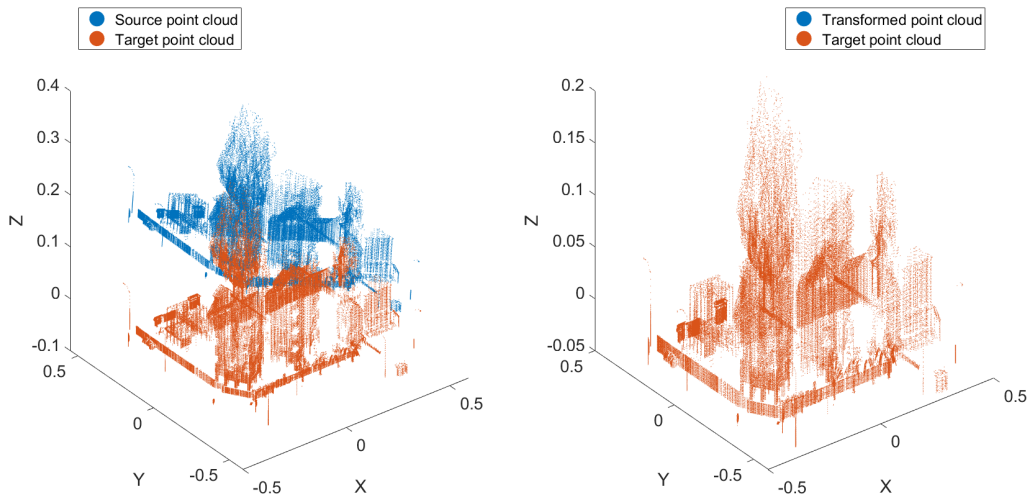


Figure 3.12: Two identical point clouds with a transformation perturbation applied to one of them. Input to the left, to the right the point clouds after applying the transformation from the PointNetLK algorithm. PointNetLK finds the exact transformation matrix between them.

The result, as seen in Figure 3.12, shows that PointNetLK has no problem matching identical map segments even though it has not been trained on such point clouds. When trying to match two different point clouds with partially overlapping surfaces, such as a map segment and a scan, the results were not as good and could often fail. This inability required that additional preprocessing had to be done to create similar looking scan and map segments.

Preprocessing for map matching To be able to match the map to a scan, some preprocessing needs to be done on the map to make it "scan-like", as described below. There were three steps used in the preprocessing pipeline for PointNetLK.

Crop The first step was to crop the map to a pre-determined size. From visual inspection of different crop sizes, 60×60 m produced a well weighted amount of geometries that were shared with and could be differentiated from the scan. The same crop size was applied to the scan to remove sparse geometries with few scan points far away from the origin of the scan.

Downsampling The map segment was then downsampled to a density similar to the scan. This was be done by using a voxel grid filter. Given a voxel size, the algorithm takes into account all occupied voxels in the grid and generates a new point by averaging all point in each respective voxel.

2.5D extraction A single LiDAR scan only contains 3D information from a single point of view, only reaching surfaces in direct line-of-sight, often called 2.5D. This 2.5D structure is rather different from the combined multi-view 3D map.
To make the 3D map segment similar to a 2.5D scan, all the points with surface normals pointing away from the predicted position was removed. In this way, most points not visible from the predicted position could be removed from the 3D segment. In this way, the map segments could be better matched with the 2.5D scans from the LiDAR. The surface normals for the points in the map are pre-calculated before run-time and aligned with the gravity vector, as described in Section 2.3.2. During matching, points that should be removed are found by taking the dot product of the vector from the 3D point, P_i, to the predicted position, $PredPos$, and the normal of the 3D point, N_{3D} as

$$
\begin{aligned}
&V_i = \text{PredPos} - P_i && \forall i \in \text{points in crop} \\
&\quad if \quad V_i \cdot N_{3D} > 0 : \\
&\quad keep \quad P_i
\end{aligned}
$$

Figure 3.13 visualizes a top view of a map segment before and after the 3D to 2.5D conversion.

3.4.4 Sliding window method

The preprocessing steps enable PointNetLK to match a scan with a map segment. But if the initial prediction was too inaccurate, PointNetLK could fail because the extracted map segment and scan could be too different. To solve the issue a sliding window approach was developed where several map segments were cropped and compared to the scan. Using the initial prediction, a larger map was cropped and several patches were extracted using a smaller window and a predefined step size.
The method helps PointNetLK to correct more and worse initial estimates as the search area increases, though with increased computational time as a payoff.

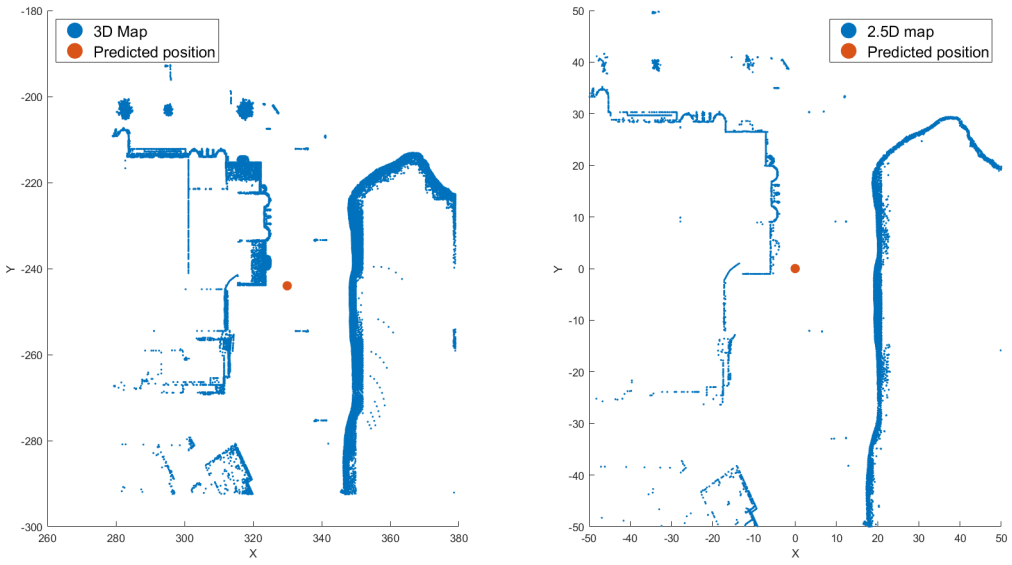


Figure 3.13: Top view of a map segment before and after 3D to 2.5D conversion.

Crop size tuning To test how large offsets PointNetLK can handle, different step sizes were tested while cropping the map segments. Step sizes of 1.7 m, 3 m and 6 m were tested. In all cases a 12×12 m area was searched around the prediction, leading to 225 iterations for 1.7 m step size, 81 iterations for 3 m step size and 25 iterations for the 6 m step size. Table 3.1 show the result.

Refinement on hard test cases		
Original prediction	6.5 m, 8.3°	7.6 m, 10.3°
Test case	Median error	Mean error
-12 m to 12 m search, 5 steps	2.7 m, 11.5°	4.3 m, 13.0°
-12 m to 12 m search, 9 steps	1.9 m, 9.5°	2.3 m, 10.7°
-12 m to 12 m search, 15 steps	1.8 m, 9.5°	2.7 m, 10.5°

Table 3.1: Results using different step size to extract map segments for matching.

