NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Jonas Strand Aasberg

Machine Learning using High Resolution Zivid Point Clouds on a High Performance Cluster

Master's thesis in Mechanical Engineering
Supervisor: Lars Tingelstad
Co-supervisor: Eirik Njåstad
June 2022

Jonas Strand Aasberg

Machine Learning using High Resolution Zivid Point Clouds on a High Performance Cluster

Master's thesis in Mechanical Engineering
Supervisor: Lars Tingelstad
Co-supervisor: Eirik Njåstad
June 2022

Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Acknowledgements

This Master's thesis is the result of my integrated five year education within Mechanical Engineering at the Department of Mechanical and Industrial Engineering at the Norwegian Institute of Science and Technology.

First of all, I would like to thank my main supervisor Lars Tingelstad for his insight and guidance through this last year. His knowledge and commitment to the field of Robotics is what lead me to pursue a thesis in this field. Further, I would like to thank my co-supervisor Eirik Njåstad for helping me in finishing this thesis. Additionally, I would like to extend my gratitude to Martin Ingvaldsen for providing many interesting insights into the Zivid camera system.

I would also like to thank my friends for sharing five memorable years at NTNU with me, and my family for supporting me through my studies. Finally, I would like to thank my girlfriend Nora for all her love and support these last five years.

Summary

In this Master's thesis, methods and work are presented that enable the use of high resolution Zivid point clouds in machine learning using the Minkowski Engine library created by Chris Choy. Additionally, a data set consisting of Zivid point clouds was created in order to perform training in the Fully Convolutional Geometric Features (FCGF) and Deep Global Registration (DGR) libraries.

The training of both the FCGF and DGR models using Zivid point clouds was performed in a Singularity container on the NTNU Idun high performance cluster. The container was created to enable the utilization of the latest Nvidia A100 GPU's for training the neural networks in PyTorch. Additionally, the container is able to perform the preprocessing of the raw Zivid point clouds in order to create a data set for use with the FCGF and DGR libraries.

The Zivid point cloud data set was created by scanning multiple objects from different angles. The data set presented in this thesis is limited in terms of size and diversity, but functions as a proof of concept when training neural networks. Each scan in the data set is close to an order of magnitude larger in terms of megabytes than in the 3DMatch data set originally used in FCGF and DGR.

The Zivid data set enabled the FCGF and DGR libraries to be used with Zivid point clouds as input data. We were able to perform training using the Zivid data set and the results indicated problems in loss calculation. The FCGF model was having issues with calculating loss which in turn lead to no apparent progress in the training process. It is assumed that tuning the loss related hyperparameters in the trainer configuration can lead to improvements in loss calculations. This is proposed as an interesting topic off study going forward. The results from training the DGR model also indicated loss related errors. Although, since the DGR trainer requires a functioning FCGF model it is assumed that these errors are related to feature extraction rather than only failure to perform registration on the point clouds.

Sammendrag

I dette masterprosjektet presenteres det metoder og arbeid som muligjør bruken av høyoppløste Zivid punktskyer i maskinlæringsbiblioteket Minkowski Engine skrevet av Chris Choy. I tillegg har det blitt laget et datasett bestående av Zivid punktskyer til å trene nevrale nettverk med kodebibliotekene Fully Convolutional Geometric Features (FCGF) og Deep Global Registration (DGR).

Treningen av både FCGF og DGR modellene med Zivid punktskyer ble utført i en Singularity container på NTNU Idun high performance cluster. Containeren ble bygget for å kunne ta i bruk Nvidia sin siste CUDA arkitektur og dermed også A100 GPUene på Idun til å trene nevrale netverk i PyTorch. I tillegg inneholder containeren de nøvendige bibliotekene til å gjennomføre preprosessering av rådataene fra Zivid 3D-kameraet for å kunne bruke punktskyene til trening i FCGF og DGR bibliotekene.

Zivid punktsky-datasettet ble laget ved å skanne flere objekter fra ulike vinkler. Datasettet som presenteres i denne oppgaven er begrenset med tanke på størrelse og diversitet. Det fungerer som et konseptbevis for å trene modellene og viser at det er mulig å generere datasettet. Hvert 3D-bilde i datasettet er omtrent en størrelsesorden større enn bildene i 3DMatch datasettet som originalt ble brukt i FCGF og DGR publikasjonene.

Zivid datasettet gjorde det mulig å bruke FCGF og DGR bibliotekene med Zivid punktskyer som inndata. Vi fikk til å kjøre treningsprogrammet som hører til FCGF biblioteket og resultatene indikerte at det var problemer med kalkuleringen av feil for hver epoke. Dette problemet førte til at det ikke var noen fremgang i ytelse for det nevrale nettverket. Vi antar at justering av de feilrelaterte hyperparameterene kan føre til forbedringer i feilkalkuleringen. Dette foreslås som et interessant tema for videre arbeid. Resultatene fra trening av DGR modellen indikerte også problemer med utregningen av feil. Siden DGR treningen er avhengig av en velfungerenede FCGF modell antas det at problemene i DGR er relaterte til feature extraction i stedet for problemer med å utføre registrering på punktskyene.

Contents

List of Figures

Chapter 1.

Introduction

1.1. Problem Statement

The future of the internet is assumed to be highly dependent upon 3D-data. The Metaverse is said to be an all encompassing user experience with virtual or augmented reality capabilities. As the fidelity and detail of available 3D-resources increase, and games and simulations creeps up on photo realism, the need for efficient solutions for processing such data skyrockets. Today, large corporation's such as Spotify, Meta (Facebook) and Tesla use machine learning and AI to improve the user experience of their products. The data that is available for such corporation's may shift towards predominantly 3D-based data and therefore also dramatically increase the need for specialized machine learning libraries focused on handling 3D-data.

When performing computations on large amounts of 3D-data, a common and efficient approach is to use GPU's rather than CPU's for their advantage in parallel computing. For even more computing capability, one can utilize a high performance cluster. Clusters typically offer unparalleled performance and are often the go-to solution for large scale machine learning. However, using specific libraries for machine learning on high performance clusters can be a challenge. Such libraries often have many specific requirements which can be difficult to manage on a system wide level. A possible solution is the use of predefined and reproducible virtual environments such as Docker or Singularity containers. Such containers can be defined using recipes that specify which version of software to install and use.

Registration of point clouds is a common problem in 3D scene reconstruction from real world data. Registration is the process of making two different 3D scans overlap to create a larger and coherent 3D scene than the two input scans. This allows for higher detail 3D scenes to be constructed using multiple scans.

Registration is dependent on common points in the two input scans, such point-pairs can be found using feature extraction and matching in feature space. In this thesis, the goal is to provide the necessary foundations for performing registration with high resolution point clouds captured with a Zivid 3D camera on a high performance GPU cluster.

1.2. Related Work

This section primarily presents some of the work of Christopher Choy (NVIDIA). He has been able to represent 3D-data as sparse tensors on the GPU for increased performance and usability in machine learning. Additionally, he has presented multiple papers utilizing this technology to solve different problems regarding 3D-data based machine learning [4, 5, 6].

1.2.1. 4D Spatio-Temporal ConvNets

Choy et al. [5] proposed using convolutional neural networks which used 3D-video as input data. These videos consisted of point-clouds with timestamps. The proposed convolution kernel type was four dimensional, using the time axis as its fourth dimension in addition to the three spatial dimensions (X,Y,Z).

When representing a 3D scene, the authors state that the scene is mostly empty space in terms of important 3D perception features. Therefore the authors choose to represent the 3D data as sparse tensors. A sparse tensor is a representation of a tensor where a specific value is removed and only the remaining data is saved. The selected data is then saved as a dense tensor and the removed values are saved as a map of regions that consisted of these values. In this way, the authors are able to represent 3D scenes as dense data-rich tensors for more efficient computations on the GPU where memory typically is a limited resource.

The authors propose the use of specializes kernels for convolution in four dimensions. The kernels can be thought of as multiple three dimensional kernels that are applied on multiple tensors at once where the number of tensors the kernels are applied to is the size of the 4D-kernel in the time axis. The best performing kernel is the hybrid kernel, which can be seen as a combination between the hypercross and hypercube kernels presented in Figure 1.1.

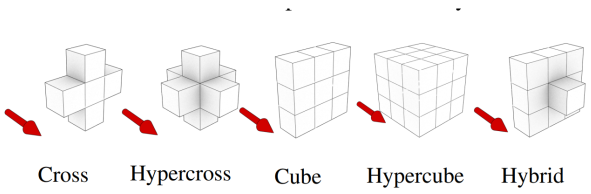

Figure 1.1.: The figure shows kernels used in 4D convolution. The red arrows indicate the time-axis and the third spatial dimension is hidden. The figure is adapted from [5].

The results indicate that 3D convolutional neural networks generally outperforms 2D networks, and that the spatio-temporal networks are more robust to noise in the 3D data.

1.2.2. Fully Convolutional Geometric Features

As a continuation of the work presented in [5], Choy et al. present a method for fast and efficient method for feature extraction from 3D data using a 3D fully convolutional neural network. Choy, Park, and Koltun state that feature extraction traditionally rely on patch extraction for features or computation of low level features to be used as input data. The authors present a solution which is able to extract features using a single fully convolutional neural network.

The work presented attempts to provide improvements in the field of 3D perception and registration tasks, as well as tracking and object detection in 3D scenes. The term registration refers to the process that is aligning multiple 3D-scans by calculating the transformation matrix that when applied to one point-cloud makes it overlap with another. In the end, the goal is one continuous scene built from the input point clouds. The proposed feature extraction method is a form of learning-based 3D-feature extractor which has increased dramatically in popularity in recent years due to increased performance and robustness.

The number of extracted features per scan is quite high at 4×10^4. As a consequence, computing all possible pairwise distances is impractical as a learning metric. The authors propose the use of hardest-contrastive and hardest-triplet as alternative learning metrics in fully convolutional feature learning. Both contrastive and triplet loss methods traditionally use randomly sampled points for triplet and contrastive pair generation. The loss is then calculated by evaluating the associated features by a distance metric, commonly euclidean distance. Contrastive pairs are pairs of points which have a large euclidean distance between them relative to other pairs. Triplets are combinations of a contrastive pair and the opposite of a contrastive pair. They consist of three points in which there are two that are close to each other and two which are far apart in terms of euclidean distance. The terms hardest-triplet and hardest-contrastive refers to the most extreme cases in the two methods described above.

The authors present results from testing multiple feature extraction methods on several data sets. These results indicate that their solution outperforms other prospective feature extraction methods by a significant amount both in speed and recall accuracy.

