Master's thesis

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

Philip Øygarden Puente

Deep Learning on 3D Feature Descriptors

Master's thesis in Informatics: Artificial Intelligence
Supervisor: Theoharis Theoharis, Bart Iver van Blokland
June 2020

NTNU
Norwegian University of Science and Technology

Philip Øygarden Puente

Deep Learning on 3D Feature Descriptors

Master's thesis in Informatics: Artificial Intelligence
Supervisor: Theoharis Theoharis, Bart Iver van Blokland
June 2020

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

NTNU
Norwegian University of
Science and Technology

I am grateful to my supervisor, Theoharis Theoharis and co-supervisor Bart Iver van Blokland for giving guidance and technical support during this thesis.

I would also like to thank my family and friends for all the motivational support and encouragement they have given. This work would have been impossible without them.

Summary

In recent years, 3D-shape data becomes more readily available due to commercially available sensors and 3D printing. It is therefore intuitive to want to apply 2D deep learning techniques to 3D data, as 2D techniques have proven to be very successful in a variety of use-cases. However, the irregular size of 3D data makes this difficult. Perhaps it would it be feasible to leverage 3D feature descriptors as a regularisation transformation for the data? Experiments done with Siamese neural networks trained on Spin Images, Viewpoint Feature Histograms and Fast Point Feature Histograms, show that not only is it feasible, but more accurate compared to more primitive point cloud trimming methods. Further more 3D feature descriptors is an effective way to reduce the overall size of the network while maintaining good accuracy, decreasing training and prediction time.

I de siste årene har 3D-figur data blitt lettere å få tak i, delvis grunnet kommersielle sensorer og utbredt bruk av 3D-printere. Det er derfor ønskelig å ta i bruk dyp-læringsmetoder, som har vist seg å være anvendbare for mange ulike formål, og tilpasse de til 3D data. Den uregelmessige størrelsen på 3D data gjør det dessverre vanskelig å overføre disse teknikkene direkte. Kanskje er det mulig å ta i bruk 3D egenskapsbeskrivelser som et omformingssteg for å regularisere dataen? Eksperimenter der Siamesiske nevrale nettverk ble trent opp på Spin Images, Viewpoint Feature Histogram og Fast Point Feature Histogram, viser at det ikke bare er mulig, men det gir mer presise resultater sammenlignet med en mer primitiv punktsky trimmingsmetode. Ikke nok er det, men 3D egenskapsbekrivelser kan også hjelpe til å redusere størrelsen på nevrale nettverk, uten å ofre nøyaktighet. Dette bidrar til å gjøre det raskere for nettverkene å trenes opp og å gjøre prognoser.

Preface

This thesis is the final submission of work done for the Master of Science degree in the Artificial Intelligence program at the Norwegian University of Science and Technology (NTNU), Department of Computer Science. The work in the thesis was done during the fall semester of 2019 and spring semester of 2020. Unfortunately, due to a a pivot in direction, a lot of the work done during the fall semester turned out to be irrelevant for this thesis. The work was still very educational and helped me build a better understanding of the computer graphics field. This thesis aims to explore a combination of computer graphics, computer vision and machine learning, as these are some subjects that interest me personally.

Table of Contents

List of Tables

List of Figures

Abbreviations

ANN	=	Artificial Neural Network
API	=	Application Programming Interface
CAD	=	Computer-Aided Design
CNN	=	Convolutional Neural Network
DL	=	Deep Learning
FPFH	=	Fast Point Feature Historgram
GPU	=	Graphics Processing Unit
LiDAR	=	Light Detection And Ranging
ML	=	Machine Learning
PCL	=	Point Cloud Library
ReLU	=	Rectified Linear Unit
SNN	=	Siamese Neural Network
SI	=	Spin Image
VFH	=	Viewpoint Feature Histogram
VR	=	Virtual Reality

Chapter 1

Introduction

1.1 Motivation

The way a problem is represented can severely impact the ability to provide an efficient solution. In fields like computer graphics and computer vision, the representation of 3D models has played an important part in being able to provide fast (often real-time) rendering, segmentation and classification of objects. The past decade has given rise to significant advances in computer vision applied to 2D images and video, especially in the field of Machine Learning (ML) and later Deep Learning (DL) [18].

However, applying the same techniques to 3D data has proven more difficult. One of the reasons for this lies in the properties of the representation of 2D vs 3D data. The most ubiquitous form of 2D data are pixel images, for 3D data the most common representation is 3D-mesh data. The advantage with pixel images when it comes to Deep Learning, is the underlying grid array structure, the property of global parametrization and having a global coordinate system [2]. These are all properties 3D-meshes lack. While 3D voxelization methods that maintain these properties for 3D-data exists, they run into issues when scaling up the resolution, as they have cubic growth, as well as sparsity in the representation of data [34].

Furthermore it would be ideal to take advantage of the increasing amounts of 3D mesh data that has become available in recent years. This is due to the increased availability of 3D and depth sensors, like in (semi-)autonomous vehicles, 3D spatial tracking for Virtual Reality (VR) and game systems, and the aggregation of 3D models to websites from the widespread adoption of commercial 3D-printers.

Using 3D-meshes directly still presents some problems. First, this kind of data is highly irregular, as the size (amount of vertices) of the input depends directly on the model itself. Secondly, the representation is complex, consisting of multiple different aspects, like vertices, edges, normals and sometimes color and texture information. These are not trivial problems, as is evident by an increasing amount of publications in the emerging field of 3D Deep Learning. This paper looks to contribute to that field by looking at possible transformations of the 3D-mesh data that would simplify their application to DL systems.

1.2 Research Questions

Research goal:
How can one feed neural networks 3D-shape data, given the irregular size of common 3D model representations? Traditional methods, such as 3D feature descriptors, are designed to extract information about shapes of arbitrary sizes. Would using these descriptors as a transformation to regularize the input be sensible, and how would this compare to existing methods of feeding 3D data to neural networks?

- **RQ1:** Would existing 3D feature descriptors be an applicable transformation to regularize 3D-shape data?
- **RQ2:** How does the descriptor method compare to other methods, such as point cloud trimming, in tasks such as model comparison?

Chapter 2

Background

This chapter will outline the needed theoretical background required to answer the research questions. The chapter contains three main sections, Machine learning (ML), 3D Models and Descriptors. Section 2.1 gives a rough overview of the field of Machine Learning, then describes different kinds of learning and some specific methods and architectures. Section 2.2 consists of an overview of 3D model representations, as well as their advantages and disadvantages in relation to ML. Section 2.3 will go into more detail about different 3D feature descriptors and some data structures needed to compute said descriptors.

2.1 Machine Learning

The purpose of AI systems is to allow a machine to make informed decisions in an environment. In order to achieve this, the system needs rules, or a knowledge base, with which it can infer decisions. With improvements to statistical methods and new systems for inference, more complex problems could be solved. However, these systems would still be limited by what knowledge the programmers had of the problem-space. In order to stop relying on human knowledge, machines would need to be able to learn on their own. In other words, the AI would need to go from being a static problem-solver to an adaptive one.

In practice ML systems can be thought of as a mapping or a function from an input (typically denoted X) to an output (typically denoted Y). The input can, for example, be a representation of the environment, referred to as ”state”. The output would then be a transformation of that state in some way, for example what action to perform next.

Learning can be thought of as the principle of extracting information from data. This usually means analysing and finding patterns in the data which one can use to make generalizations. These generalizations allow one to make informed decisions based on input. It is common to divide different learning methods into categories based on what type of information the dataset contains and how it is used:

Supervised Learning

We refer to the learning process as ”supervised”, when we have a labeled dataset. This means we have both the input and the correct answer is known in advance. In ML terms one would say that for every training case, the label (or ground truth) is known. Since we the answer is known we can use it to calculate an accuracy of the model and adjust it based on how it performs. A real life example of supervised learning would be a math test where you get feedback on how many questions you got right and wrong. This method is typically used to make predictions (regression) and classification tasks.

Unsupervised Learning

In this category we only have an unlabeled dataset. This might be because the dataset is too difficult or expensive to label or perhaps we don not even know how to label it properly. Here the model would train by trying to structure the dataset by extracting patterns and cluster similar cases together, allowing for generalization. A real life example could be learning to recognize handwritten letters. Typical tasks involve feature extraction, clustering, anomaly detection, association and noise removal.

Semi-supervised Learning

In recent times it has become more and more common to have access to large unlabeled datasets. These can be labeled manually, but this is labor intensive and slow. A workaround for this is to label a small amount of the data, train on it. Then use the model to label the rest of the data. This can be done iteratively to improve the labeling as the model itself improves. The result would be similar to supervised learning.

Reinforced Learning

This method is quite different from the others. It relies on a reward function to optimize. The reward function is an external measure of how well the model performed for a given input. The way humans can use snacks as a reward when training dogs is an example of reinforcement learning. This method is suitable for making predictions, decision making and situations where one does not have a large dataset available and wants to do continuous real-time training.

You can find more in-depth articles about the different kinds of learning on Nvidias blog [14]. In this thesis we will focus on using supervised learning.

2.1.1 Artificial Neural Networks

An artificial neural network is a data structure that mimics how the brain, a biological neural networks, functions, by copying the concepts of neurons and neural plasticity. The ANN is not a new idea by any measure. The concept can actually be traced back to the very beginnings of the computer in the late 1940s and onwards with computational models for neural networks [8] and the concept of Hebbian learning [13]. In 1958 the term ”perceptron” was introduced [26], this refers to a node in the ANN and would be analogous to a neuron.

ANNs make use of weights, biases and activation functions that create connections and control the information flow throughout the network. This is similar to how synapses and action potential control the ”firing” of a neuron and thus information flow in the brain.

2.1.2 Deep learning

Deep learning (DL) drives the current resurgence of ANNs. With recent advances in data availability and increases in parallel compute power using GPUs, ANN models have become more and more versatile problem solvers and the go-to data structures for machine learning. The combination of easily available data in large quantities and massively increased power to process it meant that networks could become larger than before. Along with this comes the realization that deep networks allow for multi-level-abstraction and solving more intricate problems than previously possible [21].

Similar to how machine learning is a subset of artificial intelligence, DL is again a subset of machine learning. What differentiates deep learning from the rest of machine learning is the usage of multiple and larger layers in the network, creating abstractions at different levels

2.1.3 Model Architecture

With the rapid growth of ANNs and DL over the last decade, different models and architectures has become a field of its own. This thesis will focus on Siamese neural networks, which will be presented in Section 2.1.3. Before that some basics terms need to be explained:

Weights and biases

Weights and biases are the name of the floating-point values used to adjust the flow of information through an artificial neural network. Weights are multiplied with the input from the previous layer of the network and biases are added during the summation before the activation function. They work together to allow the neurons to make linearly separable decisions. Selecting an activation function that is not linear is important for being able to combine these decisions, letting the network evaluate more complex problems. The non-linearity is key because of the property of linear composition, this would mean that any combination of linear functions would always result in a linear product. Section 2.1 will explain some non-linear functions in more detail.

Layer types

The first kind of distinction made in the layers of a neural networks is typically to divide the network into input-, hidden- and output-layers. Input- and output-layers are self explanatory. The hidden layers make up everything in-between, and are what actually makes the network interesting.

Within the hidden layer category there are several different kinds of of layers, designed for different tasks. Some layers are fully connected, some only partially. Convolution and Pooling layers can reduce the size of the input. Reshaping layers can pad, flatten or change

the dimensions of the input. You can perform mathematical operations like, addition, division, multiplication or taking the max or averaging the data. Recurrent layers feed into themselves, which allows the networks to retain state and ”remember” the input for the next propagation [5]. Combinations of these different kinds of layers give rise to different behavior and different architectures. It the variety of behaviour in layers and combinations of those behaviours that makes artificial neural networks so versatile.