On the tested step sizes, there were no big difference in performance between a step size of 3.0 m and 1.7 m, while the larger step size requires $15 \cdot 15 - 9 \cdot 9 = 144$ fewer iterations to cover the same map size. Though a performance drop was seen with the largest 6 m step size. A 3 m step size was used in the project as it was considered a good trade-off between accuracy and number of iterations. The map size to cover is purely a trade-off between accuracy and computational time since a larger search area increases our chances to correct inaccurate predictions.

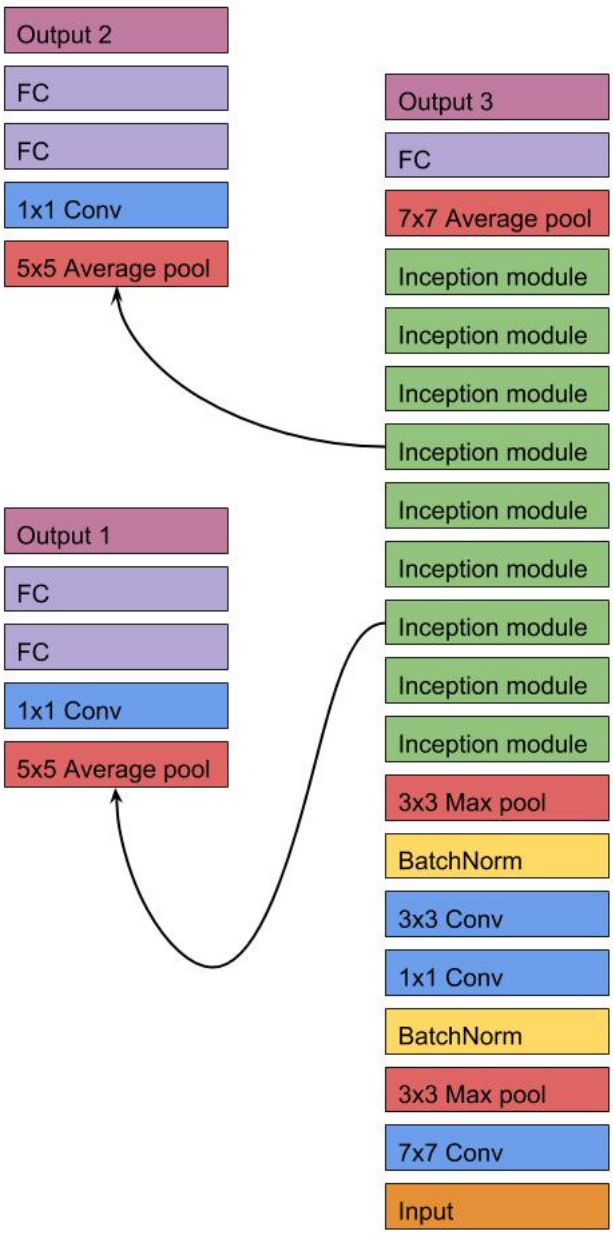


Figure 3.14: A schematic view of the GoogLeNet architecture.

4
Results

In this chapter, the results from the evaluation of the performance and characteristics of the different networks will be presented. This includes the individual performances of image networks trained on RGB, depth images and the final RGB-D network. We then present the result from the position estimate refinement with PointNetLK. Two types of test sets were created to evaluate the estimation accuracy of the networks. The first was created independently but in the same manner as the training data set, by means of an autopilot. The second was manually created by collecting data at poses not found in the training data set. This manually created data set was designed to show the ability of the RGB-D network to generalize and the potential benefits of using a PointNetLK type registration network for pose refinement.

4.1 Results from autopilot test set

Initially, the image networks were tested with two test sets gathered by using the autopilot. The first of these contained only image pairs from the clear weather setting and the second contained four different weather types. After the prediction, the error is calculated as the Euclidean distance between the ground truth pose and the predicted pose. The same test sets were used to evaluate both the single image networks and the RGB-D network.

4.1.1 Performance with clear weather data set

The first test was done on a model trained on a data set with 7600 images generated with clear weather conditions. The corresponding test set for this scenario contained 707 images.

The results presented in Table 4.1 show that the single image network is able to learn to predict a pose given either a RGB or a JET encoded depth image. The performance on both RGB and depth images show similar accuracy which implies that the network can learn to localize using both JET encoded geometry information from depth images or textures from RGB images. The result also shows that the fused information from both RGB and depth images improve the performance of the translation prediction. The angle prediction, however, is worse when using the RGB-D model.

Clear weather - Test set		
Model	Median error	Mean error
RGB	3.0 m, 2.1°	4.6 m, 3.9°
Depth	2.6 m, 2.5°	4.3 m, 3.8°
RGB-D	1.6 m, 5.0°	3.2 m, 8.3°

Table 4.1: Mean and median error with single and RGB-D networks. The results show that the additional information from both image types improve the translation accuracy but deteriorate the angle predictions.

From the histograms in Figure 4.1-4.3, this result is made clearer. The RGB-D network produces more accurate translation predictions with fewer large error predictions.

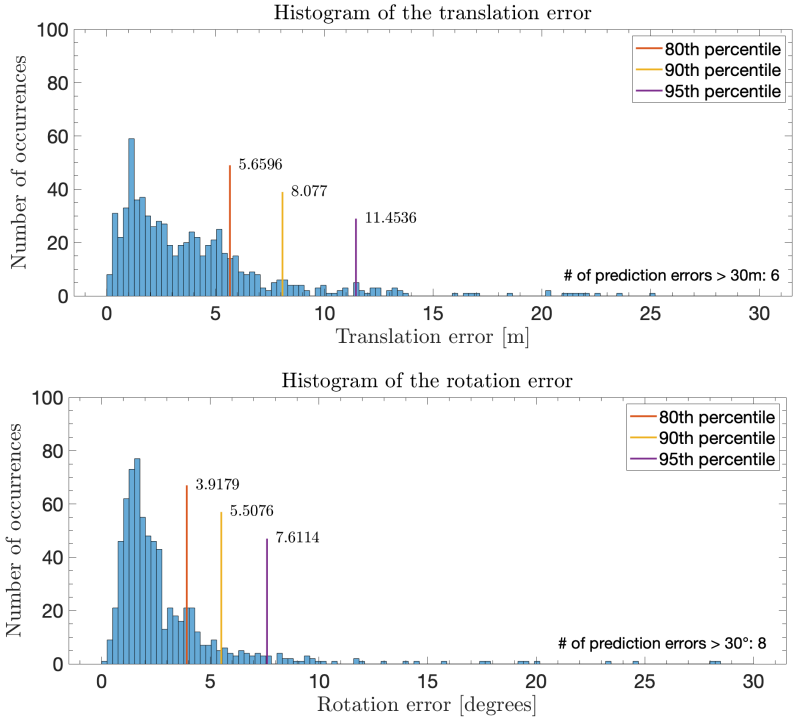


Figure 4.1: RGB network, clear weather, Every bar is 0.25 $m/degree$ wide.