1.2.3. Deep Global Registration

Recently, Choy, Dong, and Koltun proposed a method for pairwise registration of point clouds [4]. The method utilizes a 6-dimensional convolutional neural network for inlier prediction for feature pairs and a weighted Procrustes algorithm [24] for pose estimation and a robust gradient-based optimizer on the 3-dimensional special euclidean group SE(3). The Procrustes algorithm is an algorithm for estimating a 6-dimensional pose. This pose accounts for both spatial transformation and rotation in 3D space. The algorithm uses two sets of points which correspond to each other and estimate the transform and rotation to be applied to one of them to make the points overlap in the best way.

Deep global registration (DGR) utilizes features extracted using Fully convolutional geometric features (FCGF) [6]. Following extraction using a pretrained FCGF-model, the features are used for training and registration. The nearest neighbour method is used to find pairs of extracted features from both scans in the feature-space (typically 32 or 16 dimensional). These pairs are the input data for a 6-dimensional convolutional neural network that handles inlier-prediction. In contrast to the multi-dimensional output of the FCGF network, this network only outputs a probability value that predicts the likelihood of the proposed pair being a match. For all pairs, the probability is saved and used in the weighted Procrustes algorithm for pose estimation as a way of prioritizing the pairs with a higher confidence level.

The Procrustes analysis returns a translation and a rotation which are checked in the Robust Registration module. Here, the proposed solution from Procrustes is subjected to additional test which checks for diverging solutions and noisy correspondence data. This module investigates if the Procrustes solution is likely to succeed, and if not it will default to a fail-safe method such as RANSAC [11] for pose estimation.

The results indicate that Deep Global Registration is a robust method of pose estimation which outperforms other state-of-the-art methods for registration of 3D data in terms of both speed and accuracy.

1.2.4. Octree-based SIMD Strategy for ICP Registration and Alignment of 3D Point Clouds

Eggert et al. [10] propose the implementation of octrees as 3D representation of point clouds. An octree is a data structure in which each node have exactly eight branches. It is known to be a highly efficient way of representing 3D data. In this paper, the authors perform 3D point cloud registration using the iterative closest point algorithm. The algorithm uses k-nearest neighbour algorithm to determine the closest points within each cell in the octree. However, the proposed method is dependent on a high resolution reference point cloud to match the lower resolution input point cloud to. The implementation of octrees is interesting, although this specific method is not capable of utilizing the GPU. Therefore, it is comparatively slower to other registration methods.

1.2.5. The MVTec 3D-AD data set for Unsupervised 3D Anomaly Detection and Localization

Bergmann et al. [3] present a method using a 3D convolutional neural network for anomaly detection in point clouds. Additionally, the authors provide a large scale data set consisting of over 4000 point clouds captured using a Zivid One+ Medium camera. The data set is inspired by real-life visual inspection problems and is based around 10 different inspection object categories.

1.3. Objectives

In this Master's thesis, the goal is to provide the methods and insight necessary for using Zivid point clouds in feature extraction and registration tasks. The objectives for the work presented is listed below.

- Define a reproducible virtual environment for use with NTNU Idun HPC
- Perform data collection for an entry level data set using Zivid point clouds for feature extraction and registration
- Provide the ability to preprocess the Zivid point clouds for further machine learning
- Attempt feature extraction from Zivid point clouds using the Fully Convolutional Geometric Features method presented in [6]
- Attempt registration of Zivid point clouds using the Deep Global Registration method presented by Choy et al. [4]

Chapter 2.

Preliminaries

2.1. Problems in 3D Computer Vision

Currently, the automotive and robotics industries are pushing the boundaries of computer vision. Additionally, the exclusive use of 3D data has become a viable solution for creating better vision systems. The inclusion of depth in images provide a more beneficial data type for tasks such as perception, object detection, pose estimation and several other computer vision tasks. All though there are advantages with 3D over 2D, there are also challenges. A common way of processing data in vision tasks is convolutional neural networks. Even though convolution adds significant advantage over fully connected neural networks, the addition of a dimension dramatically increases the required memory of the training device compared to 2D convolutional neural networks.

2.2. 3D Cameras

To capture real world 3D data, specialized devices are required. Some of the most common 3D data capture devices are LiDAR and depth cameras. LiDAR can be incredibly fast, but offers only 3D data inherently while depth cameras are commonly able to provide color images in addition to 3D point clouds.

2.2.1. LiDAR

LiDAR devices are based either on time-of-flight or wavelength shift for determining the distance from a laser emitter to the surface of interest. Point aquisition using LiDAR can be extremely fast, and the application of LiDAR in self driving cars is becoming more popular. The device emits laser beams in a specific pattern and measures the distance for each projected beam to create 3D scans. The beams are typically projected in a grid pattern to provide points in three dimensions. Additionally, LiDAR can provide two dimensional data. In this case, the beams are projected in only one plane, often in a circle pattern from the emitter.

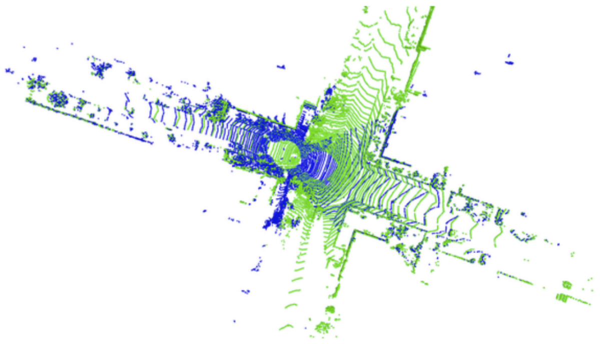

Figure 2.1.: The figure shows two overlapping point clouds from the LiDAR based KITTI data set [29]. The figure is adapted from [4].

2.2.2. Stereo Cameras

Stereo cameras refers to cameras using two sensors to capture separate images from different spatial positions and then using these two images to create 3D data. In the industry today, one of the most common tools for acquiring 3D data is the Intel® RealSense D400 series of stereo cameras. Intel® combines an RGB-sensor with two infrared sensors and one infrared emitter. Stereo vision is the classical method for achieving 3D data from the real world. It is based on calculating the distance from the camera to each point in an image based on the point's position in the images from the two sensors using the intrinsic and extrinsic parameters of the cameras. The Intel® RealSense D400 cameras use the infrared emitter as a dot projector which projects a pattern of dots onto a scene and the two infrared cameras capture images of the dots. The use of a dot pattern makes calculating the distance from the camera less computationally expensive due to the fact that it is easier to calculate correspondences between dots in a pattern than it is between pixels in RGB images.

2.2.3. Structured Light Cameras

In contrast to classical stereo cameras, structured light cameras use only one camera in combination with a projector to achieve 3D vision. The projector is responsible for projecting a set of known patterns onto the scene as can be seen in Figure 2.2. These patterns are commonly vertical lines of varying width. Projecting known patterns onto the scene from one angle and capturing an image of the scene from a different angle while different patterns are being projected creates a set of images with distorted patterns. The fact that the original patterns are known makes it possible to use intrinsic and extrinsic parameters to calculate the depth of each pixel, creating a 3D image.

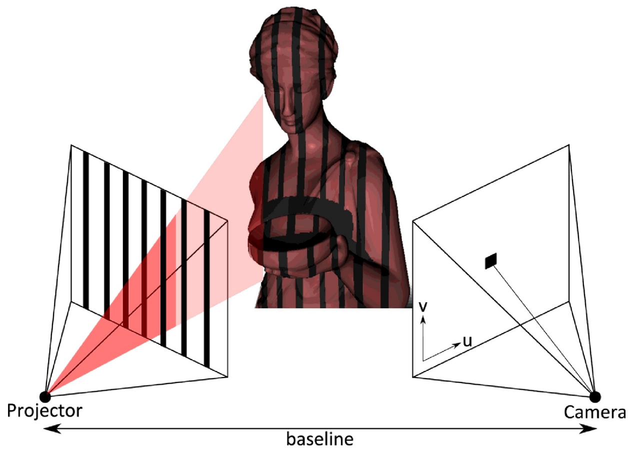

Figure 2.2.: Visualisation of the function of a structured light camera. The figure is adapted from Sarbolandi et al. [23].

Zivid is at the top of their field in structured light camera technology, and has pushed the limits of accuracy and trueness in 3D point clouds. Zivid cameras provide best-in-class resolution and per point RGB-α data in the point cloud. Compared to the Intel® RealSense D400 cameras, the Zivid cameras provide close to 10 times the per point accuracy in terms of depth [26, 8]. The Zivid camera models provide unparalleled performance which makes them highly interesting in the field of 3D-data based machine learning.

2.2.4. Advantages and Disadvantages

The Intel® RealSense D400 cameras are among the most commonly used 3D-sensors in the world. Intel offers significantly lower prices than many other providers of 3D vision solutions and has therefore been able to position themselves as the go-to solution for affordable 3D cameras. The Intel® cameras can offer faster data acquisition rates, but at the cost of resolution and accuracy compared to Zivid. The Zivid Two camera can at best offer 10 scans per second, although at the expense of image quality and point accuracy. These cameras are more suited towards stationary tasks such as bin picking where the operation is not as time sensitive.

2.3. Convolution

Convolution is the process of combining two functions, and the term often refers to the result of the process. The process is more accurately described as the changing of one function performed by another one.

2.3.1. 2D Convolution

Convolution in 2D can be visualized as an input matrix and a kernel that passes over the input moving in steps, where the size of each step is called stride. The term kernel here refers to a matrix of weights which is responsible for what the convolution process either highlights or dampens. Common kernels are Gaussian blur and edge detection kernels for image processing. The kernel weights are multiplied with each of the input matrix' corresponding values and summarised for the output matrix as can be seen in Figure 2.3.

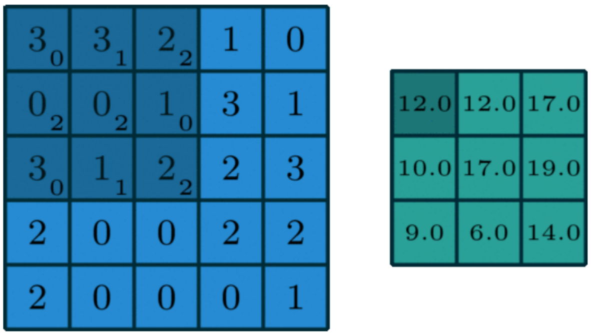

Figure 2.3.: The figure shows the input matrix (blue) and output matrix (green) as well as the kernel (denoted numbers, dark blue) for a convolution operation with stride of one. Note that the output dimensions are reduced in relation to the input matrix. The figure is adapted from [9].

As mentioned, the stride is the distance the kernel shifts for each convolution. The stride and kernel size both affect the output matrix size. Normally with a stride of one, the output matrix will be reduced in relation to the input matrix by the width and height of the kernel minus one. To counteract dimensional reduction one can pad the input matrix with values such as 0.

2.3.2. 3D Convolution

Convolution in three dimensions is only an extension of 2D convolution in terms of all parameters. The input matrix, kernel and stride all have three dimensions, although the mathematical operation is the exact same as in two dimensions. 3D convolution is commonly used for object detection and image classification.