Activation functions

$$\begin{aligned} sigmoid(x) &= \frac{1}{1+e^{-}x} \\ tanh(x) &= max(0,x) \\ ReLU(x) &= \frac{e^{x}-e^{-}x}{e^{x}+e^{-}x} \end{aligned} \tag{2.1}$$

Activation functions are analogous to action potential in neurons. They dictate the threshold for when each node in the network should fire. In Equation 2.1 you can see three of the more common activation functions and their mathematical definitions. The reason to use different kinds of activation functions is that they have different properties and thus different use-cases. The Sigmoid it good for producing results yes/no results as it squished input values into the (0, 1) range, while Rectified Linear Unit (ReLU) is designed to deal with an issues called vanishing gradient [22]. There are many more kinds of activation functions with different advantages, trade-offs and optimizations. The development of new activation functions has very much become a research topic of its own. Figure 2.1 shows the same activation functions described in Equation 2.1 as graphs, red being the sigmoid, tanh in blue and ReLU in green.

Figure 2.1 Graph showing three of the most common activation functions, sigmoid (red), tanh (blue) and ReLU (green).

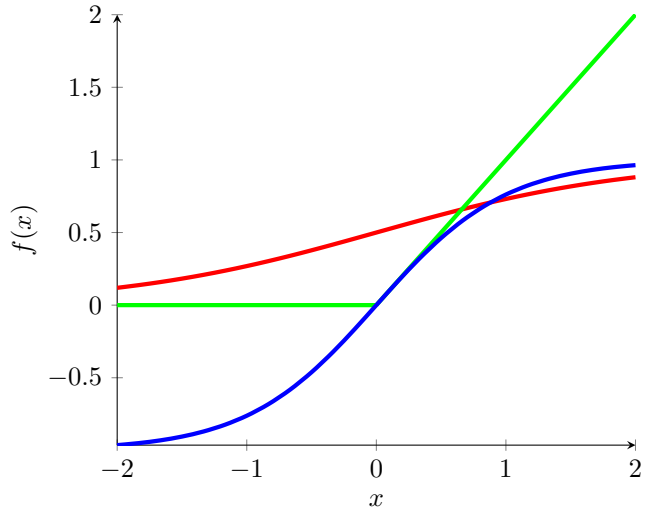
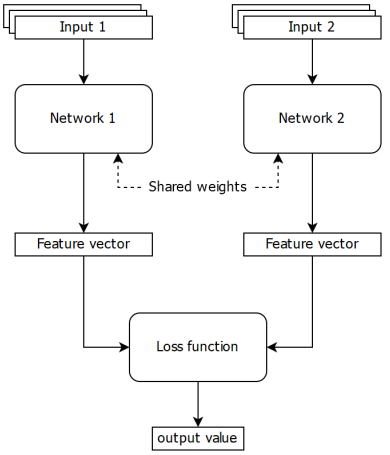


Figure 2.2 Siamese network architecture.



Siamese networks

The core concept of a Siamese network is that it operates on pairs of input data, not individual ones, the task of the network is therefore to learn to evaluate some sort of relation function between the two inputs. To achieve this behavior, the Siamese network consists of two feature encoding networks, that share weights. The two encoders produce a feature vector for one input each, that is then combined in in an aggregation layer. The final activation function is then applied to the output of the aggregation layer.

Siamese networks are great for working with datasets with a significant class imbalance, this is because the pairwise input makes it more robust to the class distribution of the dataset. Due to the nature of the comparison, Siamese networks are forced to learn embeddings and not rely on direct tells from the input data. This also makes the network relatively transferable to new classes. Imagine that you have trained a siamese network to compare fruit, and produce a positive results if it saw two fruit of the same kind. The network is trained on a dataset of common fruit, apples, oranges, bananas, pears, etc. Then let us say you show the network a pineapple. Even if the network has never seen such a sample before, it would likely be able to produce a distinct ”pineapple”-feature vector.

Since this kind of network outputs distances between classes instead of class labels, the network is well suited for comparison tasks rather than direct classification. With some modifications however it can easily function in classification scenarios too. In Figure 2.2 you can see a generalisation of the Siamese neural network architecture.

2.2 3D Models

One of the core questions in geometric machine learning is centered around what representation to choose for 3D-data [2]. Bronstein et al. [4] defines two clear categories of 3D-data which dictates some of the properties of the representation.

- **Euclidean** is data that has an underlying rigid grid structure. In 2D that would be

a pixel (raster) image. In 3D this includes projections, RGB-D data, volumetric representations and multi-view data.

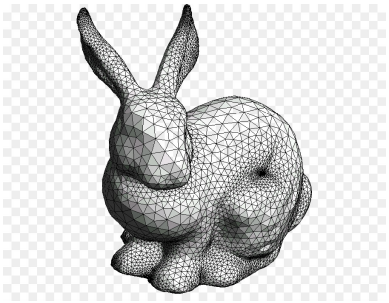
- **Non-Euclidean** is the kind of data that is defined similarly mathematical graphs. In 2D this would be formats like vector graphics. In 3D this includes point clouds, 3D mesh and graph based model representations.

In some ways this can be viewed as a distinction between discretized and continuous representations. The reason this matters is that both of the most common 3D model representations, 3D-mesh and point clouds, are non-Euclidean. Non-Euclidean representations are not regular, which makes them harder to work with in conjugation with neural networks. The other aspect to consider when choosing a representation for usage with ANNs is the information density, both these considerations will be discussed in Chapter 4. Subsection 2.2.1 to 2.2.4 will describe some of the common 3D representations in more detail.

2.2.1 Polygonal Mesh

The polygonal mesh, also called 3D mesh or triangle mesh, is likely the most common 3D model representation. In practice there are several implementations methods to represent a polygonal mesh, but the common denominator for them is a set of vertices and polygons. The vertices decide the placement of the shape in 3D coordinates and the polygons dictate which vertices connect together to form the surface of the model. The compactness and lack of restrictions this representation provides, has made it an ubiquitous format throughout video games and 3D graphics in general. In Deep learning contexts, the irregular size of the format provides a challenge as current ML techniques often require an input of a predetermined size. An example of a polygonal mesh can be seen in Figure 2.3.

Figure 2.3 Depiction of the Stanford bunny as a polygonal mesh



2.2.2 Multi-view projections

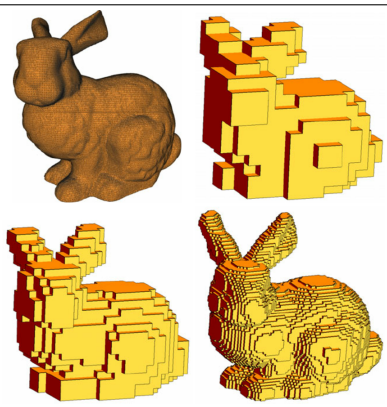
The use of 2D images is already widespread within the DL community. Object classification methods using this representation is already well established, see [11] or [33]. Multi-view projections look to apply the same approach as the human visual system, i.e. combining two 2D images to extract 3D information, like depth. This is often expanded

to more than two views (hence the ”multi”), in order to give sufficient viewing angles of the 3D model. Here arises a problem however, as too few views can cause a significant loss of information (occlusion). At the same time, too many views will require excessive amounts of computational power.

2.2.3 Volumetric

Volumetric representations look at the occupancy in a given bounding box around the model. Occupancy here simply means ”Is there a piece of the model inside this section?”. This is great for applications where the explicit solidness of the object is important. While volumetric representations are great for working with general shapes/volumes, they do struggle with loosing information about the surface/smoothness. The two most common volumetric representation are Voxelization and octrees.

Figure 2.4 Depiction of the Stanford bunny voxelized in different resolutions, taken from Karmakar et al. [17]



Voxelization

A Voxel representation consists of a regular grid in 3D space. This is the 3D analogue of a of a 2D pixel-image. Just like how the unit of an image is called a pixel, the unit of a volumetric representation is often called a voxel. Similarly to a pixel, each voxel of the grid contains a sampling of the model. This representation is both intuitive due to its similarity to pixel images and convenient due to its model independent and adjustable grid size. The drawback of this is that the size of the representation scales with the cube of the resolution required for the application. Depending of the compactness of the 3D models, a lot of the space represented can also be left empty, drastically increasing the memory needed compared to other methods. An example of a the voxelization method can be seen Figure 2.3.

Octree

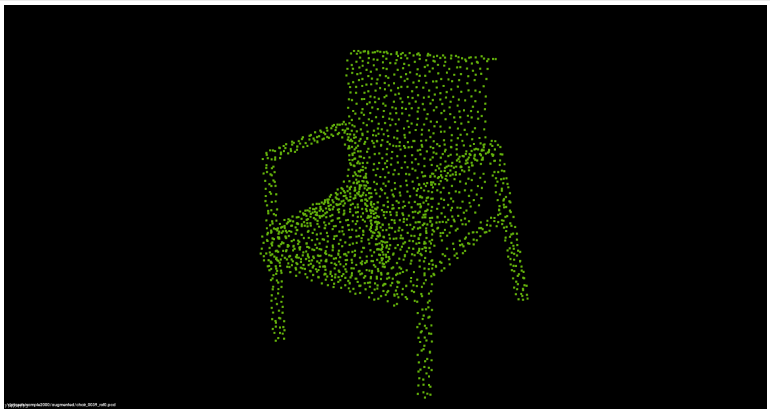
The octree is a more space efficient adaption of the voxel system. It is the 3D analogue of the quadtree, that some image compression algorithms use [20]. This method yields a hierarchical structure which can be very efficient when working with neighbourhoods. The idea here to use varying sized voxel volumes, recursively decomposing the models such that the leaf volumes are either fully inside or fully outside (limitations to recursive depth can also be applied). For solid objects this can save a lot of space as large sections can be stored in shallow parts of the tree.

2.2.4 Point Cloud

Point clouds are a set of un-ordered data points in 3D space. Unlike mesh structures, there is no explicit relation, or connection between the points. The data points in the point could are homogeneous, meaning they all contain the same type of data, thus the point cloud is not considered complex like the mesh. The exact data each data-point contains, however, can vary from the point cloud implementation. Many only store the x, y, and z coordinates for each point, others include the normal vectors for each point or even color information as well. formats, like *.pcd* can even adjust what information they contain by making use of meta-data in the header of the file. Recently many fields like, medical imaging, architecture, construction, autonomous vehicles and archaeology have taken advantage of cheaper 3D sensor and LiDAR technology. These sensors typically produce point clouds, as this representation is not constrained to grids or other limitations.

Many tools and algorithms has been developed for working with point clouds. There is even a large library, aptly named the Point Cloud Library [30], that is a collection of such methods. Feature descriptors are a partially overlapping set amongst such methods and will be discussed in Section 2.3. In Figure 2.5 you can see a render of a point cloud.

Figure 2.5 An example of a point cloud model. Rendered using the pcl_viewer comandline tool.



2.3 Feature Descriptors

A 3D feature descriptor is, as its name implies, an alternative way to describe a 3D shape. Analogous to how one can paint a mental image of a song by summarising it's genre, tempo, scale and instruments. A 3D model can also be presented through features extracted from the shape itself, such as curvature, eccentricity and other less tangible features. These descriptors are typically a more succinct representation compared to the raw shape data. Different kinds of features are relevant for different kinds of task, and have therefore given rise to different descriptor methods. Even though the result of descriptor generation varies widely, the generation of those descriptors used in this thesis start with a few common steps.

