# 3D Rotation Invariance in Deep Neural Networks for Point Cloud Processing

Kuba Perlin

Oxford Robotics Institute, Trinity Term 2020

Supervisors: Dr Maurice Fallon, Prof. Alex Rogers

A Thesis Submitted for the Degree of *Master of Science in Computer Science*

# Abstract

In recent years, several different approaches have been developed, to obtain deep point cloud processing models that are robust or invariant to 3D rotations. This area of study is still evolving very rapidly and there is no consensus on the best approach. In this work, we focus on two independent subjects in the area.

Firstly, we compare different methods of using data augmentation to encourage rotation robustness in training. Related research works usually either use a single method of augmentation, without many remarks, or set a goal of avoiding data augmentation altogether. We design three easy-to-understand benchmarks of rotation robustness and use them to quantitatively compare multiple augmentation techniques. Interesting empirical observations about the benchmark scores are presented as conclusion.

Secondly, we investigate an existing rotation-robust architecture, the Spherical Fractal Convolutional Neural Network, in detail. We analyse its theoretical and empirical properties and propose a number of modifications to the model, that are evaluated on a standard object classification dataset. We demonstrate that our modifications yield a small improvement on established classification benchmarks.

---

Approximate dissertation word count: 17154.

# Acknowledgements

Thank you to my primary project supervisor, Dr Maurice Fallon, for providing me with an opportunity to work on this project, providing computational resources, and making me a proper member of his Dynamic Robot Systems research group – that gave me invaluable insight into his field of work, and first-hand impression of what daily work of DPhil students is like. I have also benefitted from Maurice's oversight of this project and guidance on writing this dissertation.

I am equally grateful to Georgi Tinchev, a DPhil student from the Dynamic Robot Systems group, for all the advice, wisdom, and encouragement provided, as he joined Maurice in supervising my project. Weekly meetings with Georgi were key in helping me use my skills most efficiently.

As I am about to finish my Master's degree at Oxford, and move on to the next exciting research projects, I am very aware that I owe these opportunities to many great teachers that have taught me along the way. As a small symbol of gratitude and appreciation for the help I can never repay them for, I include special thanks to my most influential teachers over the years:

**Magda and Piotr Perlin** – *my parents, who have always supported me and my scientific interests, skillfully and unconditionally*

**p. Czarek Rybak** – *my primary school computer science & chess teacher, who made me enjoy school in so many different ways*

**p. Beata Ordakowska** – *my primary maths teacher, who went above and beyond her responsibilities, providing me with individually tailored maths classes, which gave me an opportunity to excel at early age*

**dr Wojciech Guzicki** – *my middle school maths teacher, who taught me what reasoning really is*

**p. Agnieszka Świtalska** – *my middle school ethics teacher, who laid foundations for my never-ending questioning of the world*

**dr Jerzy Konarski** – *my high school analysis teacher, who helped me solidify my appreciation of mathematical rigour*

**dr John Fawcett** – *my undergraduate Director of Studies, who guided my transition into the world of adult science, better than anyone else could*

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 3D Object Classification

Grouping objects into meaningful categories is one way in which humans abstractly think about the world. The modern-era classification task asks to assign observations (inputs) to the correct category, where 'correct' is usually impossible to define succinctly and depends on notions intuitively understood by humans, such as 'are those cells cancerous' or 'is there a cat in this picture'.

3D object classification, the topic of this work, is one of the most tangible settings for the classification task – it pertains to material things with well-defined shapes, which carry enough information to put the object into one of the available categories.

A well established example is a self driving car's task to understand whether what it has spotted on the side of the road is a tree, a pedestrian, or perhaps a deer. Increasing the challenge, the 'artificial general intelligence' of the future, as envisioned by most, would have the capability to look at any object, not limited to a small number of classes, and describe it in a meaningful way.