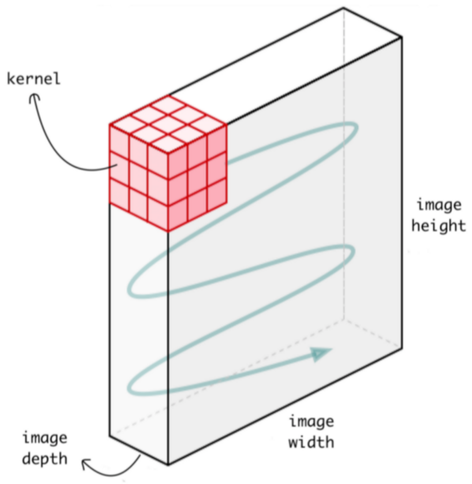

Figure 2.4.: Visualization of 3D convolution on an image where the image depth indicates the color channels red, green and blue. The kernel is shown in red. The figure is adapted from [30]

2.4. Artificial Neural Networks

Artificial neural networks (ANN) can be described as a set of neurons or nodes that together process data through interconnections and layers. ANN's can be trained to perform specific tasks, but requires data that correlates to the task at hand to do so. In machine learning there are three main principles, supervised, unsupervised and reinforcement learning [31]. These three principles also apply for neural networks. As the name implies, in supervised learning the network needs to be told what the desired output is for every input condition [14]. Reinforcement learning is not dependent on the "correct" answer for each input. However, it is dependent on certain directions. These directions are simply rewards and or punishments for choices that leads to better or worse outcomes respectively for the given problem [17]. As opposed to both supervised and reinforcement learning strategies, unsupervised learning requires no external notion of the "correct" answer for the input data. An example of unsupervised leaning is clustering and segmentation, where the networks separate data into groups only based on the input data.

2.4.1. Residual Neural Networks

A subclass of artificial neural networks are residual neural networks (ResNet) which was presented by He et al. [12]. Residual networks makes it possible to include more concatenated convolutional layers in a neural network than what was previously feasible. This is made possible by the addition of skip connections from one location in the network to another one as can be seen in Figure 2.5. It is possible to skip multiple layers as well as single layers. Providing a skip connection that skips over large parts of the network and ends closer to the end of the network may help in including higher level features in the final prediction of the network.

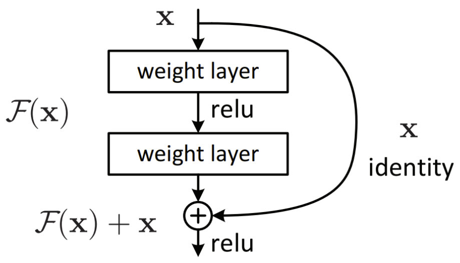
x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 2.5.: The residual building block. The figure shows a skip connection which skips two layers and is then merged with the output of the two weight layers after the data is processed. The figure is adapted from [12].

2.4.2. Convolutional Neural Networks

Convolutional neural networks are among the most common type of neural networks in machine learning today. LeCun et al. [19] proposed the use of convolution kernels as the weights in the neural network which drastically increased the maximum number of layers in neural networks. The reason for this was that previous network architectures relied heavily on fully connected layers where all nodes in one layer is connected to all nodes in the next layer with each connection having an individual weight that had to be trained and stored. The introduction of convolutional neural networks resulted in reduced memory requirements for larger and deeper neural networks.

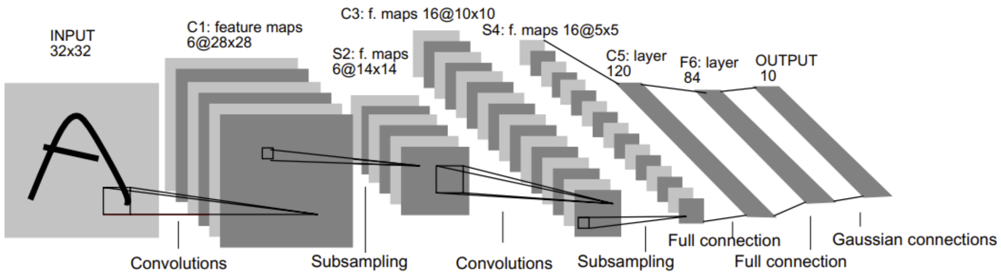

Figure 2.6.: Architecture of LeNet-5 which was presented by LeCun et al. [19].

For each convolutional layer in a neural network there is a kernel. This kernel contains the weights that are tuned during training of the network. As previously mentioned, kernels in convolution can be seen as a filter which highlights different features in the input data. During training, these features are determined independently of human interaction. The network is what determines which features are important for the final prediction.

2.5. 3D Data Representation

There are many types of 3D data representations, and in this section some of these are presented.

2.5.1. Depth Images

Depth images are images that in addition to color data have per pixel depth. Each pixel may have four data values associated with it such as the red, green and blue color channels along with a distance from the camera. From this (r, g, b, d)-data structure it is possible to calculate the coordinates of each point in relation to the camera in terms of x, y and z coordinates.

2.5.2. Point Clouds

Point clouds generally include 3D coordinates for each point in the scan. There are several ways of representing such coordinates such as a list of points using only (x, y, z) values, a list of points with pixel coordinates i.e. (px, py, x, y, z). Another example is an array where the two first dimensions of the array correspond to pixel coordinates. Further, each entry in the array is a point using the (x, y, z) format, possibly with other augmenting data such as color or transparency. There are also many point cloud file formats which can provide a challenge when gathering data for use in machine learning where standardization of input data is important.

2.5.3. Sparse Tensors

A sparse tensor is a condensed version of a dense tensor where certain information is stored in bulk in a different array. When scanning a 3D scene, the resulting point cloud is a representation of the surface that is perceived by the camera. Due to the nature of current 3D cameras both the space between the camera and the surface and the space beyond the surface is left empty in point clouds. To perform tasks such as convolution on a 3D scene, it is common to voxelize the point cloud. Voxelizing is the process of dividing up the space that encompasses the scan into volumetric pixels. When the scene is voxelized, the vast majority of the voxels are empty and are therefore very similar to each other in terms of not containing usable 3D data. Such dense tensors often contain up to 99% empty voxels. Storing and performing convolution tasks on such tensors is highly inefficient.

To make the utilization of voxelized point clouds in machine learning, we can convert the dense tensors to sparse tensors. The conversion splits the dense tensor into a dense tensor of values and a vector of indices corresponding to those values. In addition the original size of the dense tensor is saved for reconstruction purposes.

$$D = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix}, V = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}, I = \begin{bmatrix} [0,0] & [1,2] & [3,3] \end{bmatrix}, S = [4,4] \quad (2.1)$$

In Equation 2.1, the matrix D represents the dense input tensor. When converting to a sparse tensor, the V vector consists of the values in the tensor and the I vector stores the indices of each value in the original matrix. The S vector stores the original shape.

2.6. High Performance Computing Cluster

High performance clusters are networks of individual high performance computers called nodes, and there are different types of nodes for different tasks. Separate nodes are responsible for CPU compute, GPU compute and storage. These nodes are often interconnected using high speed networking. Splitting the cluster into nodes makes it easier to allocate resources between different users at the same time. Additionally, single users can allocate more than one node for computing tasks that benefit from parallel computing.

2.7. Slurm

Slurm is a solution for managing Linux clusters. It can handle queuing of jobs and allocation of resources while being easy to expand with more nodes in the future. When using Slurm for jobs on a cluster it is common to write a .slurm file, which specifies what resources are required for which amount of time together with a script that executes the job commands.

2.8. Singularity

Singularity is an open source container platform, designed for ease of use and security. Singularity is generally based on definintion files which define the building blocks and commands that are needed in order to create a specific container. Once a container is created, it is immutable. It is commonly used for installing correct versions of software to ensure compatibility. When compiled, the container provides an environment with all the specified installations, and can be shared directly as a .sif file. Alternatively, one can share the definition file and let other users compile the image themselves.

2.9. Docker

Similarly to Singularity, Docker is a solution for creating and running containers. The dockerfile is what defines a Docker container, and behaves much in the same way as the definition file in Singularity. Docker additionally support caching each instruction in the dockerfile as “layers” so that a change in one of the layers prevent the need to compile the entire container from the beginning each time. It is also possible to upload docker images to DockerHub where they are available for everyone to use.

2.10. Machine Learning Frameworks

Machine learning has gained incredible amounts of attention and has grown drastically as a field of study. As a result, multiple frameworks have been made publicly available. In these frameworks, many common machine learning tasks and functions are already implemented and allow researchers and computer scientists to hit the ground running without having to start from scratch. Common features include, convolution in neural networks, many different activation functions, multiple optimizers and tensor operations. Among the more popular frameworks are Jax, TensorFlow, Keras and PyTorch.

2.10.1. PyTorch

PyTorch is one of the most popular machine learning frameworks. It is open source and primarily written in C/C++ and Python for use in Python. It is based on the older framework Torch which is written in a fast scripting language called Lua created for scientific computing and machine learning. PyTorch includes support for GPU accelerated parallel programming using the Nvidia CUDA API. This allows for faster training of neural networks, but requires a CUDA capable Nvidia GPU.

Chapter 3.

Methods

In this chapter, a complete walk-through for training both FCGF and DGR models on Idun HPC are presented. The chapter also presents the necessary changes made to both libraries in order to perform training. In addition, both data set collection and preprocessing will be presented here.

3.1. Creating an Environment for Using the Minkowski Engine on Idun HPC

In contrast to the method presented in [2], Docker is used for creating the containerized environment as opposed to Singularity. However, Singularity will still be used to run the container on Idun HPC [27].

3.1.1. Docker Installation

The Docker Desktop application can be installed following the instructions on [13]. Here, there are guides available for Linux and Mac as well. When the installation is complete, start the Docker Desktop application.

To verify the installation, run the command below in a terminal.

```
docker version
```

3.1.2. The Dockerfile

The Dockerfile is the recipe that defines the Docker image. Here, the type and version of operating system is specified and also what is to be installed onto that operating system. Creating a Dockerfile can be done in a text editor or in an integrated development environment such as Visual Studio Code with a docker extension installed.

```
ARG VERSION="20.04"
FROM ubuntu:${VERSION}
```

Here, the `ARG` keyword defines an argument to be used later in the file. This is useful if there are several references to the same version or other arguments in the Dockerfile. The `FROM` keyword specifies what base image to build the image on top of. Here, the image will be built upon an Ubuntu image from DockerHub.

The Dockerfile also supports running commands on the base opertaing system (OS) for installing other dependencies. Any command valid for the OS can be issued using the `RUN` keyword.

```
RUN apt-get update
RUN apt-get install git
```

`RUN` commands can only be issued after pulling a base image using the `FROM` keyword. In the code shown above, the Docker compiler will update the list of available packages with `RUN apt-get update`. Further, it will install the latest version of Git available for the operating system using `RUN apt-get install git`.