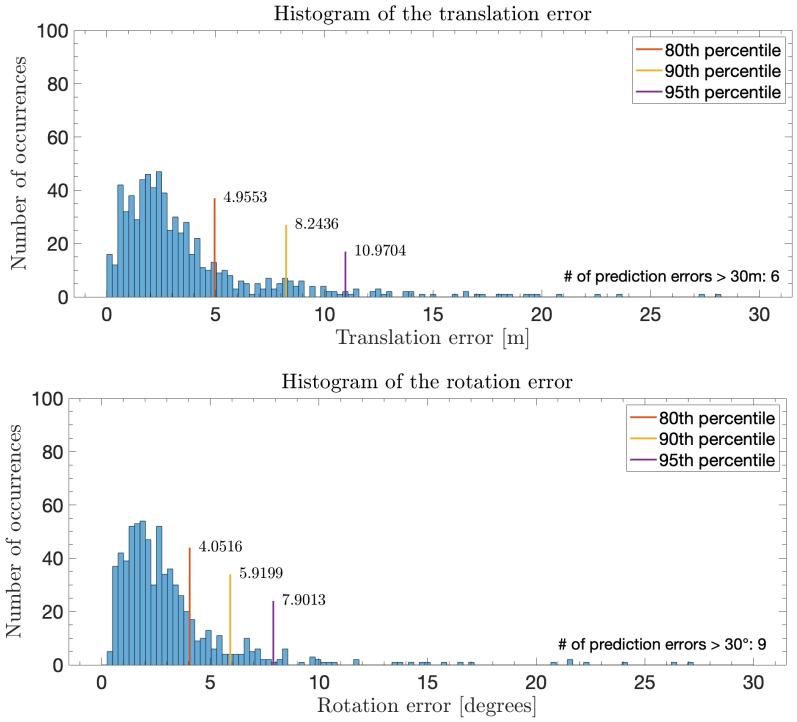


Figure 4.2: Depth network, clear weather, Every bar is 0.25 $m/degree$ wide.

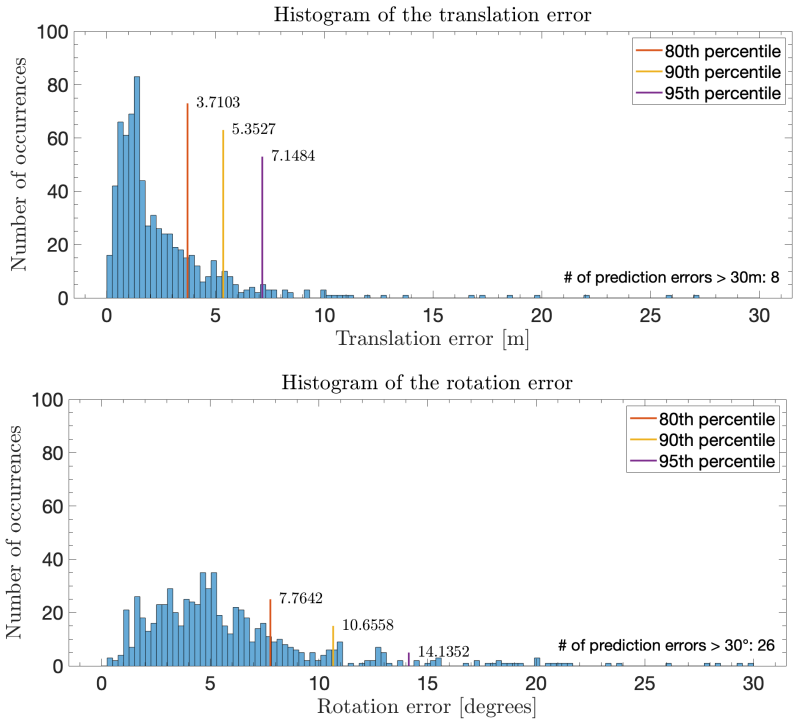


Figure 4.3: RGB-D network, clear weather, Every bar is 0.25 $m/degree$ wide.

4.1.2 Performance with varying weather

The second test was done on a model trained on a data set with varying weather. The test set contained 1011 images with an equal distribution of the same four weather scenarios as in the training set. In this case, the RGB images have additional noise added in the form of new lighting conditions and textures which should increase the difficulty of the task. The depth sensor has however not been affected by this noise as the geometries are unchanged.

Varying weather - Test set		
Model	Median error	Mean error
RGB	4.8 m, 3.1°	8.6 m, 6.5°
Depth	3.1 m, 3.3°	4.8 m, 4.9°
RGB-D	1.7 m, 2.6°	3.3 m, 4.9°

Table 4.2: Mean and median error on RGB, depth and RGB-D network on the data set with weather variation. The RGB models prediction errors increases more compared to the depth model when tasked to predict in the varying weather scenarios.

As expected, the RGB network saw decreased performance with weather variation while the depth network produces similar results as in the first case. The result also shows that when fused together, the combined information from both modalities results in a better prediction than any of the single networks. Histograms of the rotation error and translation error are presented in Figure 4.4-4.6.

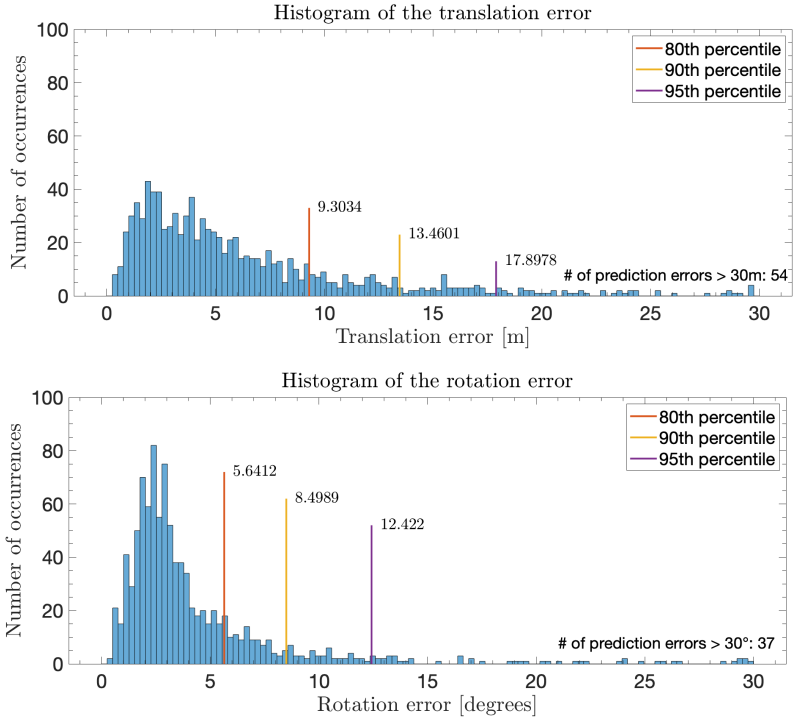


Figure 4.4: RGB network, varying weather, Every bar is 0.25 $m/degree$ wide.