First we need to load the 3D-shape data, this is usually stored as point clouds or 3D-mesh files of varying formats. If the format does not contain normal vectors for each point, they need to be calculated. For polygonal meshes this calculation is relatively simple, just add together the normals for all the polygons that contains the given vertex and normalize the final result. For point clouds the process is a bit more involved, but there exists several methods such as described in Zhou et al. [36]. When the normal information is present, the next step is to select what vertices the feature vectors will be computed for. The selected vertices will be referred to as keypoints. Most feature descriptors use local information around the keypoint to calculate the values of the vector, typically with a geometric volume, called a support region, centered at the keypoint. Several algorithms therefore require the neighbourhood of each keypoint to be calculated. With these additional data structures the final computation of the descriptor can be done.

The nearest neighbourhood search can be optimized using KD-Trees. All the descriptor algorithms used in this paper generate a KD-tree as an initial step to speed up this search. The next section will therefore detail nearest neighbourhood searches and KD-trees.

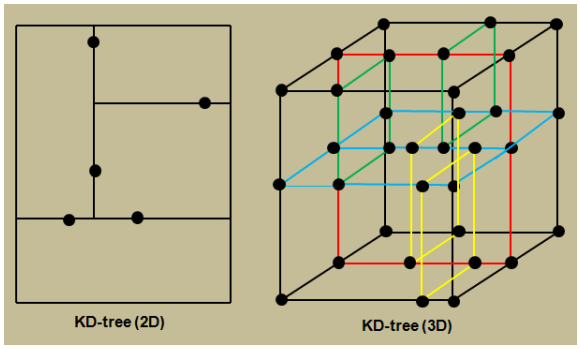
2.3.1 Nearest Neighbour Search

It is not uncommon to make use of hierarchical structures like trees when dealing with finite-element search problems, neighbourhood search is no different. In section 2.2.3 we described the octree representation of 3D shapes. While octrees can and has been used for spacial partitioning of 3D shapes in algorithms [7], there exists better methods for this exact task, namely kD-trees.

KD-trees are fairly similar to octrees, but hold an advantage over them when it comes to computation speed. This advantage is a result of the way they is calculated. Instead of partitioning the model based off the value space the model exists in, like with octrees. KD-trees partition by the values of the model itself. This results in a better balanced tree, normalizing the average search time.

KD-trees are calculated by doing the following. Each level of the kD-tree has two children and much like with binary trees, vertices are divided based of an ordering and a pivot value. This division is done along one dimension at a time (X, Y or Z for 3D models) and the pivot value is calculated based on the values of the vertices along the given dimension. This means that each level of the kD-tree corresponds to a separation along one dimension. Most kD-tree implementations calculate the mean value of the set

Figure 2.6 Visualisation of KD-trees in both 2D and 3D, image take from Park et al. [24]

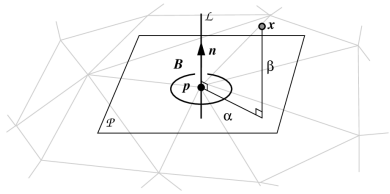


and use that as the pivot value. Due to the nature of the mean operator, the result is a balanced split of the vertex-set (save for odd sized sets). A visual representation of KD-trees can be seen in Figure 2.6.

Another advantage KD-trees give come from the traversal of the tree when it is being used by the search algorithm. Since each level of the tree is only split in two, one can quickly decide what branch to traverse, compare this to an octrees 8-way split. This might seem trivial, but the reduction in the required decision making at each step, has a significant impact.

2.3.2 Spin-Image (SI)

Figure 2.7 Oriented point basis [15]



Spin-images, introduced by Johnson [15], is a descriptor designed for use in 3D-surface matching and object detection, even in occluded scenes [16]. The descriptor encodes surface information in an object-oriented coordinate system, making it independent of the viewpoint of the scene itself.

The first step to creating a spin-image is to construct an oriented point basis, seen in Figure 2.7. The keypoint p together with its normal n forms an oriented point o. The 3D-surface information can then mapped to a 2D spin-map by using a cylindrical support region, centered at o, as a basis for a new coordinate system. One dimension of the 2D space is defined along P, the normal plane of the oriented point o. The other dimension is corresponds to L, parallel to the normal vector n. Along these to axis we define coordinates α and β. α being the radial distance to the surface normal line and β being the

axial distance to the tangent plane, making the coordinate system radially independent. The projection function, mapping 3D points x to α and β coordinates are calculated with equation 2.2 [15]. Where S_o denotes the spin-map at the oriented point.

$$\begin{aligned} S_o : R^3 \rightarrow R^2 \\ S_o(x) \rightarrow (\alpha, \beta) = (\sqrt{\|x-p\|^2 - (n \cdot (x-p))^2}, n \cdot (x-p)) \end{aligned} \tag{2.2}$$

The next step is to transform the spin-maps into spin images. This process is done by accumulating the points of the spin-map into pixel values. The α and β coordinates first have to be transformed to pixel coordinates i and j using Equation 2.3. Where W is the image width and B is the number of accumulator bins.

$$j = \left\lfloor \frac{\alpha}{B} \right\rfloor \quad i = \left\lfloor \frac{\frac{W}{2} - \beta}{B} \right\rfloor \tag{2.3}$$

Then each points contribution to the bin values is smoothed over the neighbouring bins using bi-linear interpolation. A and B are the interpolated weights for α and β and calculated like in Equation 2.4. Finally the weights are accumulated for each of the surrounding bins with Equation 2.5

$$\begin{aligned} a &= \alpha - j \cdot B \\ b &= \beta - i \cdot B \end{aligned} \tag{2.4}$$

$$\begin{aligned} SI(i,j) &= SI(i,j) + (1-a) \cdot (1-b) \\ SI(i+1,j) &= SI(i+1,j) + (a) \cdot (1-b) \\ SI(i,j+1) &= SI(i,j+1) + (1-a) \cdot b \\ SI(i+1,j+1) &= SI(i+1,j+1) + a \cdot b \end{aligned} \tag{2.5}$$

In Figure 2.8 you can see an example of both the spin-maps and the resulting spin-image for three keypoints on the model. An interesting note about the Spin Image descriptor is that it can function both as a local and as a global feature descriptor by adjusting the parameters of the spin-image generation, like image width and bin size [15].

2.3.3 Point Feature Histogram (PFH)

PFH is a descriptor developed by Rusu et al. [27]. The descriptor makes use of a spherical support region and takes a point cloud, with xyz-coodinates and normal vectors as input. Then a kd-tree is computed, this tree is used for performing neighborhood search. For each point p, the set of neighbours enclosed in the spherical support region with radius r is identified. Then for every pair of points p_i and p_j where $i \neq j$ in the neighborhood and n_i and n_j denote their respective normals, a Darboux uvn frame is defined. For this it is important that we choose p_i and p_j such that p_i has a smaller angle between its normal and the line connecting it to p_j. The frame axis are defined by u, v and w in Equation 2.6. Then the angular variations are computed as shown in Equation 2.7 and stored in the

Figure 2.8 Depiction of spin-maps and spin-images for 3 selected keypoints. Figure 2-2 from Johnson [15]

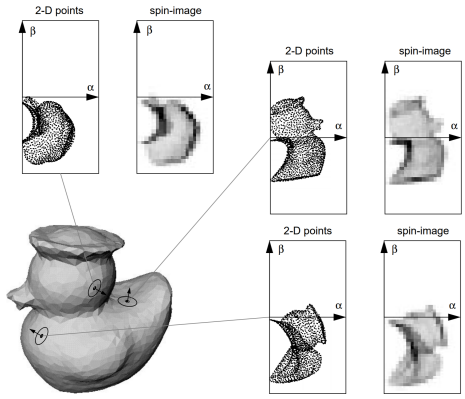
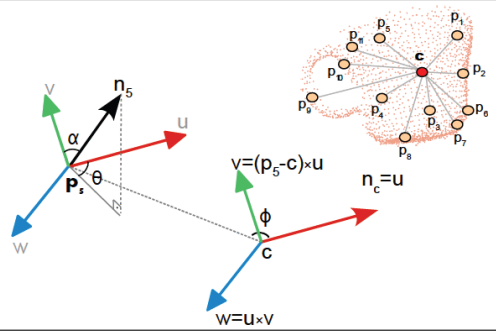


Figure 2.9 A graphical representation of the PFH features around two points. [28]



histogram. Figure 2.9 show an example of the different aspects of PFH from point c to p_5 on the model to the right, the figure is taken from chapter V in Rusu et al. [28].

$$\begin{aligned} u &= n_i \\ v &= (p_j p_i) \times u \\ w &= uv \end{aligned} \tag{2.6}$$

$$\begin{aligned} \alpha &= v \cdot n_j \\ \phi &= \frac{u \cdot (p_j p_i)}{|p_j p_i|} \\ \theta &= arctan(w \cdot n_j, u \cdot n_j) \end{aligned} \tag{2.7}$$

2.3.4 Fast Point Feature Histogram (FPFH)

In Rusu et al. [29], it is noted that the theoretical computational complexity of Point Feature Histograms is $O(n \cdot k^2)$. Where n is the number of points in a point cloud and k is

the number of neighbours for each point. This growth is not ideal, especially for dense point clouds. In Rusu et al. [29] a simplified version of the algorithm is proposed, with the lowered complexity of $O(n \cdot k)$.

To generate FPFH we first define $SPF()$, which is the angular variations in Equation 2.7, but only computed for point p and its neighbours, not all permutations of neighbors within the support radius. Then the histogram is weighted using the neighbouring SPF values for each point as shown in Equation 2.8 where ω_k is the distance between point p and the neighbour p_k.

$$FPFH(p) = SPF(p) + \frac{1}{k}\sum_{i=1}^{k}\frac{1}{\omega_k} \cdot SPF(p_k) \tag{2.8}$$

2.3.5 Viewpoint Feature Histogram (VFH)

Viewpoint Feature Histogram further builds upon the FPFH algorithm, in Rusu et al. [31], it was shown that the FPFH descriptor would act as a global descriptor by increasing the r limit on the nearest neighbour search to encompass the entire pointcloud. This method was further build upon in Rusu et al. [28], where by combining this extended FPFH descriptor with a additional viewpoint statistics would result in the new VFH histogram. The component measures, α, ϕ and θ, are computed between the central point along the view point direction and the normals for the points in the cloud. The viewpoint component consists of a histogram of angles between the viewpoint direction and the normals of each point. The way the VFH descriptor is calculated has two important effects; 1) only one histogram is generated regardless of the size of the input cloud, reducing the size of the descriptor drastically. 2) The complexity of the algorithm is lowered to $O(n)$ where n is the number of points in the cloud.

Chapter 3

Related Work

This chapter presents relevant research and contributions to the fields of computer graphics, computer vision and deep learning. Much of this work is directly utilized in the construction of the system required to answer the research questions of this thesis. Other work is tangentially related, trying to accomplish the same or similar tasks to the ones outlined in this paper with other combinations of network architectures or 3D-shape representations.

3.1 Graphics and Computer Vision

The work done in the fields of computer graphics, computer vision and 3D-shape analysis, lay the foundations for the descriptors leveraged in this thesis. Specifically Johnson [15] and Johnson and Hebert [16] where the generation and application of Spin Images is outlined. This thesis makes use of Spin Image as one of the feature descriptors fed to artifical neural networks.