In the Dockerfile, it is also possible to set environment variables. This is useful when a specific command relies on the environment variables to execute correctly or yield the desired result. In the code below, the environment variable `MAX_JOBS` is being set prior to a command execution.

```
ENV MAX_JOBS=1
```

3.1.3. Building a Docker Image

When building a Docker image it is important to note that the system needs to be able to store the image in its entirety. If one is installing large libraries on top of an operating system, the size of the final image will quickly increase. Prior to building an image, it is recommended to log in to DockerHub, either from the

Docker Desktop program or the command line using the `docker login` command. An example of how to build a Docker image from a Dockerfile is shown below.

```
docker build -t <USERNAME>/<IMAGE_NAME>:<TAG> <LOCATION>
```

In the command shown above, the `-t` flag tells Docker that we wish to tag this image. Using the username for DockerHub in place of `<USERNAME>` and naming the image `<IMAGE_NAME>`, the image is more easily uploaded to DockerHub after completing the build. The `<TAG>` is a way of handling versions of an image. Each unique tag is a different image and accessible as unique versions in Docker. When the image is built, upload the image to DockerHub using the command below.

```
docker push <USERNAME>/<IMAGE_NAME>:<TAG>
```

3.1.4. Reproducible Environment for Minkowski Engine

This section will present the Dockerfile used to build the environment used for performing preprocessing and machine learning tasks on Idun HPC. The Dockerfile presented in this section can be found in Appendix A.

```
ARG PYTORCH="1.9.0"
ARG CUDA="11.1"
ARG CUDNN="8"

FROM pytorch/pytorch:${PYTORCH}-cuda${CUDA}-cudnn${CUDNN}-devel
```

In the code above, the three arguments describe a specific version of PyTorch, CUDA and cuDNN (CUDA Deep Nerual Network library). These three arguments are used to pull the correct version of a Linux operating system where these three dependencies are installed previously. This image is pulled from a PyTorch repository on DockerHub [22].

```
ENV TORCH_CUDA_ARCH_LIST="8.0+PTX"
```

This line sets the environment variable `TORCH_CUDA_ARCH_LIST` to `"8.0+PTX"` where “8.0” determines the cuda architecture we wish to compile for when using the Nvidia Cuda Compiler (nvcc). There are several different CUDA capable GPU’s installed in Idun HPC, but this image only builds for the latest A100 cards. To expand the compute compatability of the image, one can add other numbers to the environment variable i.e. `TORCH_CUDA_ARCH_LIST="5.2 6.0 6.1 7.0 7.5 8.0 8.6+PTX"`. However, the nvcc will compile a unique build for each architecture in the list which will increase both build time and image size.

```
ENV TORCH_NVCC_FLAGS="-Xfatbin -compress-all"
```

The environment variable `TORCH_NVCC_FLAGS` is responsible for how the nvcc will handle .cu files when compiling.

```
RUN apt-get update
RUN apt-get install -y git ninja-build cmake build-essential \
    libopenblas-dev xterm xauth openssh-server tmux wget \
    mate-desktop-environment-core

RUN apt-get clean
RUN rm -rf /var/lib/apt/lists/*
```

In the code shown above, the Docker compiler will install dependencies directly on the operating system. `RUN apt-get clean` clears the install cache and avoids storing the cache in the docker image. The `rm -rf /var/lib/apt/lists/*` command deletes the stored information about packages that have been installed using apt.

```
RUN pip install matplotlib pillow numpy scipy cython \
                scikit-image sklearn opencv-python open3d \
                netCDF4 easydict h5py tensorboardX
```

This command installs a number of libraries for Python. NumPy and Open3D are requirements for using the Minkowski Engine library. NetCDF4 is used for reading the Zivid point cloud files in preprocessing.

```
ENV MAX_JOBS=1
RUN pip install -U git+https://github.com/NVIDIA/MinkowskiEngine \
                            -v --no-deps \
                            --install-option="--force_cuda" \
                            --install-option="--blas=openblas"
```

Finally, the Minkowski Engine library is installed. The line `ENV MAX_JOBS=1` determines the max number of parallel jobs that the CUDA compiler can perform when building the library back-end. The final command is the recommended way of installing the Minkowski Engine library which can be found on GitHub [21].

The Docker image is then built and pushed to DockerHub using the commands shown below. The command is issued in a command prompt after navigating to the Dockerfile location.

```
docker build -t jonassaa/minkowskiengine:latest .
docker push jonassaa/minkowskiengine:latest .
```

3.1.5. Pulling a Docker Image on Idun HPC using Singularity

For login instructions for Idun HPC, see [20] or follow the instructions provided in [2]. The unpublished specialization project [2] can be found in Appendix G. When logged in to Idun, execute the following command to create a Singularity image from a docker image on DockerHub.

```
singularity pull docker://<USERNAME>/<REPOSITORY>:<TAG>
```

This command will download an image from any public user on Dockerhub specified by `<USERNAME>`. The `<REPOSITORY>:<TAG>` specifies which image and version of that image to download. If the `<TAG>` is omitted, singularity will pull the image with the "latest" tag. To download the image used in this thesis, execute the command below.

```
singularity pull docker://jonassaa/minkowskiengine:latest
```

The command above will download the docker image from subsection 3.1.4 and convert it to a .sif file. This .sif file can be used to run the environment on Idun using Singularity.

3.2. Creating a Data Set of Zivid Point Clouds

This section presents a method of capturing and preprocessing Zivid point clouds for use with FCGF and DGR.

3.2.1. Capturing Zivid Point Clouds

To create a data set using Zivid point clouds, multiple 3D scans had to be captured. To capture the 3D scans, a computer with Zivid Studio installed was connected to a Zivid One camera. The specific camera was an older model and required an older version of the Zivid software (1.8.3). The setup is shown in Figure 3.1.

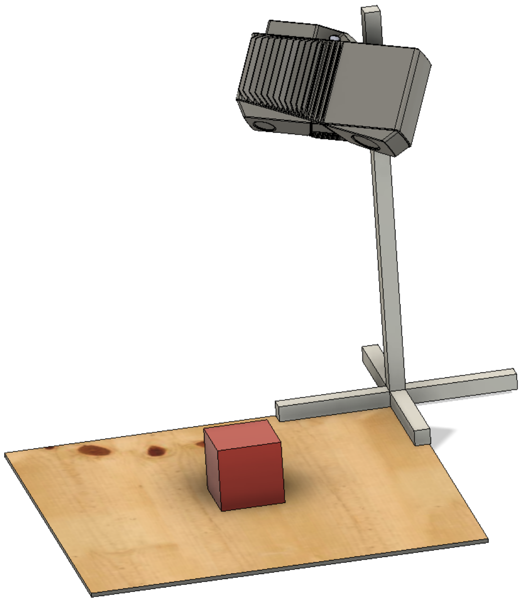

Figure 3.1.: Illustration showing the experimental setup for capturing Zivid point clouds. The red cube shows the approximate position of the objects scanned.

The resulting files was organized into the structure shown below.

- Data Set Folder
  - Object 1
    - Scan 1
    - Scan 2
    - Scan 3
  - Object 2
  - Object 3

For each object, 10 scans were captured from different positions around the object. All scans were captured using the same fixed vertical camera position as well as the same capture angle relative to the table.

3.2.2. Converting Zivid Point Cloud to NumPy Array

A program was created for preprocessing the Zivid point clouds. The code for this program can be found in Appendix B and on GitHub [16]. To read the Zivid point cloud files, a library called NetCDF4 can be used. The Zivid point clouds are saved as netCDF files inherently.

```
def loadPointCloudFromZivid(pcd):
    zividPointCloud = netCDF4.Dataset(pcd, 'a', format = "NETCDF4")
    return np.reshape(np.asarray(zividPointCloud["data/pointcloud"]),
                      (1920*1200,3))
```

Here, we define a function that reads a Zivid point cloud file (`pcd`) with a .zdf extension and converts it to a numpy array with the dimensions (1920*1200,3). This array is structured as a list of 3D vectors, where the vector describes x, y and z coordinates of each pixel in 3D space. This operation also converts the Zivid point clouds to unstructured point clouds.

3.2.3. Matching 3DMatch Data Structure

In the repositories FCGF and DGR by Chris Choy, one of the data sets used to test the performance of the two methods is called 3DMatch [32]. This data set consists of numerous scans of rooms and objects in a RGB-D video format and is publicly available. Chris Choy provides a version of this data set that is structured to work with the data-loaders in both DGR and FCGF. To match the structure of the provided data set using Zivid point clouds, the data set folder described in subsection 3.2.1 is processed using the Preprocess.py program provided in the Zivid_DGAP repository [16]. The program enumerates through the file structure of the data set folder and creates numpy arrays for each scan in the data set. Each array is named after the parent object on the format `<Name of object>_<Scan number for that object>.npz`. Here, the `.npz` extension shows that the array is saved as a compressed numpy array. For each object, unique pairs of scans are generated and the resulting pairs are listed in a text file as shown below.

```
Object_0.npz Object_1.npz
Object_0.npz Object_2.npz
Object_0.npz Object_3.npz
Object_0.npz Object_4.npz
Object_0.npz Object_5.npz
```

As can be seen in the list above, the `Object` has been scanned multiple times as denoted by the numbers. Each line represent a unique pair of two scans of the object.

Additionally the data set is split into a training and validation subset. A text file is created for each subset, listing the objects that are part of it. Finally, the entire data set is compressed into a single archive with a .zip extension. How to use the preprocessor program is described in [16].

3.3. Data Loader for FCGF and DGR

In both FCGF and DGR, the data loader is not equipped to handle Not a Number (NaN) values. In Zivid point clouds there are several points that are saved with NaN coordinates. In both repositories, the addition of the following code excludes points that contain NaN values.

```python
xyz0 = data0["pcd"][~np.isnan(data0["pcd"]).any(axis=1), :]
```

The code above creates a boolean array depending on what points have any NaN values. The boolean array is then used as a filter to only include valid points. The data loaders for the Zivid data set in FCGF and DGR can be found in the code repositories in Appendix G.

3.4. Training

This section describes the steps for executing training using Singularity on Idun HPC. Additionally, the necessary preparations will also be presented.

3.4.1. Prerequisites

On Idun HPC, create an empty directory and clone the FCGF and DGR repositories using the commands below.

```
git clone https://github.com/jonassaa/FCGF.git
git clone https://github.com/jonassaa/DeepGlobalRegistration.git
```

Transfer the processed data set files (Appendix G) to Idun HPC using the method described in [2]. For both FCGF and DGR, the configuration file has been edited to use the Zivid data set by default.