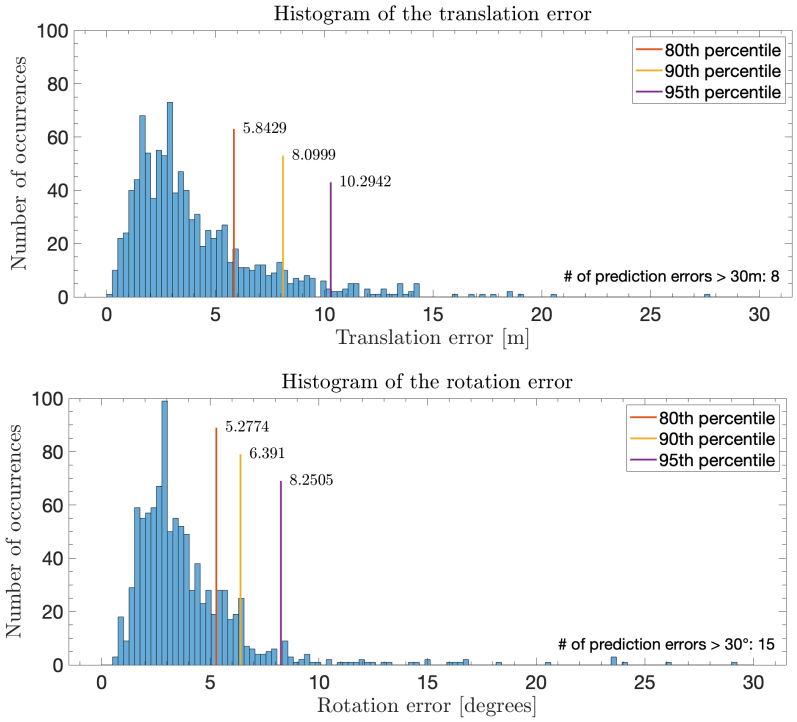


Figure 4.5: Depth network, varying weather, Every bar is 0.25 $m/degree$ wide.

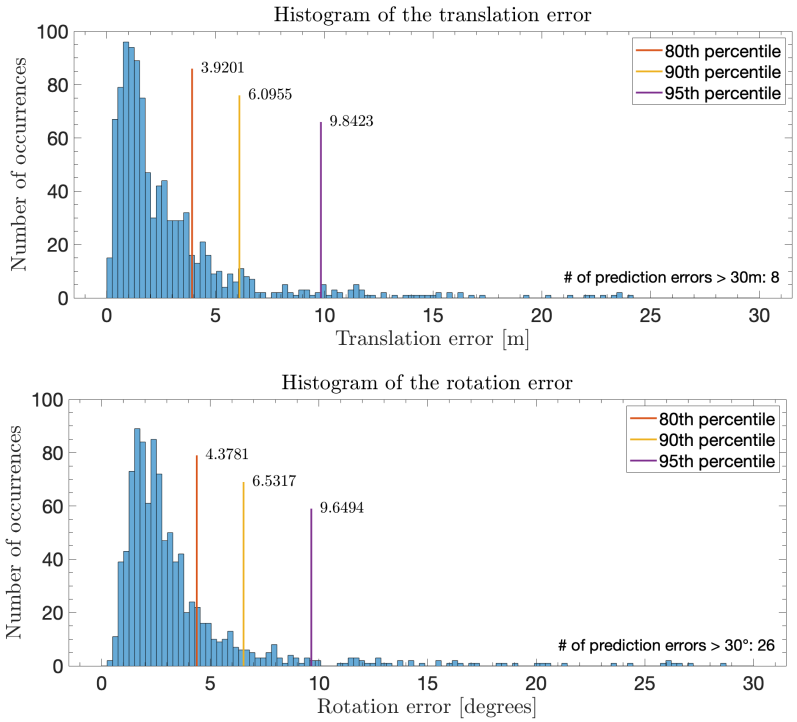


Figure 4.6: RGB-D network, varying weather, Every bar is 0.25 $m/degree$ wide.

4.1.3 Performance with PointNetLK refinement on varying weather

Refining the initial prediction from the RGB-D network with PointNetLK led to the results presented in Table 4.2 and Figure 4.7. The result shows that PointNetLK does improve the translation accuracy but struggles to improve the angular prediction. The translation histogram in Figure 4.7 show that there still exists outliers, but there are fewer predictions in the range 5-15 meters compared to the initial predictions, shown in Figure 4.6.

Varying weather - Test set		
Model	Median error	Mean error
RGB-D	1.7 m, 2.6°	3.3 m, 4.9°
Refined RGB-D	1.6 m, 5.1°	3.1 m, 8.7°

Table 4.3: Table with results from the refinement network PointNetLK compared to the RGB-D prediction network without refinement.

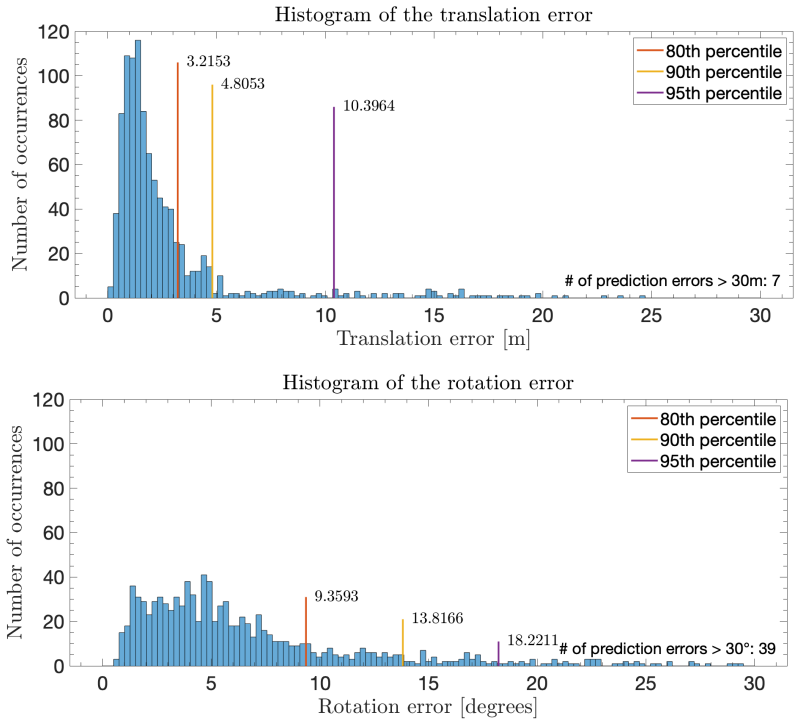
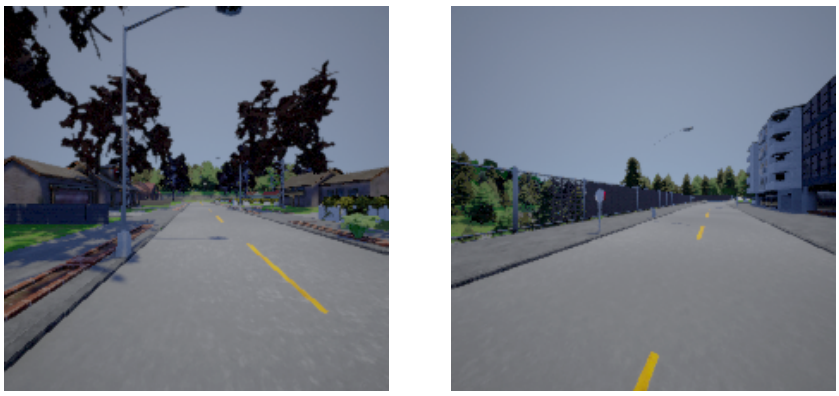


Figure 4.7: Refined prediction from RGB-D network with PointNetLK, varying weather, Every bar is 0.25 $m/degree$ wide.

For all cases, a 3m step size was used covering an area of 84x84m which resulted in a total of 81 map to scan comparisons.