Similarly, the many works of Rusu et al. have been vital to this thesis. In Rusu et al. [27] the PFH descriptor is presented. This descriptor became the basis of many improved and more specialized descriptors in later works. Rusu et al. [29], details one of these improvements, where a slight loss of detail is traded in for significant computation speed improvements, allowing for usage in real-time systems. In Rusu et al. [31], another application for the FPFH descriptor is presented. Experimental results showed that, by increasing the neighbourhood radius to encompass the entire model, the local feature descriptor could also function well globally. This further lead to a more specialized feature descriptor, called VFH [28], that expanded further into the global descriptor space by baking in a viewpoint component into the histogram. FPFH and VFH are the other feature descriptors used to train and evaluate networks in this thesis.

Rusu has also been involved with the creation of the Point Cloud Library [30], a collection of many algorithms related to the point cloud representation of 3D-shapes, written in C++. The feature descriptor implementations, in PCL where used in this thesis for generation of mentioned feature descriptors. The KD-tree implementation from PCL was also leveraged to speed up nearest neighbour searches during the generation of descriptors.

3.2 3D Machine Learning

In terms of machine learning, the work done to create tools like Tensorflow [1] and Keras [5] has been invaluable. These libraries make it ridiculously easier to quickly prototype and test networks. The creation of the ModelNet, a large 3D-model dataset intended for computer vision and learning systems, by Zhirong Wu et al. [35], has proven to be very helpful. Collecting and labeling such large datasets takes significant time. It is great that they then publish and openly distribute their efforts online, saving time for many projects, including this one.

The ease of access of both tools to construct networks and datasets to train them with, has given rise to many tangential approaches of combining 3D-data and ANNs. For example, MeshNet by Feng et al. [9]. A network trained to perform classification and retrieval directly with 3D-mesh models by using a creative network architecture. Voxelnet by Brock et al. [3], getting good results in generation and discrimination tasks by applying a volumetric transformation to regularize the mesh data. Similarly in Wu et al. [34], they fed volumetric data to several different networks, leveraging methods such as variational auto-encoders, convolution and taking inspiration from architectures like ResNet [12]. PPFNet is another interesting take, were they utilize the power of ANNs to learn to generate 3D feature vectors [6]. Of-course it is important to mention the reason Zhirong Wu et al. created the ModelNet dataset to begin with, Shapenets [35]. Shapenets took on the task to generate volumetric models from depth scans (2.5D), like those produced by commercially available sensors such as Microsoft's Kinnect. With Shapenets, they cleverly made use of CAD models as a ground truth, and simulated depth images by rendering the models.

The recent survey by Ahmed et al. [2], shows a more in-depth picture of the current progress of different deep learning systems on multiple different 3D data representations. It is also pointed out that the fact that 3D learning methods are clearly lagging behind the equivalent methods in 2D space, implies that not all techniques from 2D systems are transferable. This motivates the need for more research within the field of 3D machine learning.

Chapter 4

Methodology

This chapter goes over the important aspects and decisions made when developing and testing the ANNs and pipeline required to process the data-files into the needed formats. Section 4.1 will list some of the tools used to perform these tasks. Section 4.2 will go into detail about the steps of the pre-processing pipeline. In Section 4.3 the generation of the 3D feature decriptors will be explained. After that, the training process and Siamese networks will be discussed in Section 4.5 and 4.6. Then the evaluation tasks will be presented in Section 4.7 and 4.8.

4.1 Programming languages & libraries

The pre-processing and learning systems were developed using Python 3 and C++. Python has in many ways become the de facto language for ML work because of its many and highly optimized libraries. C++ was similarly used for the descriptor generation part of the pipeline as its ubiquitous use in visual Computing give it a similar advantage of library support in that field.

Python 3 libraries:

- **Numpy** [23] is the go-to library for working with multidimensional arrays and matrices in Python, Other libraries often build upon Numpy arrays for cross compatibility.
- **Open3D** [37] is an open-source 3D model library that allows for both high and low level interaction with 3D models and 3D model files. Open3D together with Numpy was used to normalize, sample, rotaionally augment and generate normals for every 3D model.
- **Tensorflow** [1] was used as the core library for training and evaluating ANNs.
- **Keras** [5] was used as a higher level API, running Tensorflow in the background, in order to make constructing, prototyping, saving and loading ANNs easier.

C++ libraries:

- **Eigen 3** [10] is, similarly to what Numpy does for Python, a library for that makes interaction and manipulation of matrices in C++ easier and consistent across multiple libraries.
- **Point Cloud Library** [30] is the main library used on the C++ side of the pipeline. It is an amalgamation of different computer vision algorithms for many different subfields. It also contains API interfaces for generation of multiple different kinds of 3D feature descriptors, which is our main application of this library.

4.2 Data set & Pre-Processing

This thesis made use of Princeton's ModelNet dataset [35]. They offer clean and categorized CAD models in multiple variations. Specifically the 10-class orientation-aligned set was used in this thesis. The dataset contains 10 different classes (categories) of household items and furniture, bathtubs, beds, chairs, desks, dressers, (PC-)monitors, night stands, tables and toilets. The files are delivered in the object file format (*.off*). *.off* is a minimal mesh-based 3D model format stored in ASCII encoding, containing a header, vertex and face information. The simplicity of the format makes it easy to implement a parser for the files, meaning the file-type is supported in several 3D model libraries. This makes the dataset suited for many applications, such as computer vision, computer graphics, robotics and machine learning tasks.

One challenge with the ModelNet dataset is the unbalanced representation of categories, as seen in Figure 4.1. The unbalance would mean that, in tasks such as classification, the learning process and prediction results would be skewed towards the more represented categories. This issue is somewhat mitigated by the choise of network architecture and will be addressed in Section 4.6.

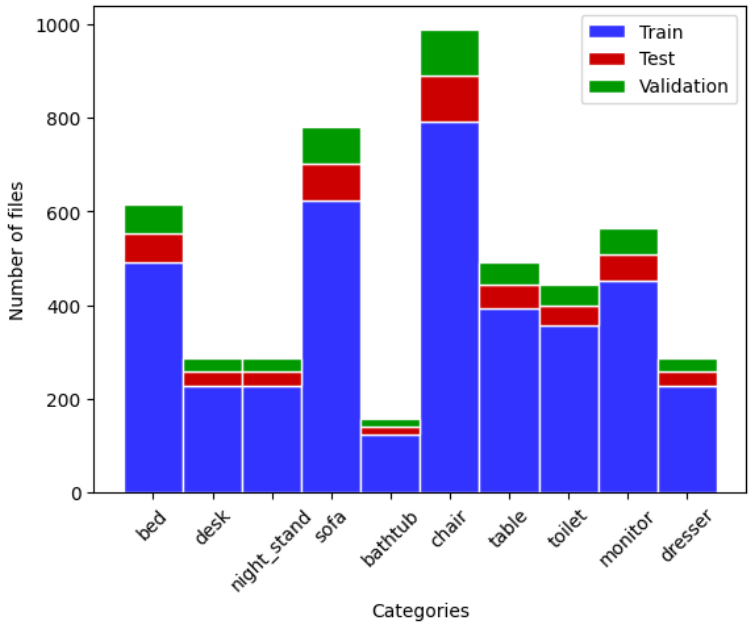
The pre-processsing pipeline consists of the following steps, each step will be discussed in separate sub-sections below:

1. Sampling
2. Normal estimation
3. Keypoint selection
4. Augmentation
5. Dataset partitioning
6. Feature descriptor generation

4.2.1 Sampling

Seeing as the dataset was stored as mesh-files, they need to first be converted into a point clouds for the purposes of calculating 3D feature descriptors. To do this I simply loaded the *.off* files with Open3D into a `TraingleMesh` structure and called the

Figure 4.1 Distribution of classes in the ModelNet10 dataset. The colored bars also display the training, test and validation split.



`.sample_points_uniformly()` method. The method takes the number of sample points as an argument meaning we can accurately control the size of the resulting point cloud.

The number we chose here is significant as it directly affects the size of the input for some of the ANNs that are trained later on. Ideally the point cloud should be as detailed as possible (e.i. as many points as possible). In this case, however, a trade-off between resolution and computation time of both the descriptors and the ANNs needs to be taken into account. Through experimental testing I found that a point cloud of 2000 points proved to be a good compromise, as it is still at a resolution where the models are distinctly discernible while not being too large to train networks and generate descriptors in reasonable time. An example of one of the generated point clouds with 2000 points can be seen in Figure 2.5.

Sampling the mesh to a point cloud reduces the complexity of the model representation. Instead of the models shape explicitly being described by vertices and face-connectivity information, it now described implicitly by the arrangement of points in 3D-space. Another important property this way of sampling gives us is to force the models into a regular size. As talked about in Section 2.2, having a dataset with a regular size makes it significantly easier to feed the data direcly to ANNs. If the dataset was not regular, one would have to make use of pooling techniques or more complex methods like in [9] to feed the network regularized chunks of the model-data.

4.2.2 Normal-Estimation

Originally the PCL API was used to calculate normals during the kd-tree construction for each point cloud. This seemed to work fine while testing the pipeline with individual files. When executing the program on the entire dataset, however, it would occasionally produce invalid results for some models, resulting in *NaN* values. Luckily Open3D also includes a normal estimation feature. This implementation worked flawlessly and was also faster in computation of the normals. It also makes more sense to do the normal estimation at the sampling step as it only needs to be performed once instead of during each descriptor generation and can make use of the face information to estimate the normals for each point more robustly.

The result of this step in the pipeline is the conversion from *.off* files containing vertex and face information to a *.pcd* format containing a point cloud with x,y,z-coordinates and a normal vector for each point.

4.2.3 Keypoint Selection

The selection of good keypoints on 3D models is a complicated field of it's own, and is actively being improved upon [19]. The selection of a keypoint detection algorithm depends on the use-case of the keypoints as well as the features for the dataset. Due to this, a specific keypoint algorithm would favor certain descriptor type. In order to reduce possible variables in the experiments described in Section 4.7, a more naïve method for keypoint selection was performed.

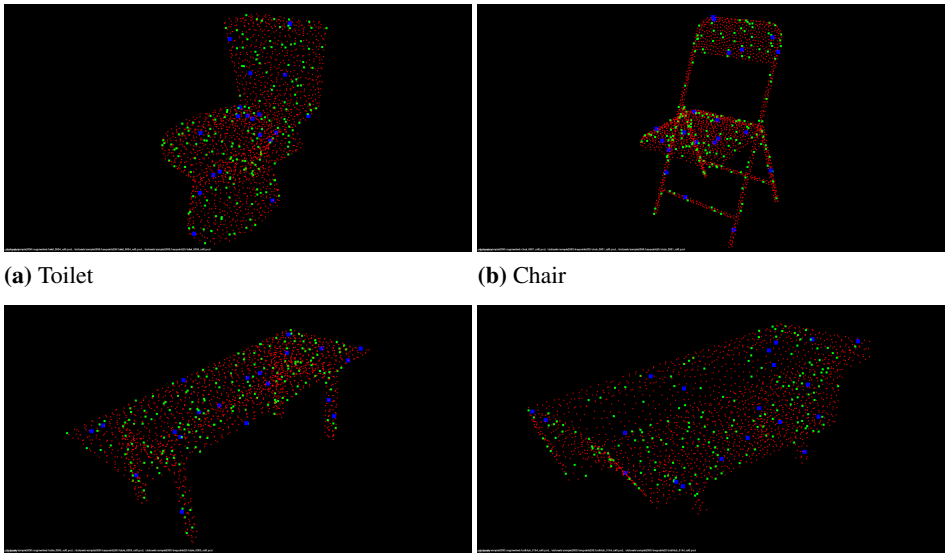
Picking n random uniformly distributed points from the point cloud, would reduce any bias in the selection of keypoints. Luckily, the *.pcd* saving implementation from Open3D stores points in sorted order along the dimension axes. This means we can simply iterate over the point cloud and take every n-th point, while preserving the uniform distribution from the original sampling algorithm. This uniform random selection does, of-course, come at the cost that the picked keypoints are likely never ideal for usage with any of the descriptors. For well defined applications of keypoints and descriptor techniques, random selection would not be a good choice. Picking the right keypoint algorithm for the given task can significantly reduce the number of required keypoints, and in turn, reduce the dimension size of the input-data drastically.