3.4.2. Training FCGF Model

For training the FCGF model, we schedule a job in the slurm queue. The slurm script defines and describes what resources are required.

```
#!/bin/bash
#SBATCH --job-name=FCGF_train
#SBATCH --mail-type=ALL
#SBATCH --mail-user=jonassaa@stud.ntnu.no
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=12
#SBATCH --mem=16000
#SBATCH --time=96:00:00
#SBATCH --output=/cluster/home/jonassaa/jobs/FCGF/outputs/FCGF_train.log
#SBATCH --partition=GPUQ
#SBATCH --gres=gpu:A100m80:1

date;hostname;pwd
singularity exec --nv minkowskiengine.sif ./train_FCGF.sh
date
```

In the commands shown above, the lines 2-12 describe the behaviour of slurm. Specific resources and time limit, as well as email notifications and log output file is determined here. This script requests one Nvidia A100 card with 80 GB of VRAM for the job. The lines 14-16 describes what commands are to be executed when the job is running. Line 14 writes the date and time, hostname and the current directory to the log file. Line 15 is responsible for executing the training script in the Singularity container. The `--nv` flag ensures that the GPU is available within the container. Line 16 prints the date and time before the job is finished.

```bash
#!/bin/bash
export OMP_NUM_THREADS=12
cd ./FCGF
nvidia-smi

python ./train.py \
                    --batch_size=2 \
                    --voxel_size=0.001 \
                    --max_epoch=100 \
                    --train_num_thread=12 \
                    --val_num_thread=12 \
                    --hit_ratio_thresh=0.0425\
                    --conv1_kernel_size=7
```

The code above is the shell script that executes the `train.py` program within the container. Batch size is the number of point clouds processed in parallel. The voxel size is defined in meters, 0.001 corresponds to 1 mm. The `hit_ratio_thresh` is a threshold for determining if a pair of features correspond or not. A lower value describes better correspondence. Finally, the flag `--conv1_kernel_size` determines the convolution kernel size in the first layer of the neural network.

To schedule the training job, execute the command below.

```
sbatch <Slurm script file>
```

3.4.3. Training DGR Model

Training the DGR model is performed in a similar manner to the FCGF model. The slurm script for training the DGR model is nearly identical to the FCGF script. It utilizes the same Singularity container and command, but executes a different shell script as shown below.

```
singularity exec --nv minkowskiengine.sif
↪  ./trainDGR_singularity_script.sh
```

```
1  #! /bin/bash
2  export OMP_NUM_THREADS=12
3  cd /cluster/home/jonassaa/jobs/DeepGlobalRegistration
4  python ./train.py \
5    --dataset ZividPairDataset \
6    --zivid_dataset_dir "/cluster/home/jonassaa/jobs/ZividOne_v2/" \
7    --feat_model ResUNetBN2C \
8    --feat_model_n_out=32 \
9    --feat_conv1_kernel_size=7 \
10   --lr 1e-1 \
11   --batch_size=2 \
12   --max_epoch=100 \
13   --voxel_size 0.001 \
14   --hit_ratio_thresh=0.0425 \
15   --weights "./modelsFCGF/3Dmatch_25mm_32dim.pth"
```

Training the DGR model is dependent upon an existing FCGF model for feature extraction. Here, the `--weights` flag is used to tell the program where to find the pretrained FCGF model. It is important to match `--feat_model_n_out`, `--feat_conv1_kernel_size` and `--feat_model` to the settings that was used when training the FCGF model to avoid incompatibility errors. Line 2 is responsible for setting the number of threads used in parallel when loading point clouds. If this environment variable is not set, the Minkowski engine library will throw a warning and set it to a default value of 16.

Chapter 4.

Results and Discussion

This Master's thesis proposes the use of high resolution Zivid point clouds in feature extraction and registration tasks, using the Minkowski engine library. In this chapter, the method described in chapter 3 as well as the results are subject to analysis and discussion. To our knowledge, attempting to train the feature extraction and registration models from FCGF and DGR on Zivid point clouds has not been performed previously, therefore relevant literature is scarce.

4.1. Virtual Environment for using the Minkowski Engine Library on Idun HPC

Idun HPC does not support the use of docker images to run containers, it only supports the use of Singularity containers. However, when attempting to change the `TORCH_CUDA_ARCH_LIST` to include the 8.0 architecture, the build method presented in [2] failed. The Singularity compiler preferred to compile the CUDA code in the Minkowski engine library in parallel regardless of the instructions provided. This lead to an error being thrown and the compile process failing. The assumption is that parts of the CUDA code was being compiled before its dependencies was finished compiling.

Singularity does however support directly converting a docker image from DockerHub. Building upon the Dockerfile from the Minkowski engine repository, a Dockerfile was created that included the necessary python libraries and the correct Cuda architecture. Building this image with Docker was a success. The resulting image can be found on DockerHub [15], and the Dockerfile can be found in Appendix A. The transition to Docker also provides a more available image. As mentioned, the image can be downloaded and converted directly on Idun HPC which eliminates the need for users to compile the image themselves and upload it to Idun.

The use of Docker as opposed to Singularity also makes it faster to alter the contents of the container. The different commands in the Dockerfile are cached as layers by default when compiling and makes recompiling a Docker image significantly faster than recompiling a Singularity container.

4.2. Zivid Point Cloud Data Set

Using the method described in subsection 3.2.1, 219 scans divided among 22 different scenes were captured. Compared to the 3DMatch data set consisting of 2189 scans presented in [32], our data set is comparatively small. When capturing the point clouds for the Zivid data set, each scan took between 30-90 seconds. Due to limited time with access to the Zivid One camera, expanding the data set further was not possible. The aim of this data set as opposed to the 3DMatch data set was to provide scans that were of higher quality as well as more relevant to bin picking in the robotics industry.

As mentioned, one of the aims of creating this data set was to provide a more relevant data set for bin picking. An example of the resulting scans from the Zivid One camera is shown in Figure 4.1.

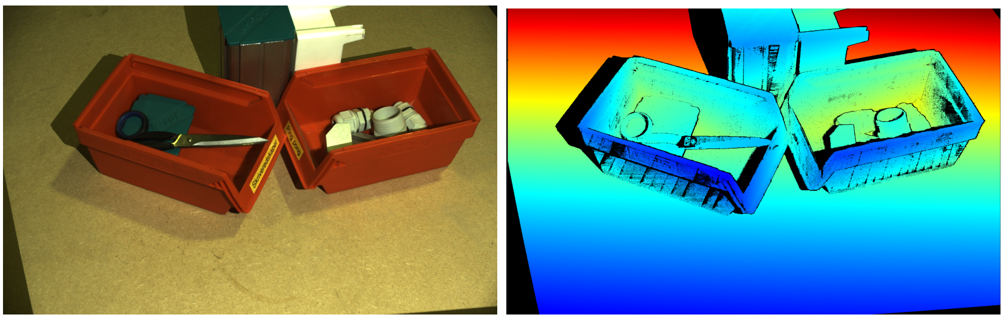

Figure 4.1.: Left: RGB image of bins with plastic fittings and a pair of scissors. Right: Depth map of the same scene as to the left.

One of the reasons each scan took so long was the fact that the camera occasionally produced unusable scans. These were scans where the computation of depth seemed to have diverged and resulting in a scan where the 3D data did not resemble the object in any way. In hindsight, it was brought to our attention that the specific camera that was used to capture the data set had been damaged and was therefore prone to outputting scrambled data.

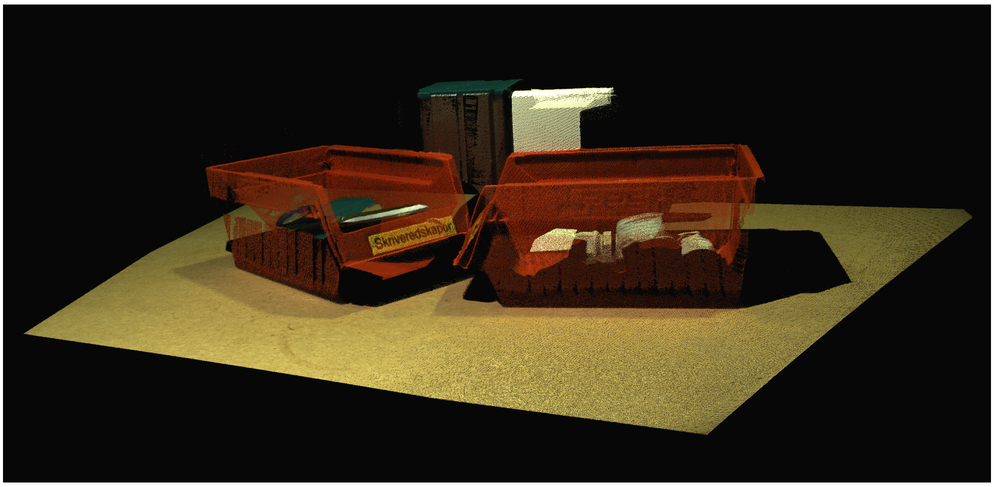

Figure 4.2.: Visualization of a 3D point cloud in the Zivid data set. This point cloud is the result of the same scan as in Figure 4.1

Zivid point clouds have higher resolution than competitive vision systems while including color, normal and position data. As a result, these point clouds are significantly larger than the scans produced by other lower resolution cameras. In turn, the raw data captured is larger in terms of storage space. Interestingly, the processed Zivid point clouds are less data-dense than the original files. In the compressed numpy arrays, only 3D data remain. However, the processed data set is 6.06 GB compared to 4.75 GB unprocessed as can be seen in Table 4.1. This raises the question of what is more efficient, pre-processing the raw data beforehand, or implementing the processing in the dataloader directly.

	3DMatch Data Set	**Zivid Data Set**
Number of scans	2189	219
Raw data set size	Not applicable	4.75 GB
Processed data set size	8.12 GB	6.06 GB
Avg. size per scan	3.72	27.64 MB

Table 4.1.: Comparing the Zivid data set to the 3DMatch data set.

The Zivid SDK [25] supports sub-sampling the point clouds for reduced resolution and file size. Due to unknown errors, we were not able to install the Zivid SDK onto the Docker image used on Idun HPC. This was the reason for using the NetCDF4 library for reading the point clouds. Additionally, the implementation of the SDK would allow sub-sampling in the data set generation process and in turn allow for a smaller overall data set as well as faster training times.

It would be interesting to investigate the impact of sub-sampling from Zivid point clouds in future studies. It is assumed that sub-sampling will yield generally less noisy data, which may be better for training. The ability for subsampling using the Zivid SDK is built into the Zivid_DGAP repository, but was not utilized in this project. It is possible to perform subsampling of point clouds using other libraries such as Open3D or by using the methods proposed in [1]. However, these methods were not utilized in this project due to time constraints.

As the Zivid SDK include ROS functionality, it would be interesting to use a robot for data set capture. Programming predetermined or random positions and then capturing multiple scans of objects would be both easier and less time consuming. This would allow for more known data in the data set. Overlap ratio is described in Zeng et al. [32] as the ratio of each scan overlapping the next. The 3DMatch data set even include subsets of pairs with an overlap ratio of least 0.3 and 0.7 to ensure registration is possible. Knowing the exact positions of the camera for each scan in relation to the object would allow us to create overlap ratio subsets in the Zivid data set for better training.