4.2 Results from challenging data set

To validate the RGB-D and PointNetLK networks further, a challenging test set was created consisting of 26 different poses were the car is shifted perpendicular and/or not aligned parallel with the road. This type of test is interesting as the images will be fairly different from what exists in the training data set, which would indicate to what degree the RGB-D network has learned to generalize. Two examples from the test set can be seen in Figure 4.8. The rest of the collection can be found in Appendix A.1



(a) Example image, car driving in the opposite lane

(b) Example image, car rotated 28° counter clockwise in lane

Figure 4.8: Example images from the hard test data set.

4.2.1 Performance with RGB-D and PointNetLK networks

Testing the two networks on the challenging data set show that the RGB-D network is considerably less accurate when encountered with shifted images. From Table 4.4, the reliability of the predictions is shown to vary considerably, even though the input images are changed in a similar way throughout the whole data set. From the corresponding images in Appendix I, we see that these estimates are taken from viewpoints likely to be encountered in a dynamic setting. These results will be discussed further in Section 5.1

Refining the initial predictions with PointNetLK led to the results, also presented in Table 4.4. This test case shows the benefit of using PointNetLK since many large prediction errors is improved. Though, when the prediction error is too large, PointNetLK is not able to refine it. This is likely due to the search area of the sliding window algorithm being too small to extract a map segment similar to the scan.

Case	RGB-D pos. error	w/ PointNetLK pos. error	RGB-D angle error	w/ PointNetLK angle error
1	140.1 m	120.8 m	31.9°	18.2°
2	25.4 m	8.9 m	18.0°	25.0°
3	5.6 m	0.6 m	4.9°	5.4°
4	32.4 m	19.3 m	21.3°	23.2°
5	7.5 m	1.9 m	3.8°	10.5°
6	6.1 m	1.0 m	7.3°	7.7°
7	116.8 m	131.6 m	65.5°	72.6°
8	26.2 m	7.5 m	11.1°	12.0°
9	4.6 m	1.9 m	11.5°	11.7°
10	11.6 m	3.1 m	28.8°	47.6°
11	58.2 m	71.6 m	17.1°	39.3°
12	6.9 m	1.5 m	5.1°	4.7°
13	0.9 m	2.6 m	8.9°	19.5°
14	24.6 m	20.7 m	28.9°	31.7°
15	33.0 m	35.6 m	26.5°	172.2°
16	5.6 m	2.1 m	3.1°	6.2°
17	3.3 m	10.1 m	21.6°	32.9°
18	46.0 m	69.7 m	10.2°	19.4°
19	8.3 m	3.3 m	19.0°	11.9°
20	8.2 m	1.8 m	19.8°	14.7°
21	14.1 m	2.0 m	3.6°	3.4°
22	72.7 m	64.7 m	20.9°	31.8°
23	8.8 m	2.4 m	1.7°	4.6°
24	2.4 m	2.4 m	22.5°	4.6°
25	8.2 m	1.2 m	32.5°	19.1°
26	22.1 m	2.8 m	26.9°	15.0°
Mean	26.9 m	**22.7 m**	**17.7°**	25.9°
Median	10.2 m	**3.0 m**	17.5°	**16.6°**

Table 4.4: Summary of test results. PointNetLK can refine the translation estimate in several cases.

4.3 Visual results of point cloud matching

Figure 4.9 shows a map segment and a scan prior to and after the refinement process. This shows that PointNetLK is able to find a transformation that better match two point clouds even though they contain non-identical points and with partial overlap of the 3D structure.

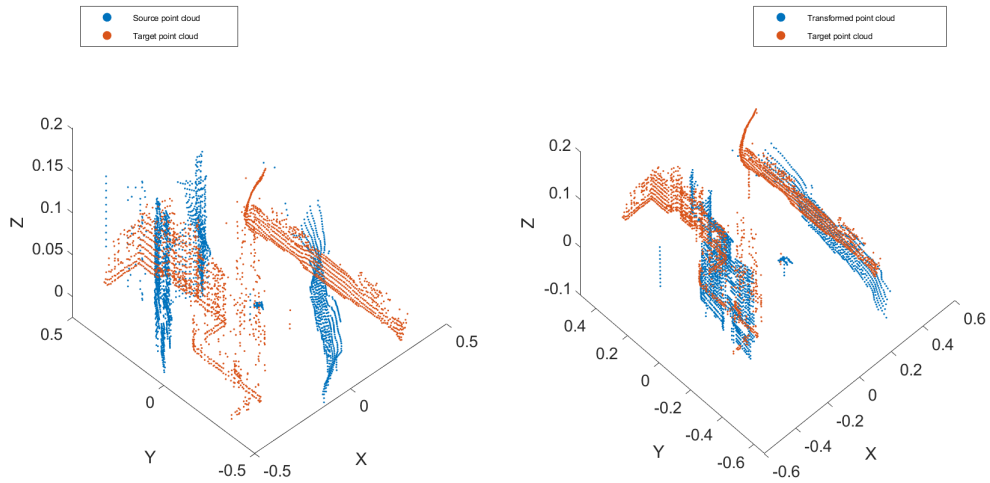


Figure 4.9: Case were PointNetLK match two different point clouds. In the left image, the blue LiDAR scan point cloud is overlaid on the red map segment produced from the RGB-D networks initial prediction. The right image shows the result after transforming the source point cloud with the found transformation from PointNetLK.

4.3.1 Failure cases

In this section, cases were PointNetLK failed to refine the point clouds are visualized and discussed.

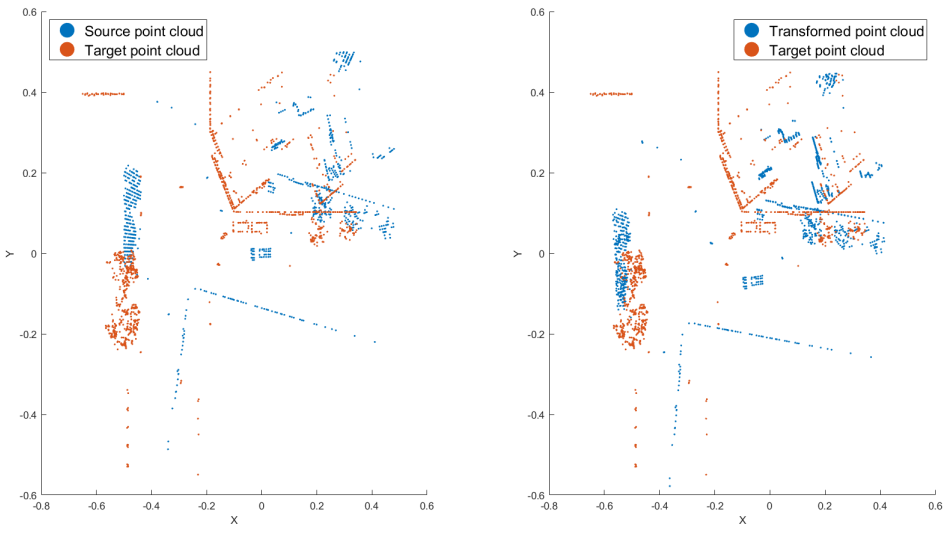


Figure 4.10: Case when PointNetLK were not able to refine the pose accurately. Input point clouds to the left, refined points clouds to the right.