Two keypoint sizes where selected for every model, one of size 20 and another one of size 200. These numbers where chosen as to give a good variance in the magnitude of points as the keypoints, in combination with the full point cloud, would give us 2000-, 200- and 20-point versions of each model. In Figure 4.2 you can see examples of a point cloud (red) and its 20 (blue) and 200 (green) keypoint samples overlaid on the point cloud.

4.2.4 Augmentation

In Section 4.2 It was briefly mentioned that the dataset being used is orientation-aligned. Having the dataset pre-aligned is not really that important to this application. In fact, it would actually be preferable that our network to have the property of orientation invariance. That would mean to learn the 3D model comparisons regardless of the models

Figure 4.2 Showing overlapped picture of the point cloud (red), 200 keypoints (green) and 20 keypoints (blue) for 4 different models



(a) Toilet **(b)** Chair

(c) Table **(d)** Bathtub

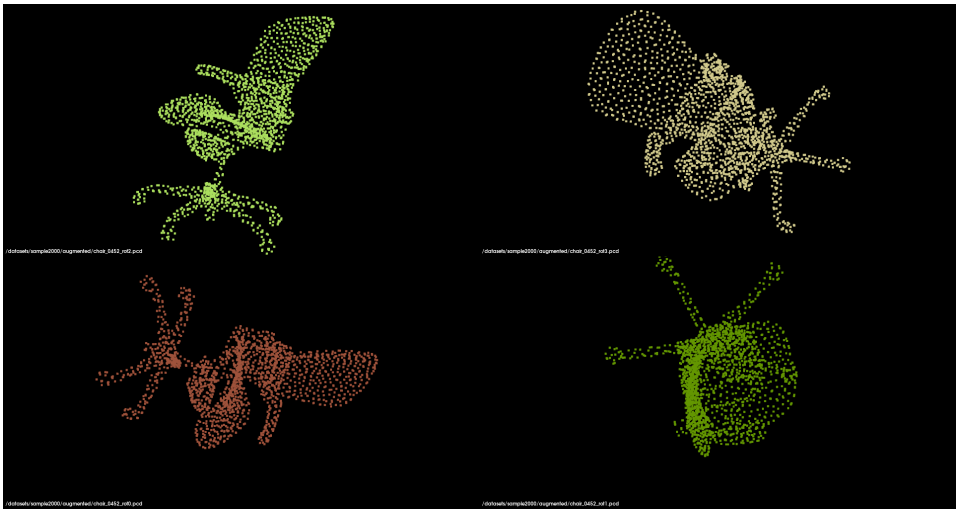
orientation. It would also be ideal to have multiple "samples" of each model, so that the network learns to generalize the model input as best as possible.

Luckily both these issues can be addressed in one step, by applying multiple uniformly random 3d rotations to each model. This will produce multiple instances of the same model, but with different rotations. Rotational augmentation is a well established technique when learning on 2D images and video [25]. The only issue is that defining uniformly random 3D rotations is a bit trickier that doing the same in two dimensional space. In 2D it is intuitive that a rotation would be uniformly distributed as long as the rotation angle is uniformly distributed in the range $\theta = 0, \ldots, 2\pi$. This property does not, unfortunately, carry on to higher dimensions. For more details on this peculiarity and how to implement an algorithm that ensures uniform distribution can be found in *Graphics Gems III* [32]. In Figure 4.3 you can see the result of the application of these random rotation. In this thesis, 4 rotations where applied to each model, effectively multiplying the dataset size by 4.

4.2.5 Partitioning the Data Set

The ModelNet dataset already came split into a training and test set. For the purposes in this thesis, I also needed a validation set to use when evaluating the finished models. The original split also was not proportional to the number of files per class. I therefore decided to merge the original two-way-split and divide the dataset anew into in three parts. For each class 80% of the files where used for training, 10% for validation during training and the last 10% reserved for testing when the training was complete. As the original dataset

Figure 4.3 Displaying the 4 rotation augmentations of single a point cloud model



had high variation in the number of files per class, as can be seen in Figure 4.1, I chose to only set aside a low proportion of each class in order to avoid having too few training-files for some of the classes.

The reason for splitting datasets to begin with is to reduce bias in the model from training with certain datasets. If one does not split the dataset when training the problem of over-fitting will quickly become apparent. Over-fitting would mean the network does not generalize the data but simply learns to commit each training case to memory.

Instead of having the split encoded by putting the different files in different folders, I chose to generate an index file. This mean I could keep all the model files in one folder, simplifying batch file operations when generating descriptors later on. The same index file could also be used for all the different representations (pointcloud, keypoints, SI-, VFH-, FPFH-decriptors) of a given model. This was because the indexation only listed the unique file name, and base file paths and file endings, as well as rotational modifiers for each file, was stored separately from the file name. To make generation and loading of this index file simple it was stored as a JSON-dictionary.

The end result of this process was a partitioning of the dataset that was proportional to the number of files in each category, see Figure 4.1. As well as an index file that could be used when generating training batches working on the file path level. Allowing a more efficient use of system resources by only loading the current batch of model-files into memory, compared to loading the full dataset or dynamically traversing folders.

4.3 Generating 3D Feature Descriptors

Almost all the work of generating 3D feature descriptors is handled by PCL. By leveraging this library, only simple programs calling their descriptor API, had to be written. The general procedure is outlined in pseudo-code 1. When the API returned the result, the

feature vectors were simply written to *.csv*-files as it is a convenient and simplistic format for storing data, especially as the dimensions of the data is known in advance.

I chose to generate three types of descriptors, Spin Images, Fast Point Feature Histograms and Viewpoint Feature Histograms. Originally I wanted to have a more extensive set of feature descriptors, including RICI, USC, SHOT, TriSI, RoPS, etc. However the generation of the descriptors for large datasets, as well as looking for implementations or trying to write implementations for them myself took more time than estimated. I therefore decided to stick to the three mentioned above as they all had implementations in the Point Cloud Library and featured similar interfaces for generation. These three descriptor types also inhabit different properties, SI being rotational invariant, FPFH being a denser representation and VFH containing viewpoint information.

Descriptor parameter optimization would introduce several new variables, complicating later evaluations. For this reason the default descriptor parameters where used for all three descriptor types. This does imply that the generated descriptors are not fine-tuned for the specific dataset used, and it is possible that better results could have been achieved with optimization tweaking. Unfortunately investigating such speculation falls outside of the scope of this thesis.

Algorithm 1 Descriptor Generation

function GENERATEDESCRIPTOR(FilePath f, Keypoints k, Parameters p)
    $cloud, normals \leftarrow pcl :: loadFile(f)$
    $kdtree \leftarrow pcl :: calculateKDTree(cloud)$
    $descriptor = pcl :: Descriptor()$
    $descriptor.setCloud(cloud)$
    $descriptor.setNormals(normals)$
    $descriptor.setSearchIndices(k)$
    $descriptor.setSearchMethod(kdtree)$
    $descriptor.setParameters(p)$
    $result \leftarrow descriptor.compute()$
    $writeToCSVFile(result)$

4.4 Pipeline Summary

Table 4.1 shows the steps of the pipeline as well as some info about the files in each step. After the Augmentation step the pipeline splits in three, producing the three different descriptors in each path. The exact sizes for the different descriptors are from the standard implementation of them in PCL [30]. Spin Images size are dictated by the `image_width` variable. The default value for this is 8 resulting in a histogram size of 153 per index point from the calculations in Equation 4.1. For VFH the descriptor size is a result of the summation of the initial parts. The viewpoint angles are divided into 128 bins, the α, ϕ and θ angles are divided into 45 bins and an additional 45 bins are used for the distance from each point to the centroid. $3 \times 35 + 128 + 45 = 308$, thus result is a VFH histogram size of 308. For FPFH the bin size for α, ϕ, θ is 11, resulting in a histogram size of 33 per

index point.

$$s = (w + 1)(w2 + 1) \tag{4.1}$$

Step	Format	File type	Dimensions
Start: ModelNet10	3D Mesh	*.off*	irregular
Point cloud sampling	PC (point cloud)	*.pcd*	2000x3
Normal-Estimation	PC	*.pcd*	2000x6
Keypoint Selection	1x PC + 2x keypoints	*.pcd*	2000x6 + 200x6 + 20x6
Dataset augmentation	4x PC + 2x keypoints	*.pcd*	4x2000x6 + 200x6 + 20x6
SI generation	array of Spin Images	*.csv*	20x153 and 200x153
VFH generation	single VFH signature	*.csv*	308 and 308
FPFH generation	array of FPFH signatures	*.csv*	20x33 and 200x33

Table 4.1: Overview of format, file-types and dimensions per step in the pipeline

4.5 Training

After the dataset is pre-processed and feature descriptors are generated. The 3D-data can now be fed to artificial neural networks. To keep comparison fair, each type of data will be fed to similar networks, trained and then evaluated. The same core network will be used for all data-types, due to the different input dimensions, however, the first layer will need to be adaptive and adjust to the input. This causes differences in the network sizes.

Within each batch during the training process, a pairwise model selection scheme makes sure to keep positive and negative cases balanced. This is done by selecting two models, A and B. Then every permutation AA, AB, BA, BB of those two models will be added to the batch set.

4.6 Siamese networks

The reason for using Siamese networks is that, as shown by Figure 4.1, the classes of the dataset are imbalanced. Dealing with such imbalance is one of the key advantages of Siamese networks.

Another key reason is stated in Section 2.1.3, the Siamese network functions well for comparison comparison tasks. In a Siamese network, the aggregation layer in the network is essentially trying to learn a distance function with the two feature vectors as input. The distance function that the network will learn is dependant on the label data it is given during training. For example, if the label says that every input-pair with the same base model (they might have different rotations) is marked **True**. Then the model will learn a comparison function, where it tries to discern if the input are different models or the same models. This would effectively make it a rotation-invariant comparing function given a high enough accuracy.

One can easily change the use case of the network. By giving it a label based on if the models are the same class instead of the same model base it can act as a form of classification network. In general, as long as you can clearly define a relation between two inputs you can make a Siamese network learn to differentiate using that relation. It is also possible to learn multiple relations at once if one makes use of a one-hot encoding technique for the output.

What is really interesting about Siamese networks is that once it has learned a relation, you can easily feed it unlabeled data together with a baseline data and let the network label for you. As a tangible example imagine that you train the network to look at two images of cars and output a value indicating if they are the same model or not. Then you might take a large collection of unlabeled car images, and compare each picture to already labeled images. If the result is high it likely means the unlabeled image should have the same label as the baseline image.