A similar task, though different in essence, is recognising a single object, unchanging in its identity, given multiple different observations of it. This is key in modern mobile robotic

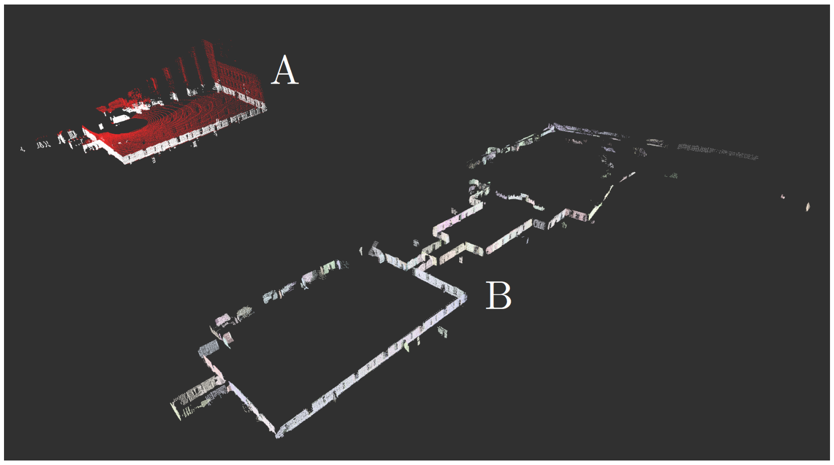


Figure 1.1: A robot using a 3D scan of its surroundings to localise itself. A – in red: a recent LiDAR scan of the robot's surroundings; in white: a height-wise truncated version. B – a premade map that the robot needs to place itself in. The visualisation features the Newer College Dataset [1] and was made using RViz software [2].

navigation. Upon recognising a familiar landmark, a robot should be able to tell its current location – that might be its main way of navigation, or it might be a useful resource in correcting errors of other localisation methods, such as odometry, or GPS. A mobile robot's localisation attempt is illustrated in Fig. 1.1.

As is the case in numerous other fields of broadly understood artificial intelligence, modern state-of-the-art approaches to 3D object classification also involve *deep learning*. Thus this project entirely focuses on (supervised) deep learning methods.

## 1.2 Rotation Invariance

Consider a robot of the future walking around a parking lot and using its LiDAR sensor to scan surrounding objects. It obtains two laser scans of two similar cars, saved in the robot's main coordinate frame. It so happens that in one of the scans, the car's heading is aligned with the robot's $x$-axis, but in the other scan, it is aligned with the robot's $y$-axis.

The robot ends up with two scans of very similar objects, but the two datasets it has collected are quite different – they are only similar up to a rotation. As the makers of the robot, we would want its object classifier to produce the same output – '*it is a car*' – for both scans.

The orientation of the scans is an arbitrary variable not related to the nature of the task. A car is a car no matter which way the owner parked it, or what angle the robot is looking at it from.

The rule that, however we rotate the input, we would want the output to be unchanged, is referred to as *rotation invariance* of the 3D object classification task. This is an example of *domain knowledge* – some abstract property of the problem that we are lucky enough to not only understand abstractly, but also formulate mathematically, and therefore use. Not all tasks concerning 3D objects are rotation invariant (e.g. determining whether a fish is alive).

While a perfect classifier should, by definition, classify both observations as cars, there is no such thing as a perfect classifier, and in practice we may try to use our domain knowledge to aid our algorithms, as opposed to feeding raw data into generic multi-purpose learning algorithms, agnostic to whether they are consuming 3d data, word encodings, or discretised audio signals.

The multi-purpose algorithm certainly has its merits – if two algorithms perform equally well in all other aspects, then we would prefer the one, which requires less human intervention to adapt to new problem domains. However, when interested in optimising performance in a specific, frequently encountered task, it is natural to use all the knowledge available to aid the algorithm.

Abstract domain knowledge of similar nature has in the past helped motivate the use of many ground-breaking algorithms, such as the Convolutional Neural Networks (CNNs) [3] used with great success for image analysis, and based on the understanding that spatial locality has an important role in how humans interpret information from images.

## 1.3 Aims of the Project

The goal of our project is to investigate, qualitatively and quantitatively, methods of achieving invariance or robustness to rotations in the setting of 3D object classification, in modern, neural-network-based classifiers.