In this case, PointNetLK is able to improve the prediction but still produces an error

at around 5 m. The scan and the map segment are too different for PointNetLK to refine it further. The large differences between the map segment and the scan appear when the 3D map is converted to a 2.5D representation. The initial prediction is so inaccurate that points being removed in the 2.5D conversion actually appear in the scan and vice versa.

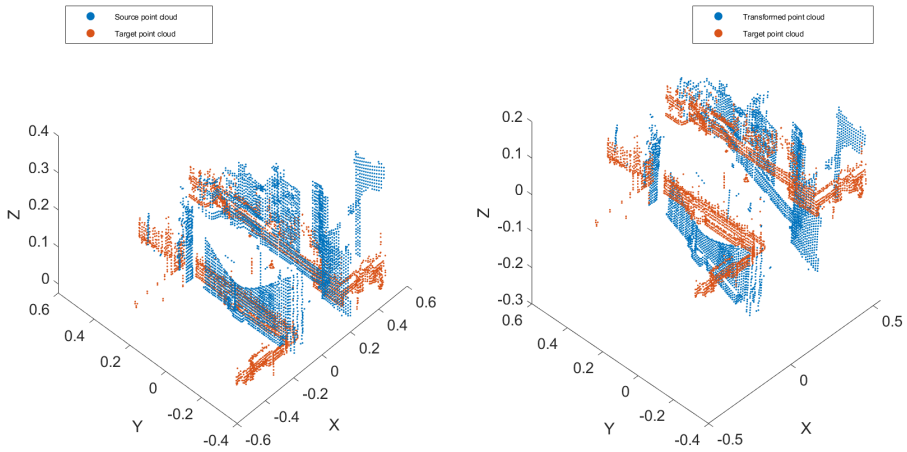


Figure 4.11: Case when PointNetLK made the position estimate worse. The scan contains points higher up than the map. Input point clouds to the left, refined points clouds to the right.

In this case, the initial guess is already accurate but the refinement process makes it worse. The scan and the map segment are from the same area, but the scan contains points higher up than the map segment does. The difference causes PointNetLK to "pull down" the scan, resulting in a worse position estimate. In essence, since the prediction network is trained in a flat environment, the z-error in the prediction is always spot on and the z-error could actually be disregarded in the refinement. However, this failure case shows that a difference in geometry in the z-axis direction could be problematic in other scenarios as it could worsen the refinement results. This can be seen in a histogram showing the xyz and xy error respectively.

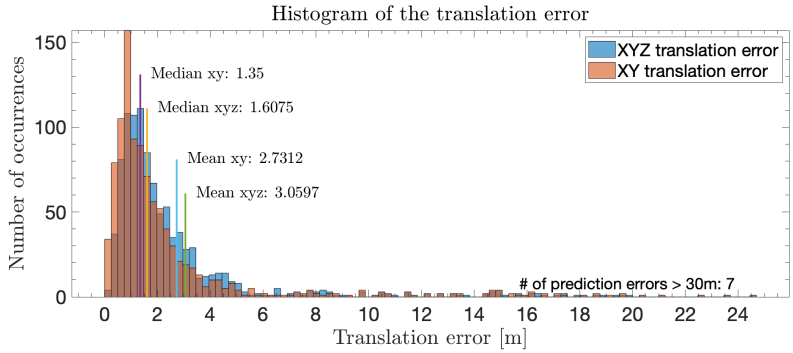


Figure 4.12: RGB-D network and refined, varying weather, Every bar is 0.25 m wide.

5
Discussion

In this chapter, the results from the previous chapter are discussed. We try to explain some of the observed properties and discuss problems found in the performance.

5.1 RGB-D network performance

We see several interesting properties in the multi-modal RGB-D network compared to the uni-modal. We see increased translation prediction performance in all of the test sets. Most noticeably, the gains grow larger with more difficult scenarios. The increased robustness to varying scenes was one of the main motivation when pursuing multi-modal image networks, which makes these results encouraging.

As was already mentioned in Section 1.3, the focus with this thesis was never to reach the accuracy performance of the current state-of-art network such as VLocNet++ [14], but rather to investigate if RGB-D fusion in outdoor environments could lead to any performance gains. While this seems to be the case, we also saw that running the image networks on poses that were less similar to what is found in the training data set produced much less accurate predictions or could often fail entirely. PointNetLK was added to improve some of these inaccurate predictions, though it was not able to refine very large errors in a reasonable computational time. We believe that a larger, more diverse training data set could solve this issue, while unfortunately also limiting the feasibility of using pose prediction networks in real world applications. In a recently published paper, Sattler et al. [42], draws similar conclusions with a thorough theoretical explanation on the limitations of pose prediction CNNs, arguing that they are actually more related to image retrieval networks than methods that regress the pose from 3D geometries. This, in principle, says that when the network is fed a new query image, the estimation will be constituted of a combination of previously learned poses of similar images. This, in turn, means that query images taken at poses which is not in the set of combinations will not be accurately predicted as the network does not actually learn the spatial relation of similar images, rather retrieve similar poses.

5.2 PointNetLK performance

PointNetLK shows great potential with the ability to find the transformation between similar point clouds and thus refine a pose. The big benefits of PointNetLK is

the fact that the computational time is much less than commonly used ICP methods [29] and it doesn't need to be retrained for different point clouds.

A substantial limitation is, however, that the inputs have to be similar for PointNetLK to work well. Due to this reason, a 3D to 2.5D conversion and sliding window crop comparison were needed to achieve high accuracy, resulting in a lot more computations. This data preprocessing needed to create similar point cloud was dependent in a fairly good initial prediction. If the predicted position is bad or unfortunately guessed, the 2.5D transformation algorithm, described in Section 3.4.3, can remove prominent features visible in the scan and reduce the possibility for a good match, as seen in Figure 4.10. If more tuning had been performed on the creation of the 2.5D segments this would hopefully also correct the large angular error that we see occur during refinement.

5.2.1 Network training

PointNetLK was only trained to match two identical objects with different pose in 3D space. Even though the inputs in this scenario are not identical, the algorithm is often able to find a good transformation between them. It would be very interesting to see if the matching process could be improved if the network was trained on matching scans to a map. Though it is a tedious process to create a large enough data set which requires a lot of manual work and since the matching worked without this training it was not prioritized.

5.3 Lack of benchmarking test set

The customary way of comparing the performance of neural networks is to benchmark them against some established data set. The combination of RGB and depth images together with a high resolution 3D map of a city were nothing we were able to find, wherefore, a comparison to a public benchmark have been omitted. This lack of available data also resulted in all development being done on simulated data. While the virtual environment was sophisticated, there is still a big difference between simulated and the real world data, therefore it is hard to say how this system would perform in a real world scenario.

5.4 Execution times

One of the largest benefits with the use of neural networks for localization is the fast and constant inference time. A prediction with the RGB-D Network takes about 11 ms on a Nvidia GeForce RTX2080 Ti. Each PointNetLK iteration takes 0.2-0.5 seconds depending on the number of 3D points in the current scan and map segment. Though the total running time depends on the number of PointNetLK iterations to run. There is a big trade-off between speed and accuracy, as a larger search radius and more iterations lead to an improved chance to correct bad initializations. One idea to reduce the computational time was to introduce an early

stopping routine, which would stop the process if it converged or if a metric were below some threshold. Though none of these methods performed well when tested and since the focus has been on the accuracy performance rather than the computational speed the issues of finding a stable metric and tuning thresholds were not prioritized.