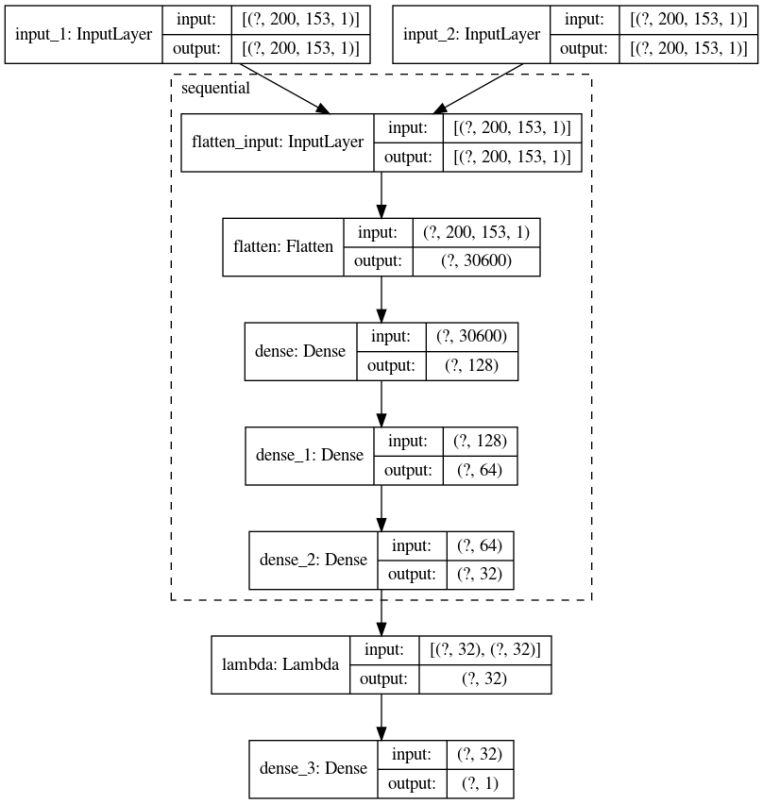
Architecture

Figure 4.4 shows the architectural properties of the networks used in this thesis. The model pictured is for the SI 200 network, only the first layer of the network would change depending on the input however.

The Flattening and Dense layers where used to ensure the dimensions of the following layer would be compatible. Common methods like convolution was not applied as they might prove unfairly beneficial for some representations. Below is the source code to generate the networks:

```
def model(input_shape):
    seq_conv_model = [
        layers.Flatten(),
        layers.Dense(128, activation=activations.relu),
        layers.Dense(64, activation=activations.relu),
        layers.Dense(32, activation=activations.sigmoid)
    ]

    seq_model = tf.keras.Sequential(seq_conv_model)

    input_x1 = layers.Input(shape=input_shape)
    input_x2 = layers.Input(shape=input_shape)

    output_x1 = seq_model(input_x1)
    output_x2 = seq_model(input_x2)

    distance_euclid = layers.Lambda(lambda tensors:
        K.abs(tensors[0] - tensors[1]))([output_x1, output_x2])
    outputs = layers.Dense(1,
        activation=activations.sigmoid)(distance_euclid)
    model = models.Model([input_x1, input_x2], outputs)
    model.compile(loss=keras.losses.binary_crossentropy,
```

Figure 4.4 Realized Siamese model for SI input




```
        optimizer=keras.optimizers.Adam(lr=0.0001), metrics=['accuracy']

    return model
```

4.7 Task 1: Comparison

In this task, we are going to use the model comparison relation briefly mentioned in Section 4.6. To specify; two inputs are considered equal if they stem from the same base model. If two inputs are from two different models, the label will be 0, if they are from the same model (including different rotations) the label will be 1. During this task, accuracy results, training time will be monitored and post-training evaluation will be performed. The training and validation data from the dataset-partitioning mentioned in Section 4.2.5 will be used during the training stage. The testing set will be used during evaluation, after training. The following data-inputs will be used:

- Point cloud (xyz and normal data)

- Keypoints (20)
- Keypoints (200)
- Spin Images (generated at the 20 Keypoints)
- Spin Images (generated at the 200 Keypoints)
- VFH (20)
- VFH (200)
- FPFH (20)
- FPFH (200)

Ten networks will be trained for each of those input-data types, resulting in a total of 90 trained networks. The sizes of the networks will vary slightly as the input-layer size is dependant on the dimensions of the input data. The dimension sizes for each input-data is listed in Table 4.1. The first layer of the network is a *Flattening* layer. This means that after the first layer, all the values of the input are squished into one dimension. From there on the networks are equal across the different input-data. An example of the network architecture can be seen in Figure 4.4. The data collected can be found in Chapter 5.

For the evaluation, each network is trained for a set 200 epochs, this is to keep the comparisons fair between the data-types. The number 200 was chosen from experimental testing, at this point, none of the networks where showing significant improvements.

4.8 Task 2: Classification

Originally I wanted to look at the classification task for 3D shapes, as it is a pretty common task to use for evaluation. During the project this task proved difficult to complete, largely due to complications like a lack of compute power available at home, as I could no longer access the computer lab due to the COVID-19 pandemic. This lead to not being able to finish training and tweaking hyper-parameters to fit each category (descriptor). Which further lead to the networks over-fitting and generally not producing any interesting results. As I ran out of time to gather data for all the data-types I chose to drop this task.

Chapter 5

Results and Analysis

This chapter presents the collected data from Task 1 in Section 4.7 and an analysis of it. In Section 5.1, post-training results will be presented and discussed. In Section 5.3 we will evaluate the trade-offs of the different models in terms of execution speed. Finally Section 5.2 will look at the accuracy growth during the training process.

5.1 Evaluation

Table 5.1 shows 200 evaluations per network for the four categories of model combinations [same model, same class, different model, different rotation]. The abbreviations in the second row of the table stands for True Positive, False Negative, False Positive and True Negative. The True labels list how many many evaluations the network evaluated correctly, the False label is the inverse, or how many evaluations the network got wrong.

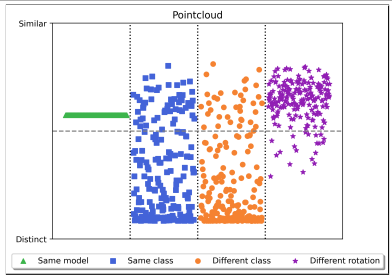
Name	Same model		Same class		Different model		Different rotation	
	TP	FN	FP	**TN**	FP	**TN**	**TP**	FN
point cloud	200	0	55	145	37	163	180	20
Keypoints200	200	0	57	143	33	167	176	24
Keypoints20	200	0	6	194	4	196	25	175
SI200	200	0	13	187	5	195	200	0
SI20	200	0	3	197	2	198	200	0
FPFH200	200	0	2	198	1	199	200	0
FPFH20	200	0	0	200	0	200	200	0
VFH200	200	0	13	187	22	178	200	0
VFH20	200	0	13	187	23	177	199	1

Table 5.1: Predictions for each of the different categories, TP = True Positive, FN = False Negative, TN = True Negative, FP = False positive

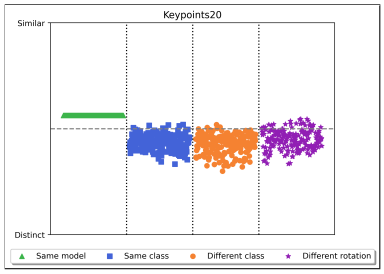
The same data is also visualized in the scatter plots of Figure 5.1 and Figure 5.2. Here

the comparison categories are separated along the x-axis and the y-axis designates the prediction value in the space $(0, 1)$ were a value higher that 0.5 indicates that the network considers the models to be equal, a lower value would be considered different.

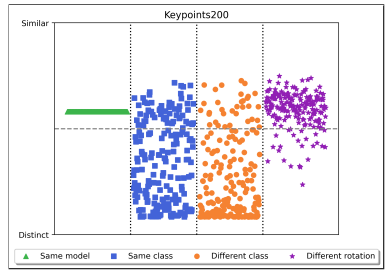
Figure 5.1 Prediction plots, clouds



(a) Full point cloud (2000 points) as input



(b) 20 Keypoints as input



(c) 200 Keypoints as input

Prediction plots of inputs of same model, same class, different model and different rotations.

It is interesting to note that **all** the models got the ”same model”-comparison correct, even the ones that performed poorly overall. This tells us that the ”same model”-relation is easier to learn compared to the other relations. This makes sense as each sub-network in the Siamese network would produce the same feature vector when the input is of the same model. This means the two feature vectors would cancel each other out in the aggregation step. The network therefore only has to learn that when the feature vectors cancel out, the output should be slightly above 0.5. This also explains why this category has no variation in the scatter plots in Figure 5.1 and 5.2. In this case it does not actually matter what the feature vector is, due to the fact that they cancel out.

For the other categories, however, the encoding of the feature vector becomes important. The network now has to learn to extract enough features so that it can differentiate between two different models and two different rotations of the same model. The 20 keypoint model completely fails to do this, likely as 20 keypoints simply does not give enough information to learn good enough feature representation. For the 200 keypoint and the full point cloud networks, the distinction is slightly better. From the plots in Figure 5.1 you can see that the categories with different models are more spread out, with an inclination to gather near the lower end of the graph. Here the ”different rotation”-case also has it’s

center of gravity moved into the correct division. These two networks still struggle with the robustness of the features, as the classification categories have a wide spread, leading to many False Positives and False Negatives.

The descriptor-fed networks looks to be performing significantly better. Several of the plots in Figure 5.2 have well defined lines for the ”different rotation”-category. This can be explained by the descriptors being rotational invariant, like SI, or posses properties which make them more robust in terms of rotation. While there is still some spread with the remaining two categories, they seem to be significantly more weighed towards the bottom of the value space. This means the descriptor networks produce noticeably less False Positive and False Negative predictions.

An important aspect to notice here is that an increased number of keypoints when generating the descriptors, does not seem to improve their ability to properly predict the input. In fact, for both the SI and FPFH, the smaller 20 keypoint variant seem to be performing better than the 200 keypoint version. It is possible that having less keypoints force the networks to learn better feature vectors rather than relying on the input vectors being more unique due to a higher number of keypoints like in the 200 versions. Comparing the 20 Keypoint point cloud result to the 20 Keypoint descriptor results seem to imply that information density is an important factor for getting a good accuracy.

All in all the FPFH descriptor seems to be performing the best, with SI following in a close second place. It is interesting to note here that the FPFH input dimension will be around 4.5 times smaller if generated like in Section 4.3. This further underlines that choosing the choice of descriptor can significantly reduce size of- and time spent training the network.

5.2 Accuracy During Development

It is interesting to look at, not just at the final accuracy of each model, but also their growth during the training process. In Figure 5.3a and Figure 5.4, each training run is displayed with dotted lines. To reduce variance, the results were averaged together, this is displayed as the thicker colored line. Ten networks were trained for each data-type, as mentioned in Section 4.7 Table 5.2 shows the averaged numerical training accuracy at some given intervals. Note that these accuracy numbers are from evaluation during the training, that means they use the validation dataset, not the testing dataset used in Section 5.1.

In Figure 5.3 you can see that the networks struggle to achieve any decent result, even over many epochs the Full point cloud and the 200 keypoint networks only get around an accuracy of 80%. The 20 keypoint networks even struggle to get to a 65% accuracy, suggesting that with 20 keypoints, the input does no longer contain sufficient information to discern many shapes. What is interesting is that all three point based networks have a plateau around the 60-65%. When you compare the curvature it looks like they follow the same growth, but the cloud size determines how fast that growth is undergone.

The Spin Image networks are very quick to get a high percentile, likely as the rotation invariance of the descriptor causes both the ”same model” and ”different rotation” cases to be simple to handle due to the feature vectors being the same. This means the network only has to learn two things. 1) Tune the weights of the aggregation layer such that any non-zero value would produce a value below 0.5 after going through the Sigmoid activation

Type	Epoch 5	Epoch 10	Epoch 20	Epoch 50	Epoch 100	Epoch 200
Full point cloud						
point cloud	62.7832	63.7988	67.6270	78.9258	81.8066	82.2754
200 Keypoints						
Keypoints	55.4102	58.6914	62.6172	63.4473	74.4043	78.8184
SI	99.6387	98.6328	97.8223	97.8418	98.0273	98.2129
VFH	69.0918	75.4785	84.2676	92.2461	94.3457	95.7812
FPFH	76.3379	83.1934	89.8535	**98.1445**	99.6191	99.6777
20 Keypoints						
Keypoints	49.8535	56.6211	60.1367	62.8516	63.0566	64.6582
SI	**99.8926**	**99.8047**	**98.7402**	97.9590	98.2031	98.5059
VFH	64.2969	67.4805	72.7832	86.5039	93.8086	95.2832
FPFH	75.2148	79.0039	84.6777	96.0156	**99.7852**	**99.8145**

Table 5.2: Validation accuracy at set intervals

function. 2) Make the bias values of the activation layer slightly positive so that feature vectors that cancel out would produce a value slightly greater than 0.5 in the activation layer.