We study two very different ideas in detail and present our findings. The first one is data augmentation – an extra step that can be applied to any supervised learning pipeline, which involves modifying the training data that the model receives. We compare the effects of different ways of augmenting the data, working with a point cloud architecture called PointCNN [4].

The second subject of study is a particular deep neural network architecture, the Spherical Fractal Convolutional Neural Network (SFCNN) [5], which, by construction, encourages robustness to rotations. We analyse its performance and reactions to input rotations, as well as proposing, defining, and evaluating original modifications to the model.

## 1.4 Structure of this Dissertation

Chapter 1 has introduced the project and its problem domain.

Chapter 2 contains the necessary project-specific background, assuming rudimentary familiarity with machine learning. It introduces 3D object representations, the 3D point cloud classification and description tasks, and the concepts of rotation invariance and data augmentation. It also contains a brief introduction of the Spherical Fractal Convolutional Neural Network model.

Chapter 3 presents existing deep architectures for 3D point cloud processing, introducing fundamental architectures of the field, as well as the most recent ideas geared towards obtaining invariance or robustness to rotations.

Chapter 4 presents our investigation of different data augmentation methods for achieving rotation robustness, on the example of a particular deep model (PointCNN). The experimental results are presented at the end of that chapter.

Chapter 5 contains an in-depth presentation, analysis, and discussion of the SFCNN architecture, and our proposed modifications.

Chapter 6 presents the experiments performed with the SFCNN and our proposed modifications, and their results.

Chapter 7 contains a brief summary of the project and outlines potential avenues for future work.

# Chapter 2

# Background

## 2.1 3D Object Representations

To propose working scientific solutions to the 3D object classification problem, one must have concrete representations to work with. Multiple alternatives have been conceived and utilised in the last decades. While our work uses point clouds as inputs, related research has been done using other representations, which we describe in this chapter.

For the purposes of this project, it is not essential to have a rigorous, generic, definition of a 3D object, because we focus on the point cloud representation. It suffices to consider 3D objects to be 'sensible' closed subsets of the Euclidean space $\mathbb{R}^3$.

### 2.1.1 Voxel Representations

A *voxel grid* is defined to be a regular grid, partitioning all of $\mathbb{R}^3$ into axis-aligned cubes (*voxels*) of a fixed size.

Given a voxel grid, a 3D object can be represented as a mapping from voxels to values, such as booleans (e.g. is the voxel occupied or not) or reals (e.g. what portion of the voxel is occupied).

Finer voxel grids (with smaller voxels) preserve more information about the object's shape but,

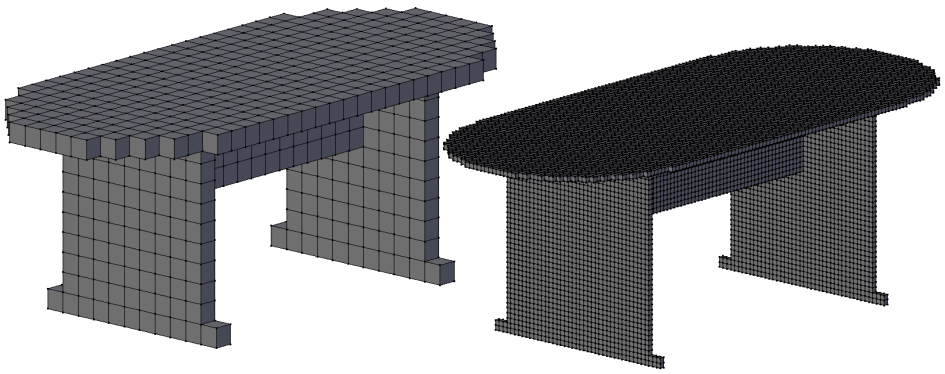


Figure 2.1: Two voxelisations of a 3D model at different levels of detail. The model pictured is `table_0394` from the ModelNet40 dataset [10].

naturally, also take up more memory. An illustration of the same 3D model being represented in voxel grids of different densities is shown in Fig. 2.1. A common extension of the voxel grid idea is the *octree* data structure [6], in which select cubes may be subdivided further than others, in order for the grid's resolution to cost-efficiently adapt to the local complexity of the data.