The Zivid_DGAP repository used for processing data on Idun HPC as well as the raw and processed 3D scans can be found in Appendix G.

4.3. Training FCGF Model

Training the FCGF model was performed as described in subsection 3.4.2. The models provided by Chris Choy on the FCGF Github page has a final feature match ratio (FMR) of 0.9578. This indicates that over 95% of the extracted features in one scan could be found in the paired scan on average. However, this number is dependent upon many, factors such as voxel size and the threshold for what is a valid feature match. In our testing, we were able to achieve an FMR of 0.9222. Even though this number is close to the best feature match ratio obtained by Choy, Park, and Koltun, this metric does not guarantee a well performing model. The feature match ratio is the default validation metric, and in our testing we found that it was directly related to the hit ratio threshold which is a tuneable parameter in the configuration. Tuning this variable did not seem to produce a better model in terms of either the translational or rotational error.

Unfortunately, the training with the Zivid data set did not yield a good model. There seems to be no significant improvement in performance metrics in correlation to epochs. The validation metrics fluctuate sporadically and no clear trend can be seen in terms of increase or decrease over time. Many attempts have produced the best validation score early in terms of epochs and show no sign of increasing despite being run for 100 epochs. One would expect some form of overfitting as described by Lawrence et al. [18], as the data set is quite limited in terms of scans compared to the 3DMatch data set. Interestingly, this seems to not be the case. There is no apparent increase or decrease in performance metrics, which leads to the assumption of hidden errors or improper hyperparameter tuning.

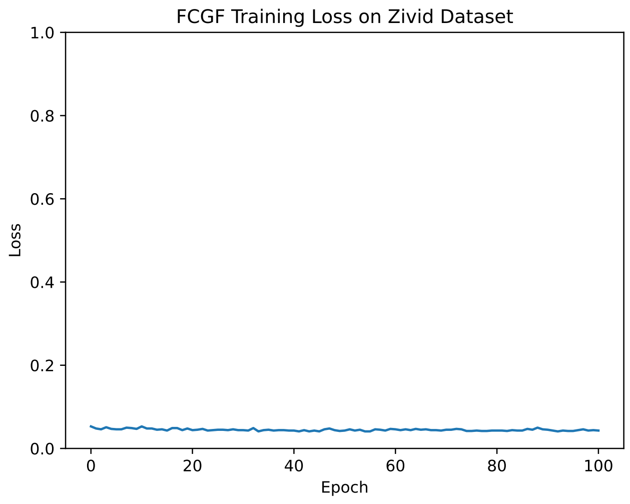

Figure 4.3.: The graph shows the final loss as a function of epochs when training the FCGF model on the Zivid data set.

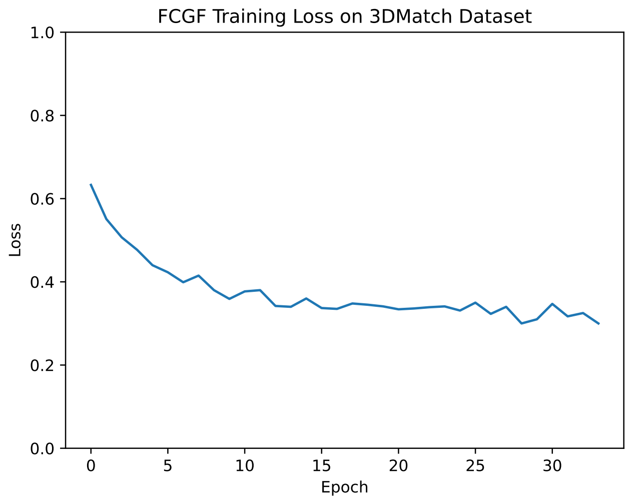

Figure 4.4.: The graph shows the final loss as a function of epochs when training the FCGF model on the 3DMatch data set.

Training an FCGF model on the Zivid data set yielded substantially different results than what was expected in terms of loss. Even though the parameters are different in the two cases, the results are assumed to be comparable. As can be seen in Figure 4.3 and Figure 4.4, the loss is lower by an order of magnitude when training on the Zivid data set compared to the 3DMatch data set. The 3DMatch training loss resembles what is to be expected as shown in Figure 4.5.

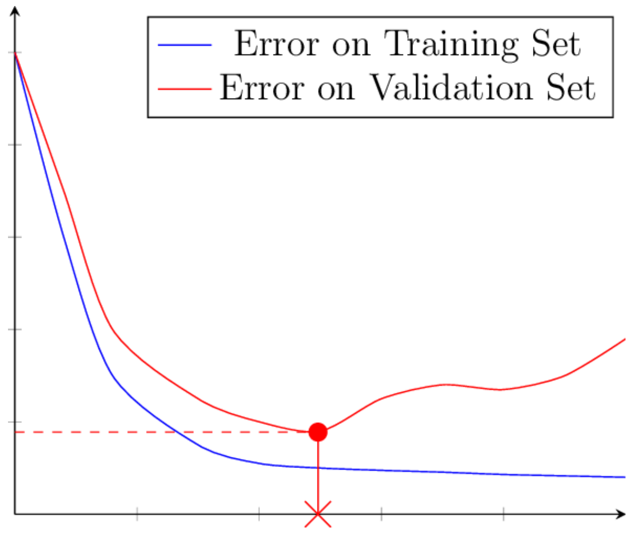

Figure 4.5.: Visualization of expected loss behaviour when training a neural network. The figure is adapted from Suárez-Paniagua [28].

Due to the generally low loss values when training on the Zivid data set compared to the 3DMatch data set, we assume that the loss function for FCGF is incompatible with spatially smaller scenes such as in the Zivid data set. The 3DMatch data set consists of scans from 3D videos of rooms and hallways where the point clouds approximately measure 2x2x2 meters. For further studies using Zivid point clouds with FCGF, it would be interesting to investigate other means of calculating loss.

When testing the training process in the FCGF library, many different combinations of `voxel_size`, `conv1_kernel_size` and `hit_ratio_thresh` were tested. Unfortunately, this did not lead to better model performance. This strengthens our assumption that training problems have been related to loss calculation.

In hindsight, the method for capturing Zivid point clouds may have differed too far from the 3DMatch data set to which it is compared in testing. Theoretically, capturing the same scenes as the ones in 3DMatch with a Zivid camera and setting the voxel sizes equal for the two data sets should yield very similar results. It would be ineteresting to investigate if using a Zivid camera to create a data set that resembled 3DMatch more closely would impact the performance of the model. This would be useful for determining the reason the FCGF training does not perform well with the Zivid data set.

The FCGF repository used for training on Idun HPC can be found in Appendix G.

4.4. Training DGR Model

Training a DGR model is dependent upon having an FCGF model. The Deep Global Registration library utilizes an FCGF network for feature extraction prior to inlier prediction and registration. Performing registration is in turn dependent on good feature extraction, which requires a well performing feature extraction model. As described in section 4.3, we were not able to produce a well performing FCGF model based on the Zivid data set. Multiple trained FCGF models were tested when training the DGR model. All FCGF models tested, resulted in an infinite loss error. Moreover, it was also investigated whether using an FCGF model from the FCGF Model Zoo [7] would yield different results. This approach also resulted in infinite loss errors as can be seen in the log provided in Appendix G. It is assumed that this was due to the features in the 3DMatch data set not being similar to the ones in the Zivid data set.

The DGR registration program was modified to accommodate for the NaN values in the Zivid point cloud tensors. The registration module takes two tensors and a set of weights as input data as shown below.

```
def weighted_procrustes(X, Y, w, eps):
```

The initial problem was that the tensors X and Y consistently had a size mismatch due to the NaN values being removed independently in the dataloader. To ensure matching sizes, the following code was added to the weighted Procrustes function.

```
# Cleaning up input data for nan, removing the same rows from all
↪  tensors
newX = X[~torch.any(X.isnan(),dim=1)]
newY = Y[~torch.any(X.isnan(),dim=1)]
new_w = w[~torch.any(X.isnan(),dim=1)]

newX = newX[~torch.any(newY.isnan(),dim=1)]
new_w = new_w[~torch.any(newY.isnan(),dim=1)]
```

```
newY = newY[~torch.any(newY.isnan(),dim=1)]

X = newX
Y = newY
w = new_w
```

In the code shown above, the program removes the points from all arrays that correspond to NaN values in both input tensors X and Y as well as the weight tensor w. These point clouds are unstructured, meaning they only contain a list of points as opposed to points with additional pixel coordinates. Removing points from the point clouds is assumed to not impact the registration in a significant manner. This is due to the similarity between removing points and parts of the geometry being obstructed when scanning. Further, the weighted Procrustes function had issues with performing singular value decomposition (SVD). SVD generally fails when the input matrix is singular, meaning its determinant is equal to zero. The solution was the addition of the code shown below as opposed to `U, D, V = torch.svd(Sxy)`.

```
try:
    U, D, V = torch.svd(Sxy)
  except:
    # torch.svd may have convergence issues for GPU and CPU.
    U, D, V = torch.svd(Sxy +
     ↪  1e-2*Sxy.mean()*torch.rand(Sxy.shape[0],3))
```

As shown above, the program adds random noise to the `Sxy` tensor if regular SVD fails and recalculates SVD afterwards. The random noise is scaled to be maximum 1% of the mean value in the tensor. These changes enabled the registration program to run. However, the exact reason for failing SVD is unknown. We assume that the `Sxy` tensor which is calculated with both the extracted features and the weights from the DGR model somehow becomes singular. This could happen in the case that all weights from the DGR inlier predictions being zero. This is most likely due to bad feature extraction in FCGF which directly affects the `Sxy` tensor and the DGR weights.

The DGR repository used for training on Idun HPC can be found in Appendix G.

Chapter 5.

Conclusion and Future Work

5.1. Conclusion

The work presented in this Master's thesis enables the use of high resolution Zivid point clouds for machine learning using the Minkowski Engine library on the NTNU Idun HPC. Further, a data set of Zivid point clouds was created to attempt feature extraction and registration using the FCGF and DGR libraries.

The reproducible environment for using the Minkowski Engine library presented in this thesis is able to utilize the latest Nvidia CUDA architechture for training neural networks in PyTorch. All training in this project was performed using Nvidia A100 GPU's. Further, the environment contains the necessary requirements to be able to perform the preprocessing of the Zivid point clouds to create the Zivid data set used for training the FCGF and DGR models.

The Zivid data set was created using a time consuming by-hand approach and the result is closer to a proof of concept rather than a data set to be used as is. This is due to the limited size and diversity in the data set. Further, the large size of each scan in the data set makes it impractical to increase the number of scans by a substantial amount.