FPFH and VFH look to have a more steady growth. This is natural as they actually have to learn good feature vector representations to solve the problem. In the FPFH case the learned feature vectors are so good that they even out-compete the SI network. One reason for this could be because the SI network is more prone to changes as it has a lower fitness requirement to get good results to begin with, although more testing would need to be done to say so conclusively. Another reason could be the local robustness of the FPFH descriptor [29] making it well suited for this task.

A possible reason for VFH performing worse that the other two descriptors could be the viewpoint components. The descriptor is purposefully designed to not be rotational invariant as the use-case it was designed for was computer vision for robots, where stereo/depth information is crucial [28]. In this scenario, however, it means that the network needs to learn to extract the relevant information from the descriptor and ignore the viewpoint sections. This would help to explain why the ”different rotation”-category for VFH is way more spread out compared to FPFH or SI in Figure 5.2.

5.3 Execution Speed

Table 5.3 displays some information about how long each training step for each network took, as well as some information about the dimensions in the first layer of the network. Input Dimensions is the size of the input files, Input total are the dimensions multiplied together resulting in the number of nodes in the first layer.

There is a clear correlation between the training speed and the input size of the network. Since the first layer is fully connected, the increased input size would exponentially increase the total number of weights the network has to train. More weights means more

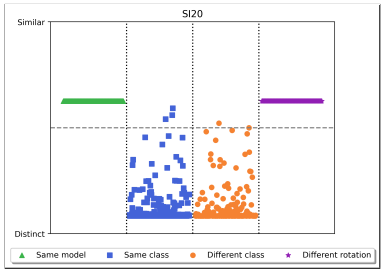
computation, it then follows that the training takes more time. There are also external factors, such as data alignment on the GPU, caching of computations or case specific optimisations in libraries that can affect the performance measured here.

Name	Input Dimensions	Input total	Time per step
Full point cloud			
point cloud	2000x6	12000	56 ms/step
200 Keypoints			
Keypoints	200x6	1200	11 ms/step
FPFH	200x33	6600	149 ms/step
VFH	1x308	308	13 ms/step
SI	200x153	30600	603 ms/step
20 Keypoints			
Keypoints	20x6	120	06 ms/step
FPFH	20x33	660	21 ms/step
VFH	1x308	308	11 ms/step
SI	20x153	3060	75 ms/step

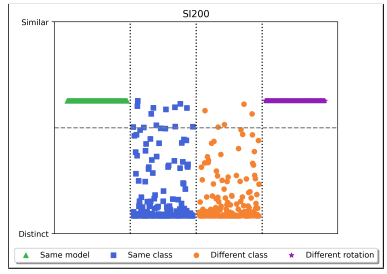
Table 5.3: Training speeds and sizes for the different networks.

When it comes to the calculation time of the system in a broader picture, it is important to be aware of how and when that calculation time is spent. Let us take a manually designed computer vision system and an artifical neural network based computer vision system. The ANN will have a much higher up-front cost, often requiring both a large labeled dataset and a lot of compute-time to train the model initially. One the model is trained however, the network would likely spend less time that the manual system per task it performs. This difference is pretty crucial as many use-cases for task might require the system to work in real-time. In such situations, being able to ”do the work upfront” might be a significant advantage.

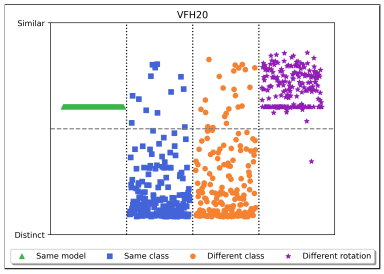
Figure 5.2 Prediction plots of inputs of same model, same class, different model and different rotations.



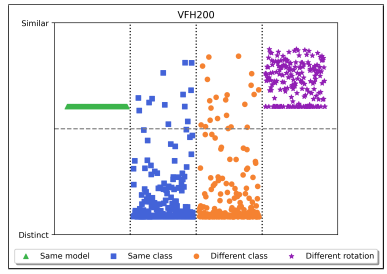
(a) SI (20 keypoints)



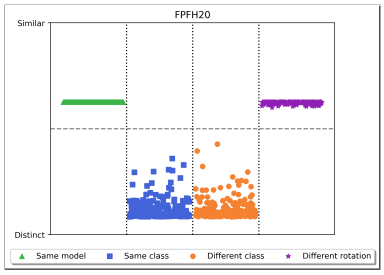
(b) SI (200 keypoints)



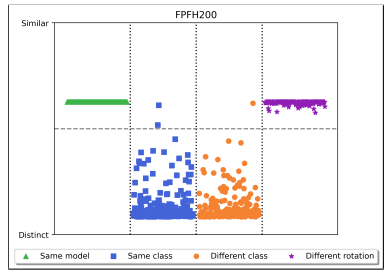
(c) VFH (20 keypoints)



(d) VFH (200 keypoints)

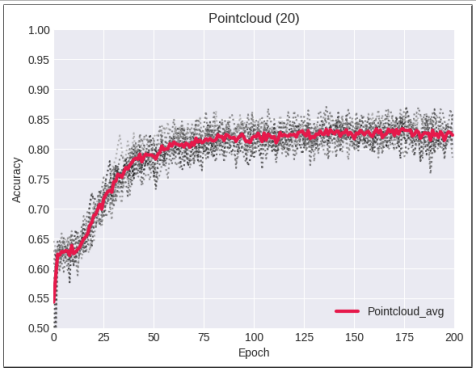


(e) FPFH (20 keypoints)

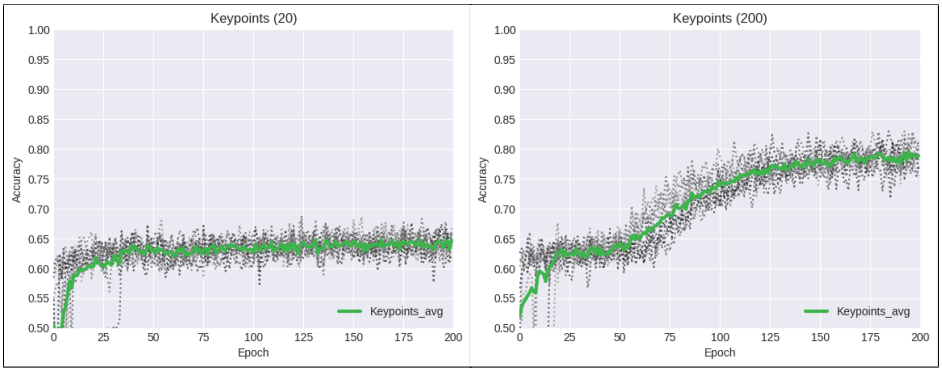


(f) FPFH (200 keypoints)

Figure 5.3 Plots of validation accuracy during training.



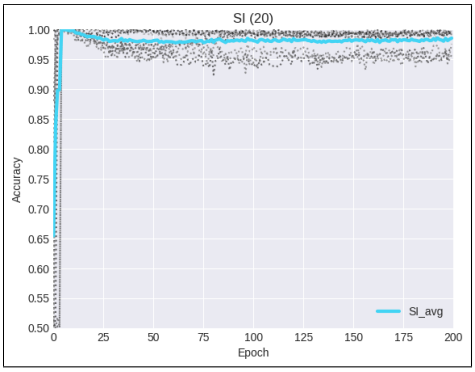
(a) Full point cloud (2000 points) as input



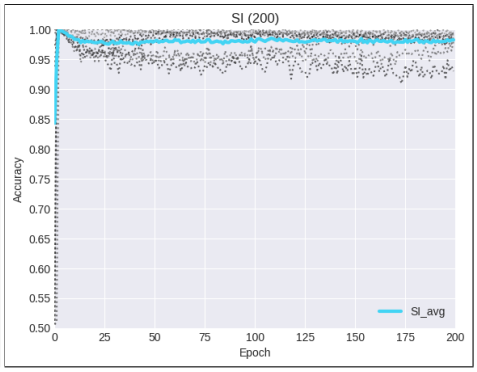
(b) 20 Keypoints as input

(c) 200 Keypoints as input

Figure 5.4 Plots of validation accuracy during training.

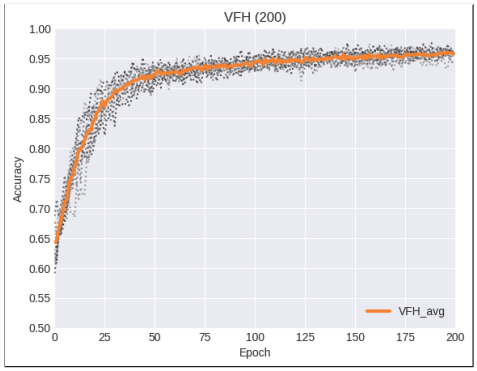


(a) SI (20 keypoints)

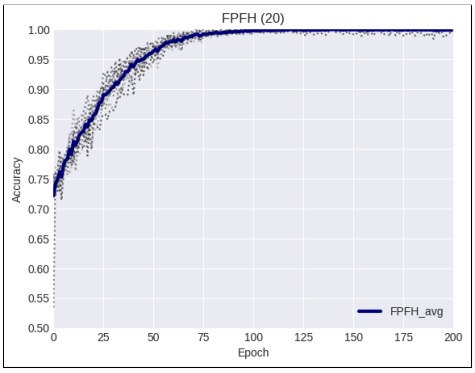


(b) SI (200 keypoints)

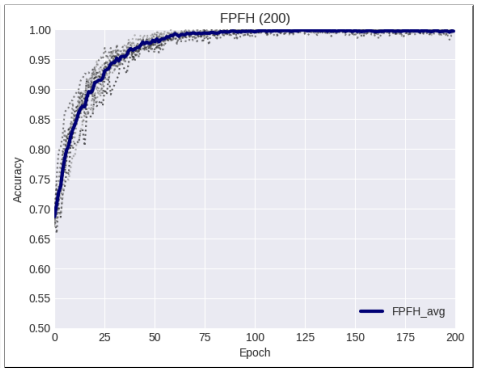
(c) VFH (20 keypoints)



(d) VFH (200 keypoints)



(e) FPFH (20 keypoints)



(f) FPFH (200 keypoints)

Chapter 6

Conclusion

This chapter will seek to answer the research questions stated in Section 1.2, by summarising the results of the work and evaluation done during this thesis.

Research done while exploring the applicability of 3D-shape data to artificial neural networks showed that, direct applications of methods from 2D-space do not necessarily transfer to 3D. Problems like irregularity need to be addressed by either regularizing the input data, or leveraging memory based networks. This thesis seeks to investigate 3D feature descriptors as a form of regularization. Which brings us to RQ1.

RQ1: Would existing 3D feature descriptors be an applicable transformation to regularize 3D-shape data?