6
Conclusion

The purpose of this thesis has been to explore one way of combining RGB and depth images and evaluate to what degree multiple modalities would be beneficial when training end-to-end neural networks. We saw two main benefits with this approach.

More accurate predictions by learning to fuse information from two modalities The two systems were shown to complement each other well as the image network often could predict what region images were taken, but produced fairly inaccurate results when placed in poses not occurring in the training data set. While the registration network was not able to compare single scans to the entire map, it could use the prediction to extract comparable regions, reducing the number of comparisons needed. The two networks were seen to complement each other and together they produced a more accurate pose.

Self-contained pose estimation system suitable for outdoor areas With reservation that we perform all development and testing in a simulated world, we are able to localize in an outdoor environment where we are constrained to move not too far from where the training data was gathered. We had hoped that an end-to-end trained CNN might share the ability to generalize about image inputs which CNNs have been shown to do in other applications [41]. Though, results show that it fails to produce a viable initial prediction when the input image is taken at poses that largely differs from what is found in the training data. This would indicate that the network has not learned a "virtual map" but instead finds similarities to previously seen images and the prediction is rather a combination of previously learned poses.

6.1 Future Work

Based on our work we see that point cloud registration methods based on PointNetLK can improve the position estimate greatly. PointNetLK shows great potential and for future work, it would be interesting to see if the matching performance can be increased by training the network on partially visible scans in a map. Hopefully, it would increase the robustness of the system and lower the computational time, as the number of iterations could be decreased.

Regarding the RGB-D network, as we found issues when using the image network to predict on poses not found in the training data, additional work should preferably focus on making it robust to pose changes. Since PointNetLK is able to correct many

inaccurate predictions, this could open up the possibility to use different methods to generate an initial pose. We suggest that the future prediction system should be designed to learn a scene representation, rather than continuing making more accurate estimations based on PoseNet.

Bibliography

[1] R. Hussain, and S. Zeadally, "Autonomous Cars: Research Results, Issues and Future Challenges", *IEEE Communications Surveys Tutorials*, 2018, doi: 10.1109/COMST.2018.2869360, ISSN: 1553-877X.

[2] S. Tang, V. Kumar, "Autonomous Flight", *In: Annual Review of Control, Robotics, and Autonomous Systems*, volume 1, pages 29-52, 2018, doi: 10.1146/annurev-control-060117-105149.

[3] C. Benson, P. Sumanth, A. Colling, "A Quantitative Analysis of Possible Futures of Autonomous Transport", *In: INEC 2018 Conference*, arXiv:1806.01696.

[4] A. Lawrence, "An Outline of Inertial Navigation," *In: Modern Inertial Technology, Mechanical Engineering Series*, 1998, Springer, New York, NY. doi: 10.1007/978-1-4612-1734-3_2

[5] D. Nistér, O. Naroditsky, J. Bergen, "Visual odometry," *In: Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2004)*, vol. 1, pp. I–652 (2004)

[6] G. Grisetti, S. Grzonka, C. Stachniss, P. Pfaff, W. Burgard, "Efficient estimation of accurate maximum likelihood maps in 3D," *In: IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2007 (IROS 2007), pp. 3472–3478 (2007)

[7] J. Han, L. Shao, D. Xu and J. Shotton, "Enhanced Computer Vision With Microsoft Kinect Sensor: A Review," *In: IEEE Transactions on Cybernetics*, vol. 43, no. 5, pp. 1318-1334, Oct. 2013.

[8] L. Shaoa, Z. Caic, L. Liud, K. Lue, "Performance Evaluation of Deep Feature Learning for RGB-D Image/Video Classification", *In: Information Sciences, 385*. pp. 266-283. ISSN 0020-0255, 2017.

[9] B. Behroozpour, P. A. M. Sandborn, M. C. Wu and B. E. Boser, "Lidar System Architectures and Circuits," *In: IEEE Communications Magazine*, vol. 55, no. 10, pp. 135-142, Oct. 2017. doi: 10.1109/MCOM.2017.1700030

[10] Lumotive (2019), [Online]. Available: https://www.lumotive.com/

[11] D. Lowe, "Distinctive image features from scale-invariant keypoints," *In: International Journal of Computer Vision*, 60, 2 (2004), pp. 91-110.

[12] R. Mur-Artal and J. D. Tardós "ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras," *In: IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255-1262, Oct 2017. doi: 10.1109/TRO.2017.2705103

[13] M.A. Fischler and R.C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *In: Communications of the ACM*, 24(6):381–395, 1981.

[14] N. Radwan, A. Valada, W. Burgard, "VLocNet++: Deep Multitask Learning for Semantic Visual Localization and Odometry", *In: IEEE Robotics and Automation Letters (RA-L)*, 3(4):4407-4414, 2018.q

[15] A. Valada, N. Radwan, W. Burgard, "Deep Auxiliary Learning for Visual Localization and Odometry", *In: IEEE International Conference on Robotics and Automation (ICRA)*, 2018.

[16] J. Shotton, B. Glocker, C. Zach, S. Izadi, A. Criminisi, and A. Fitzgibbon. "Scene coordinate regression forests for camera relocalization in RGB-D images." *In: Computer Vision and Pattern Recognition (CVPR)*, 2013

[17] A. Kendall, M. Grimes, and R. Cipolla, "Posenet: A convolutional network for real-time 6-dof camera relocalization," *In: ICCV*, 2015.

[18] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," arXiv preprint arXiv:1409.4842, 2014

[19] E. Rublee, V. Rabaud, K. Konolige, G. Bradski, "ORB: an efficient alternative to SIFT or SURF," *In: IEEE International Conference on Computer Vision (ICCV)*, 2011

[20] H. Bay, A. Ess, T. Tuytelaars, L. Van Gool, "SURF: Speeded Up Robust Features", *In: Computer Vision and Image Understanding (CVIU)*, Vol. 110, No. 3, pp. 346–359, 2008

[21] S. Zia, B. Yüksel, D. Yüret, Y. Yemez, "RGB-D object recognition using deep convolutional neural networks", *In: Proc. IEEE Int. Conf. Comput. Vis. Workshop (ICCVW)*, pp. 887-894, Oct. 2017.

[22] J. Engel, T. Schoeps, D. Cremers, "LSD-SLAM: Large-scale direct monocular SLAM", *In: ECCV*, 2014.

[23] J. Engel, V. Koltun, D. Cremers, "Direct Sparse Odometry", *In: IEEE Transactions on Pattern Analysis and Machine Intelligence*, Mar. 2018.

[24] J. Shotton, B. Glocker, C. Zach, S. Izadi, A. Criminisi, A. Fitzgibbon, "Scene Coordinate Regression Forests for Camera Relocalization in RGB-D Images", *In: IEEE Conference on Computer Vision and Pattern Recognition*, Mar. 2013.

[25] X. Song, L. Herranz, S. Jiang, "Depth CNNs for RGB-D scene recognition: learning from scratch better than transferring from RGB-CNNs", *In: AAAI*, 2017.

[26] A. Eitel, J. T. Springenberg, L. Spinello, M. A. Reidmiller, W. Burgard, "Multimodal Deep Learning for Robust RGB-D Object Recognition", *In: IROS 2015*, 2015.

[27] A. Dosovitskiy, G. Ros, F. Codevilla, A. López, V. Koltun, "CARLA: An Open Urban Driving Simulator", *In: CoRL*, 2017.