The creation of the Zivid data set enabled the training of FCGF and subsequently DGR models using Zivid point clouds. The results indicated that the FCGF library was having problems calculating the loss when using Zivid point clouds as opposed to the point clouds in the 3DMatch data set. This problem is assumed to be the reason that the FCGF model was not able to properly extract features. Training the DGR model also resulted in errors. These errors was assumed to be related to improper feature extraction using the FCGF model. There are many parameters relating to loss and error calculation in the FCGF configuration. Tuning these hyperparameters to suit the Zivid data set provided in this thesis could improve the loss calculation and in turn the FCGF model performance.

5.2. Future Work

Based on the results and insights gained from working on this thesis, the following topics are proposed for future study.

- Enhanced data set collection using a robot mounted Zivid camera with ROS integration.
- Utilize Zivid SDK features or other methods to investigate the impact of sub-sampling.
- Create a data set of Zivid point clouds resembling the 3DMatch data set to more closely compare the performance of high resolution point clouds in feature extraction and registration tasks.
- Investigate alternative methods for calculating loss in FCGF or tune hyper-parameters to suit the Zivid data set.

References

[1] *3D Point Cloud processing tutorial by F. Poux | Towards Data Science.* URL: https://towardsdatascience.com/how-to-automate-lidar-point-cloud-processing-with-python-a027454a536c.

[2] Jonas S. Aasberg. *Machine Learning using 3D Data on a High Performance Computing Cluster (Unpublished specialization project).* 2021.

[3] Paul Bergmann, Xin Jin, David Sattlegger, and Carsten Steger. "The MVTec 3D-AD Dataset for Unsupervised 3D Anomaly Detection and Localization". In: (). URL: https://orcid.org/0000-0002-4458-3573.

[4] Christopher Choy, Wei Dong, and Vladlen Koltun. "Deep Global Registration". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 2020.

[5] Christopher Choy, JunYoung Gwak, and Silvio Savarese. "4d spatio-temporal convnets: Minkowski convolutional neural networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 2019, pp. 3075–3084.

[6] Christopher Choy, Jaesik Park, and Vladlen Koltun. "Fully Convolutional Geometric Features". In: *Proceedings of the IEEE International Conference on Computer Vision.* 2019, pp. 8958–8966.

[7] *chrischoy/FCGF: Fully Convolutional Geometric Features: Fast and accurate 3D features for registration and correspondence.* URL: https://github.com/chrischoy/FCGF.

[8] *Depth Camera D435 – Intel® RealSense™ Depth and Tracking Cameras.* URL: https://www.intelrealsense.com/depth-camera-d435/.

[9] Vincent Dumoulin, Francesco Visin, and George E P Box. *A guide to convolution arithmetic for deep learning.* Tech. rep. 2018. URL: http://ethanschoonover.com/solarized.

[10] D. Eggert and S. Dalyot. "OCTREE-BASED SIMD STRATEGY for ICP REGISTRATION and ALIGNMENT of 3D POINT CLOUDS". In: *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences* 1 (July 2012), pp. 105–110. ISSN: 21949050. DOI: 10.5194/ISPRSANNALS-I-3-105-2012.

[11] Martin A. Fischler and Robert C. Bolles. "Random sample consensus". In: *Communications of the ACM* 24.6 (June 1981), pp. 381–395. ISSN: 15577317. DOI: 10.1145/358669.358692. URL: https://dl.acm.org/doi/abs/10.1145/358669.358692.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition". In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* 2016-December (Dec. 2015), pp. 770–778. ISSN: 10636919. DOI: 10.1109/CVPR.2016.90. URL: https://arxiv.org/abs/1512.03385v1.

[13] *Install Docker Desktop on Windows | Docker Documentation*. URL: https://docs.docker.com/desktop/windows/install/.

[14] Tammy Jiang, Jaimie L Gradus, and Anthony J Rosellini. *Supervised Machine Learning: A Brief Primer*. Tech. rep. 2020. URL: www.elsevier.com/locate/bt.

[15] *jonassaa/minkowskiengine Tags | Docker Hub*. URL: https://hub.docker.com/r/jonassaa/minkowskiengine/tags.

[16] *jonassaa/Zivid_DGAP: Dataset generator and data preprocessor for zivid pointclouds for use with FCGF and DGR (Chris Choy, NVIDIA)*. URL: https://github.com/jonassaa/Zivid_DGAP.

[17] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. "Reinforcement Learning: A Survey". In: *Journal of Artificial Intelligence Research* 4 (May 1996), pp. 237–285. ISSN: 1076-9757. DOI: 10.1613/JAIR.301. URL: https://www.jair.org/index.php/jair/article/view/10166.

[18] Steve Lawrence, C Lee Giles, and Ah Chung Tsoi. "Lessons in Neural Network Training: Overfitting May be Harder than Expected". In: (1997), pp. 540–545. URL: www.aaai.org.

[19] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. "Object Recognition with Gradient-Based Learning". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 1681 (1999), pp. 319–345. ISSN: 16113349. DOI: 10.1007/3-540-46805-6{_}19. URL: https://link.springer.com/chapter/10.1007/3-540-46805-6_19.

[20] *Login – High Performance Computing Group*. URL: https://www.hpc.ntnu.no/idun/getting-started-on-idun/login/.

[21] *NVIDIA/MinkowskiEngine: Minkowski Engine is an auto-diff neural network library for high-dimensional sparse tensors*. URL: https://github.com/NVIDIA/MinkowskiEngine.

[22] *pytorch/pytorch Tags | Docker Hub*. URL: https://hub.docker.com/r/pytorch/pytorch/tags.

[23] Hamed Sarbolandi, Damien Lefloch, and Andreas Kolb. "Kinect range sensing: Structured-light versus Time-of-Flight Kinect". In: *Computer Vision and Image Understanding* 139 (Oct. 2015), pp. 1–20. ISSN: 1077-3142. DOI: 10.1016/J.CVIU.2015.05.006.

[24] Peter H. Schönemann. "A generalized solution of the orthogonal procrustes problem". In: *Psychometrika 1966 31:1* 31.1 (Mar. 1966), pp. 1–10. ISSN: 1860-0980. DOI: 10.1007/BF02289451. URL: https://link.springer.com/article/10.1007/BF02289451.

[25] *SDK for 3D vision developers - Zivid*. URL: https://www.zivid.com/sdk.

[26] *See more. Do more. Zivid Two industrial 3D camera - Zivid*. URL: https://www.zivid.com/zivid-two.

[27] Magnus Själander, Magnus Jahre, Gunnar Tufte, and Nico Reissmann. "EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure". In: *arXiv:1912.05848 [cs]* (Dec. 2019). arXiv: 1912.05848 [cs].

[28] Víctor Suárez-Paniagua. "Deep Learning for Information Extraction in the Biomedical Domain". PhD thesis. July 2019.

[29] *The KITTI Vision Benchmark Suite*. URL: http://www.cvlibs.net/datasets/kitti/raw_data.php.

[30] *Understanding 1D and 3D Convolution Neural Network | Keras | by Shiva Verma | Towards Data Science*. URL: https://towardsdatascience.com/understanding-1d-and-3d-convolution-neural-network-keras-9d8f76e29610.

[31] Xin Yao. "Evolving artificial neural networks". In: *Proceedings of the IEEE* 87.9 (1999), pp. 1423–1447. DOI: 10.1109/5.784219.

[32] Andy Zeng, Shuran Song, Matthias Nießner, Matthew Fisher, Jianxiong Xiao, and Thomas Funkhouser. "3DMatch: Learning Local Geometric Descriptors from RGB-D Reconstructions". In: *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017* 2017-January (Mar. 2016), pp. 199–208. DOI: 10.48550/arxiv.1603.08182. URL: https://arxiv.org/abs/1603.08182v3.

Appendix A.

Dockerfile for Docker image used on Idun HPC

```
ARG PYTORCH="1.9.0"
ARG CUDA="11.1"
ARG CUDNN="8"

FROM pytorch/pytorch:${PYTORCH}-cuda${CUDA}-cudnn${CUDNN}-devel

##############################################
# You should modify this to match your GPU compute capability
# 8.0 is compatible with A100 cards on Idun HPC
ENV TORCH_CUDA_ARCH_LIST="8.0+PTX"
##############################################

ENV TORCH_NVCC_FLAGS="-Xfatbin -compress-all"

# Install dependencies
RUN apt-get update
RUN apt-get install -y git ninja-build cmake build-essential
↪ libopenblas-dev \
    xterm xauth openssh-server tmux wget mate-desktop-environment-core

RUN apt-get clean
RUN rm -rf /var/lib/apt/lists/*

RUN pip install matplotlib \
                pillow \
                numpy \
                scipy \
                cython \
                scikit-image \
                sklearn \
                opencv-python \
                open3d \
                netCDF4 \
                easydict \
                h5py \
                tensorboardX

# Use only MAX_JOBS=1, build fails otherwise
ENV MAX_JOBS=1
RUN pip install -U git+https://github.com/NVIDIA/MinkowskiEngine -v
↪  --no-deps \
                           --install-option="--force_cuda" \
                           --install-option="--blas=openblas"
```

This Dockerfile is an adaptation of the one provided in the MinkowskiEngine repository [21].

Appendix B.