The work in this thesis showed that the applicability of the representation depends on the ability to encode the required information, in order to solve the task the network is trained for. 3D feature descriptors are hand crafted representations of 3D shapes, designed to be more information dense than the raw shape representations. Experimental results confirm this by showing that 3D feature descriptors can transform the 3D-shape data to a size-regular representation while maintaining the 3D-shape information required to compare and discern between similar and different models, successfully.

RQ2: How does the descriptor method compare to other methods, such as point cloud trimming, in tasks such as model comparison?

In this thesis made use of multiple feature descriptors, SI, VFH, FPFH. These descriptors, in addition to different resolutions of point clouds, were fed to Siamese neural networks and trained to perform direct model comparison. Evaluation showed that one can achieve both faster training and more accurate results by using feature descriptors over trimming point clouds. These statements are backed up by experimental results and evaluations done in Section 5.1 of this thesis. Further more, results show that the descriptors, unlike point clouds, do not require a large dimensions to produce networks with high

accuracy. Hence the descriptor networks are not only more accurate, but also smaller and therefore faster to train and use for prediction.

6.1 Contribution

This thesis main contribution is a novel application of 3D feature descriptors in relation to artificial neural networks. The work done here has shown that 3D feature descriptors can be an effective way to regularize 3D-data. This further demonstrates the versatility of 3D feature descriptors and the groundwork Johnson [15] and Rusu et al. [27] has laid down. In doing so this work has contributed to exploring the field of geometric deep learning at the intersection of machine learning and computer vision as defined by Bronstein et al. [4]. It is important to note that this thesis is not an extensive evaluation of descriptor based networks, but a proof of concept that the combination of descriptors and ANNs can provide well functioning solutions.

6.2 Further Work

- This thesis focused on three descriptor types, Spin Image, Viewpoint Feature Histograms, and Fast Point Feature Histograms, and a type of ANNs, Siamese networks. It therefore natural to seek to expand on this research by looking at **other types of networks** or doing extensive evaluations of **other descriptors**. Looking at **other tasks**, such as classification, retrieval, registration and synthesis could also be a direction for further work.
- Looking into how the **trade-off in computation time** between point clouds and descriptors grows when when training networks for higher resolution models could be an interesting direction.
- It would be interesting to investigate the upper limits of what these networks are capable of by looking into **optimizing the hyper-parameters** of the geometric neural networks presented in this paper. It would also be interesting to look into **optimization of the the descriptor parameters** in the generation algorithms in relation to the dataset.
- An interesting expansion of the techniques presented here would be to see how well one Siamese network could handle **multiple relations** at once by introducing **one hot encoding** into the label.
- Looking into **ANN based keypoint selection** system could prove beneficial, not just for the methods presented in this thesis. But the field of computer vision in general.
- It would be interesting to **compare 3D-data networks classification results those done on 2D-images**, as it could be a good indicator of how well the 3D techniques have become.

Bibliography

[1] Abadi, M., A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng
2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[2] Ahmed, E., A. Saint, A. E. R. Shabayek, K. Cherenkova, R. Das, G. Gusev, D. Aouada, and B. Ottersten
2018. A survey on deep learning advances on different 3d data representations.

[3] Brock, A., T. Lim, J. M. Ritchie, and N. Weston
2016. Generative and discriminative voxel modeling with convolutional neural networks. *CoRR*, abs/1608.04236.

[4] Bronstein, M. M., J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst
2017. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):1842.

[5] Chollet, F. et al.
2015. Keras. `https://keras.io`.

[6] Deng, H., T. Birdal, and S. Ilic
2018. Ppfnet: Global context aware local features for robust 3d point matching.

[7] Elseberg, J., S. Magnenat, R. Siegwart, and A. Nuchter
2012. Comparison on nearest-neigbour-search strategies and implementations for efficient shape registration. *Journal of Software Engineering for Robotics (JOSER)*, 3:2–12.

[8] Farley, B. and W. Clark
1954. Simulation of self-organizing systems by digital computer. *Transactions of the IRE Professional Group on Information Theory*, 4(4):76–84.

[9] Feng, Y., Y. Feng, H. You, X. Zhao, and Y. Gao
2018. Meshnet: Mesh neural network for 3d shape representation.

[10] Guennebaud, G., B. Jacob, et al.
2010. Eigen v3. http://eigen.tuxfamily.org.

[11] Hang Su, Subhransu Maji, E. K. and E. Learned-Miller
2015. Multi-view convolutional neural networks for 3d shape recognition.

[12] He, K., X. Zhang, S. Ren, and J. Sun
2015. Deep residual learning for image recognition.

[13] Hebb, D.
2005. *The Organization of Behavior: A Neuropsychological Theory*. Taylor & Francis.

[14] Isha Salian, N.
2018. Supervize me: Whats the difference between supervised, unsupervised, semi-supervised and reinforcement learning? *https://blogs.nvidia.com/*.

[15] Johnson, A. E.
1997. Spin-images: A representation for 3-d surface matching.

[16] Johnson, A. E. and M. Hebert
1998. Surface matching for object recognition in complex three-dimensional scenes. *Image Vis. Comput.*, 16:635–651.

[17] Karmakar, N., A. Biswas, P. Bhowmick, and B. Bhattacharya
2011. Construction of 3d orthogonal cover of a digital object. Pp. 70–83.

[18] Krizhevsky, A., I. Sutskever, and G. Hinton
2012. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25.

[19] Lin, X., C. Zhu, and Y. Liu
2016. Mesh interest point detection based on geometric measures and sparse refinement.

[20] Markas, T. and J. Reif
1992. Quad tree structures for image compression applications. *Information Processing and Management*, 28(6):707 – 721. Special Issue: Data compression for images and texts.

[21] Michael Copeland, N.
2016. Whats the difference between artificial intelligence, machine learning and deep learning? *https://blogs.nvidia.com/*.

[22] Nair, V. and G. E. Hinton
2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, J. Frnkranz and T. Joachims, eds., Pp. 807–814.

[23] Oliphant, T. E.
2006. *A guide to NumPy*, volume 1. Trelgol Publishing USA.

[24] Park, H., S. Lim, J. Trinder, and R. Turner
2010. Voxel-based volume modelling of individual trees using terrestrial laser scanners.

[25] Perez, L. and J. Wang
2017. The effectiveness of data augmentation in image classification using deep learning.

[26] Rosenblatt, F.
1958. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, Pp. 65–386.

[27] Rusu, R., N. Blodow, Z. Marton, and M. Beetz
2008. Aligning point cloud views using persistent feature histograms. Pp. 3384–3391.

[28] Rusu, R., G. Bradski, R. Thibaux, and J. Hsu
2010. Fast 3d recognition and pose using the viewpoint feature histogram. Pp. 2155–2162.

[29] Rusu, R. B., N. Blodow, and M. Beetz
2009a. Fast point feature histograms (fpfh) for 3d registration. In *2009 IEEE International Conference on Robotics and Automation*, Pp. 3212–3217.

[30] Rusu, R. B. and S. Cousins
2011. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China.

[31] Rusu, R. B., A. Holzbach, M. Beetz, and G. Bradski
2009b. Detecting and segmenting objects for mobile manipulation. In *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops*, Pp. 47–54.

[32] Shoemake, K.
1992. Iii.6 - uniform random rotations. In *Graphics Gems III (IBM Version)*, D. KIRK, ed., Pp. 124 – 132. San Francisco: Morgan Kaufmann.

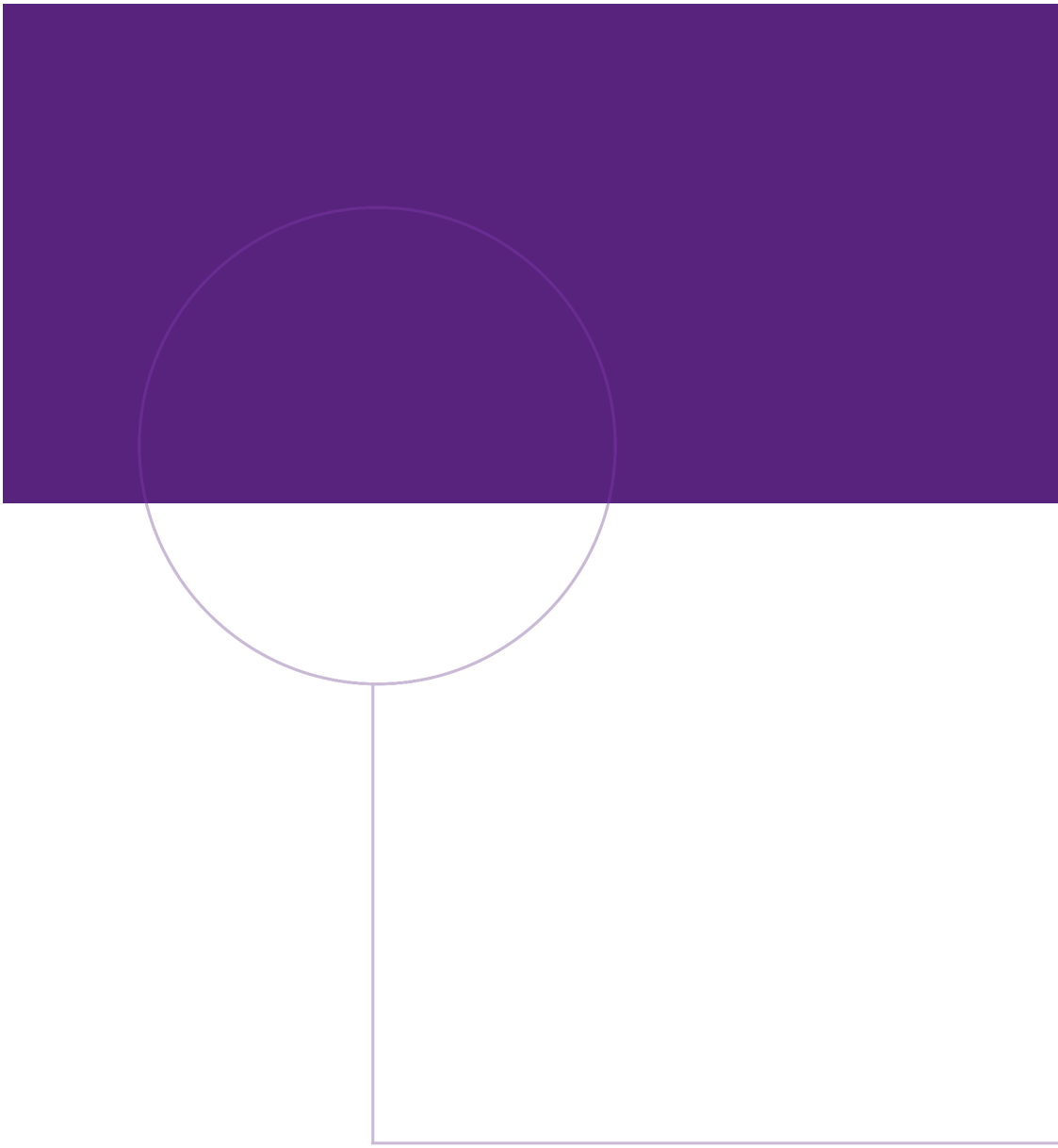
[33] TAFASCA, S.
2019. Multi-view image classification.

[34] Wu, Z., S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao
2014. 3d shapenets: A deep representation for volumetric shapes.

[35] Zhirong Wu, S. Song, A. Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and J. Xiao
2015. 3d shapenets: A deep representation for volumetric shapes.

[36] Zhou, J., H. Huang, B. Liu, and X. Liu
2019. Normal estimation for 3d point clouds via local plane constraint and multi-scale selection.

[37] Zhou, Q.-Y., J. Park, and V. Koltun
2018. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*.



NTNU

Norwegian University of
Science and Technology