[28] C. Qi, H. Su, K. Mo, L. Guibas, "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation", arXiv:1612.00593v2, 2017.

[29] Y. Aoki, H. Goforth, R. A. Srivatsan, S. Lucey, "PointNetLK: Robust & Efficient Point Cloud Registration using PointNet", arXiv:1903.05711v1, 2019.

[30] P. Besl, N. McKay, "A Method for Registration of 3-D shapes", *In: IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1992.

[31] B. Lucas, T. Kanade, "An Iterative Image Registration Technique with an Application to Stereo Vision", *In: Proceedings of Imaging Understanding Workshop*, pp. 121-130 (1981).

[32] S. Baker, I. Matthews, "Lucas-Kanade 20 Years On: A Unifying Framework", *In: "International Journal of Computer Vision 56(3)"*, p221–255, 2004.

[33] F. Chollet et al, "Keras," https://keras.io, 2015.

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

[35] D. P. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization," *In: International Conference on Learning Representations (ICLR)*, 2015

[36] Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang and J. Xiao, "3D ShapeNets: A Deep Representation for Volumetric Shapes" *In: Proceedings of 28th IEEE Conference on Computer Vision and Pattern Recognition (CVPR2015)*, 2015

[37] D. E. Rumelhart, G. E. Hinton, R. J. Williams, "Learning Representations by Back Propagating Errors", *In: Nature* volume 323, p533-536, 1986

[38] Cybenko, G.V. "Approximation by Superpositions of a Sigmoidal function". van Schuppen, Jan H. In: Mathematics of Control, Signals, and Systems. Springer International. pp. 303–314, 2006

[39] M. Ester, H-P. Kriegel, J. Sander, X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise", *In: Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining*, Portland, Oregon, 1996.

[40] F. Hemprech, C. Schnörr, B. Jähne, "Pattern recognition : 29th DAGM symposium," Heidelberg, Germany, September 2007, p164-165 : proceedings, (electronic resource)

[41] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva. “Learning Deep Features for Scene Recognition using Places Database.”, *In: Advances in Neural Information Processing Systems 27 (NIPS)*, 2014. PDF Supplementary Materials

[42] T. Sattler, Q. Zhou, M. Pollefeys and L. Leal-Taixe. "Understanding the Limitations of CNN-based Absolute Camera Pose Regression", *In: Computer Vision and Pattern Recognition (CVPR)*, 2019

[43] A. Kendall, Y. Gal, and R. Cipolla, "Multi-task learning using uncertainty to weigh losses for scene geometry and semantics.", *In: CVPR*, 2018.

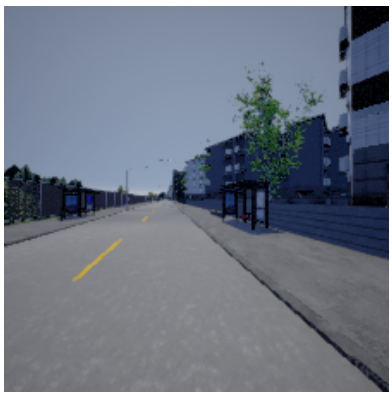
[44] A. Kendall and R. Cipolla, "Geometric loss functions for camera pose regression with deep learning", *In: ICRA*, 2016.

A

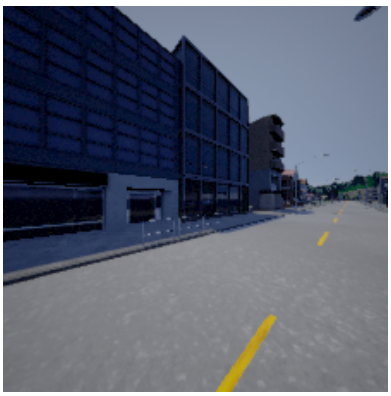
Appendix 1

A.1 Challenging test dataset

Below are all images used in the hard dataset presented. They can result from the car being either in the wrong lane or not aligned with the road as it is in the training images.



(a) Hard test case 1



(b) Hard test case 2

(c) Hard test case 3

(d) Hard test case 4

(e) Hard test case 5

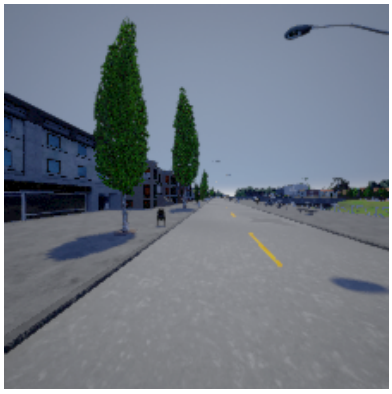
(f) Hard test case 6

(g) Hard test case 7

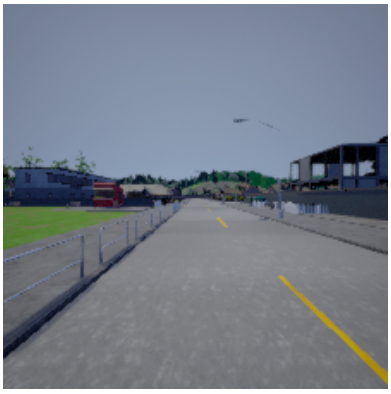
(h) Hard test case 8

(i) Hard test case 9

(j) Hard test case 10



(k) Hard test case 11

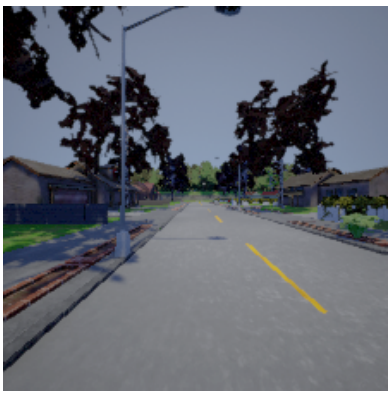


(l) Hard test case 12

(m) Hard test case 13

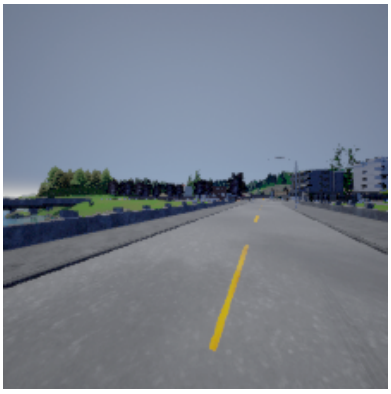
(n) Hard test case 14

(o) Hard test case 15



(p) Hard test case 16

(q) Hard test case 17



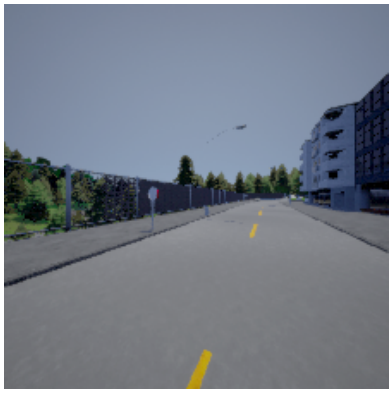
(r) Hard test case 18

(s) Hard test case 19

(t) Hard test case 20

(u) Hard test case 21

(v) Hard test case 22



(w) Hard test case 23

(x) Hard test case 24

(y) Hard test case 25

(z) Hard test case 26