Zivid_DGAP: Dataset Generator and Data Preprocessor

```python
import numpy as np
import argparse
from os import path, walk
import os
import shutil
from datetime import date
import time
import random
import netCDF4

def loadPointCloudFromZivid(pcd):
    zividPointCloud = netCDF4.Dataset(pcd,'a', format = "NETCDF4")
    return np.reshape(np.asarray(zividPointCloud["data/pointcloud"])
    ↪  ,(1920*1200,3))

def loadRGBFromZivid(pcd):
    zividPointCloud = netCDF4.Dataset(pcd, 'a', format = "NETCDF4")
    return np.reshape(np.asarray(zividPointCloud["data/rgba_image"])
    ↪  [:,:,0:3], (1920*1200,3))

def readZividPCD(framefile,subsample):
    frame = zivid.frame.Frame(framefile)
    pcd = frame.point_cloud()

    if subsample is not None:
        pcd = pcd.downsample(subsample)

    xyz = pcd.copy_data("xyz")
    rgba = pcd.copy_data("rgba")
    normals = pcd.copy_data("normals")

    return xyz, rgba, normals


def saveZividPcdAsNpz(savedir,savename,framefile,normVectorSave = False, save_color=False, subsample = None,scale_to_meters = True):
    if not os.path.exists(savedir):
        os.mkdir(savedir)

    xyz,rgba,normals = readZividPCD(framefile=framefile,subsample=subsample)

    if scale_to_meters:
        xyz = xyz/1000

    if  normVectorSave and save_color:
        np.savez_compressed(os.path.join(savedir,savename), rgb = rgba[:, :, 0:3].reshape(np.shape(xyz)[0]*np.shape(xyz)[1], 3), pcd = xyz.reshape(np.shape(xyz)[0]*np.shape(xyz)[1], 3), normals = normals)

    elif  normVectorSave and not save_color:
        np.savez_compressed(os.path.join(savedir,savename), pcd = xyz.reshape(np.shape(xyz)[0]*np.shape(xyz)[1], 3), normals = normals)

    elif  not normVectorSave and save_color:
        np.savez_compressed(os.path.join(savedir,savename), rgb = rgba[:,:, 0:3].reshape(np.shape(xyz)[0]*np.shape(xyz)[1], 3), pcd = xyz.reshape(np.shape(xyz)[0]*np.shape(xyz)[1], 3))
    elif   not normVectorSave and not save_color:
        np.savez_compressed(os.path.join(savedir,savename), pcd =
        ↪  xyz.reshape(np.shape(xyz)[0]*np.shape(xyz)[1], 3))

def saveZividPcdAsNpzNetCDF(savedir, savename, framefile,
↪  save_color=False, scale_to_meters = True):


    xyz = loadPointCloudFromZivid(framefile)
    if scale_to_meters:
        xyz=xyz/1000
    rgb = loadRGBFromZivid(framefile)

    if save_color:
        np.savez_compressed(os.path.join(savedir,savename),
                                    pcd = xyz,
                                    rgb = rgb)
    else:
        np.savez(os.path.join(savedir,savename),
                                    pcd = xyz)


def testZividSaveLoad():
    FILEPATH = "C:\Users\jonas\Documents\NTNU\" +
    ↪  "MasterThesis\DummyDataset\Object1\black cable - zvd1ps.zdf"
    saveZividPcdAsNpz('./testfolder','testname',FILEPATH)

    loaded = np.load('./testfolder/testname.npz')
    print(loaded["rgba"][200,200])
    print(loaded["xyz"][200,200])
    print(loaded["normals"][200,200])

def splitTrainVal(objectList,trainFile,valFile,valFraction = 0.1):

    trainN = int(len(objectList)*1-valFraction)

    trainList = random.sample(objectList,trainN)
```

```python
    valList = []

    for n in objectList:
        if not trainList.__contains__(n):
            valList.append(n)


    f = open(trainFile,"a")
    for t in trainList:
        f.write(f"{t}\n")
    f.close()

    f = open(valFile,"a")
    for t in valList:
        f.write(f"{t}\n")
    f.close()


def main():
    #Parsing arguments
    parser = argparse.ArgumentParser()
    parser.add_argument("--dir",help="Directory of Zivid
    ↪  pointclouds",type = str,default =
    ↪  "C:/Users/jonas/Documents/NTNU/MasterThesis/DummyDataset")
    parser.add_argument("--zivid_camera_file",help="Path to .zfc camera
    ↪  for camera emulation", type=str, default =
    ↪  "./FileCameraZividOne.zfc")
    parser.add_argument("--val_fraction", type=float, default=0.1)
    parser.add_argument("--include_normals",type=bool, default=False)
    parser.add_argument("--include_color",type= bool, default=False)
    parser.add_argument("--subsample", type=str, default = None, help =
    ↪  "Default is no subsamplng, ommit this flag for raw data input.
    ↪  For subsampling input arguments by4x4 by3x3 or by2x2 ")
    parser.add_argument("--dataset_output_name",type=str,default =
    ↪  "ZividDataset")
    parser.add_argument("--train_file_name",type= str,
    ↪  default="trainZivid.txt")
    parser.add_argument("--val_file_name",type= str,
    ↪  default="valZivid.txt")
    parser.add_argument("--scale_to_meters", type=bool,default=True)
    parser.add_argument("--zivid", type=bool,default=False)
    parser.add_argument("--no_compress", type = bool, default=False)

    args = parser.parse_args()


    if args.zivid:
        import zivid
        # Connecting ZividFileCamera
        app = zivid.Application()
        camera = app.create_file_camera(args.zivid_camera_file)
        settings =
        ↪  zivid.Settings(acquisitions=[zivid.Settings.Acquisition()])
        #frame = camera.capture(settings)

    datasetFolder = args.dir

    print("\n")
    print("Preprocessing Zivid pointclouds")
    print("\n")
    print(f"=> Processing data in {datasetFolder}")

    directories = []

    for (dirpath, dirnames, filenames) in walk(datasetFolder):
        directories.extend(dirnames)


    if args.no_compress:
        outputFolder = args.dataset_output_name
    else:
        outputFolder = outputFolder


    if os.path.exists(outputFolder):
        shutil.rmtree(outputFolder)
    os.makedirs(outputFolder)

    splitTrainVal(directories, os.path.join(outputFolder,
    ↪  args.train_file_name), os.path.join(outputFolder,
    ↪  args.val_file_name), args.val_fraction)

    for dir in directories:
        print(f"\nNow processing files in
        ↪  {os.path.join(datasetFolder,dir)}")
        files = os.listdir(os.path.join(datasetFolder,dir))
        i = 0
        f = open(os.path.join(outputFolder,str(dir)+"_all.txt"),"x")
        for file in files:


            if args.zivid:
                saveZividPcdAsNpz(outputFolder, f"{dir}_{i}",
                ↪  os.path.join(datasetFolder, dir,
                ↪  file),normVectorSave=args.include_normals,
                ↪  save_color=args.include_color,
                ↪  subsample=args.subsample, scale_to_meters =
                ↪  args.scale_to_meters)
            else:
                saveZividPcdAsNpzNetCDF(outputFolder, f"{dir}_{i}",
                ↪  os.path.join(datasetFolder, dir, file),
                ↪  scale_to_meters = args.scale_to_meters)

            f.write(f"{dir}_{i}\n")
            i += 1
        f.close()

    #Create .txt files
    filesInSaveFolder = os.listdir(outputFolder)
    for file in filesInSaveFolder:
        if file.__contains__(".txt"):
            name = file.split("_all.")

            listFile = open(f"{os.path.join(outputFolder,file)}","r")
            listFileContents = listFile.read().split("\n")
            pairFile = open(f"{os.path.join(outputFolder,
            ↪  name[0])}.txt", "x")

            listFileContents1 = listFileContents
            usedElements = []
            for element1 in listFileContents1:
                for element2 in listFileContents1:
                    if element1 == element2 or len(element1)<2 or
                    ↪  len(element2)<2 or
                    ↪  usedElements.__contains__(element1) or
                    ↪  usedElements.__contains__(element2) :
                        None
                    else:
                        pairFile.write(f"{element1}.npz
                        ↪  {element2}.npz\n")

                usedElements.append(element1)

            listFile.close()
            pairFile.close()

    t = time.localtime()
    current_time = time.strftime("%H%M", t)

    if not args.no_compress:
        shutil.make_archive(f"{args.dataset_output_name}_" + "ss_{
        ↪  args.subsample}_colors_{args.include_color}_normals_{
        ↪  args.include_normals}_{
        ↪  str(date.today())}_{str(current_time)}" , 'zip',
        ↪  outputFolder)
        shutil.rmtree(outputFolder)



if __name__=="__main__":

   main()
```

Appendix C.

Shell Script for Training FCGF Model

```bash
#!/bin/bash
export OMP_NUM_THREADS=12
cd ./FCGF
nvidia-smi

echo "#################################################"
echo "              TRAINING FCGF"
echo "#################################################"


python ./train.py \
                    --batch_size=4 \
                    --voxel_size=0.002 \
                    --stat_freq=10\
                    --max_epoch=200 \
                    --train_num_thread=12 \
                    --val_num_thread=12 \
                    --lr 1e-1\
```

Appendix D.

Slurm File for Training FCGF Model

```
#!/bin/bash
#SBATCH --job-name=FCGF_train
#SBATCH --mail-type=ALL
#SBATCH --mail-user=jonassaa@stud.ntnu.no
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=12
#SBATCH --mem=16000
#SBATCH --time=96:00:00
#SBATCH --output =
↪  /cluster/home/jonassaa/jobs/FCGF/outputs/FCGF_train.log
#SBATCH --partition=GPUQ
#SBATCH --gres=gpu:A100m80:1

date;hostname;pwd

singularity exec --nv plain_minkowskiengine_latest.sif
↪  /cluster/home/jonassaa/jobs/singularityExecScripts/train_FCGF.sh

date
```

Appendix E.

Shell Script for Training DGR Model

```bash
#! /bin/bash
export OMP_NUM_THREADS=12
cd /cluster/home/jonassaa/jobs/DeepGlobalRegistration
nvidia-smi

echo "##############################################"
echo "                  TRAINING DGR"
echo "##############################################"
python ./train.py \
  --dataset ZividPairDataset \
  --zivid_dataset_dir "/cluster/home/jonassaa/jobs/ZividOne_v2/" \
  --feat_model ResUNetBN2C \
  --feat_model_n_out=32 \
  --feat_conv1_kernel_size=7 \
  --lr 1e-1 \
  --batch_size=2 \
  --val_batch_size=1 \
  --max_epoch=40 \
  --voxel_size 0.001 \
  --hit_ratio_thresh=0.0425 \
  --weights "/cluster/home/jonassaa/modelsFCGF/3Dmatch_25mm_32dim.pth"
```

Appendix F.

Slurm File for Training DGR Model

```
#!/bin/bash
#SBATCH --job-name=DGR_train
#SBATCH --mail-type=ALL
#SBATCH --mail-user=jonassaa@stud.ntnu.no
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=12
#SBATCH --mem=16000
#SBATCH --time=24:00:00
#SBATCH --output=DGR_train
#SBATCH --partition=GPUQ
#SBATCH --gres=gpu:A100m80:1

date;hostname;pwd


singularity exec --nv
↪  /cluster/home/jonassaa/jobs/plain_minkowskiengine_latest.sif
↪  /cluster/home/jonassaa/jobs/singularityExecScripts/trainDGR.sh

date
```

Appendix G.

Zipfile

This appendix describes the folder structure of the accompanying zipfile uploaded alongside this thesis. In this Zipfile, all code used to produce the results presented in this thesis is provided as well as the data set of Zivid point clouds.

- CodeRepositories
  - DGR
  - FCGF
  - Zivid_DGAP
- IdunHpcScripts
  - SingularityExecScripts
    - train_FCGF_3dmatch.sh
    - train_FCGF_Zivid.sh
    - train_DGR_Zivid.sh
  - SlurmScripts
    - train_FCGF_3dmatch.slurm
    - train_FCGF_Zivid.slurm
    - train_DGR_Zivid.slurm
- Plotting
  - plotFromLog.py
- TrainingLogs
  - train_FCGF_3dmatch_log.log

  - train_FCGF_Zivid_log.log
  - train_DGR_Zivid_log.log
- ZividDataset
  - Processed
    - ZividOneDataset
  - Raw
    - ZividOneDataset
- Project_thesis_Jonas_Strand_Aasberg.pdf