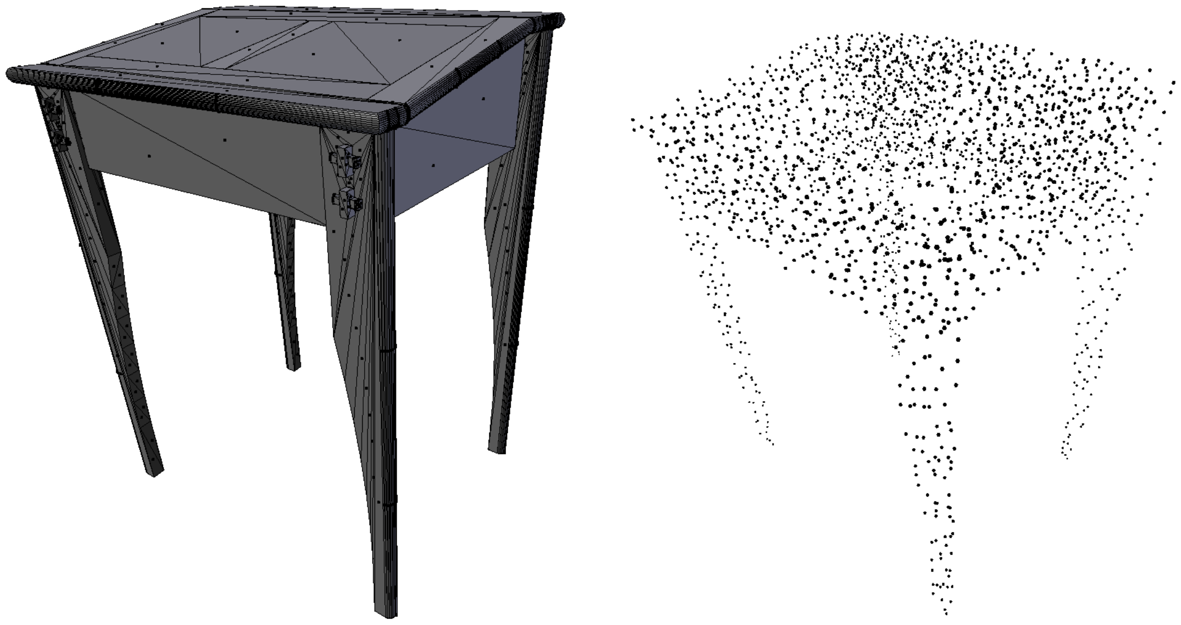
Due to their regular structure, voxel representations lend themselves easily to convolution-based methods [7]. Voxels also find use in 3D data visualisation, used in diverse fields such as medical imaging or computational fluid dynamics [8]. One common visualisation method is the *marching cubes* algorithm [9], which converts volumetric data into easy-to-render polygonal surfaces.

### 2.1.2 Surface Meshes

The key idea behind this representation is to define a 3D object by its boundary. Approximating the object's boundary with a polygonal mesh yields the *surface mesh* representation. An example is shown in Fig. 2.2a.

A finer mesh (with more faces) can approximate a complex 3D shape better, but takes more memory to store.

A commonly used digital representation of meshes, employed e.g. in the computational geometry C++ library CGAL [11], is the half-edge data structure [12], which stores vertices, edges,



(a) The mesh from the original ModelNet40 dataset.

(b) The 2048-point point cloud representation that we work with in this project

Figure 2.2: An example object from the ModelNet40 dataset [10] – `night_stand_001`.

and faces of a mesh, together with the information necessary to traverse and manipulate the mesh in meaningful ways, e.g. ‘iterate over neighbours of a vertex in clockwise order’ (‘clockwise’ being distinguished from ‘counter-clockwise’ by using the inside/outside distinction as reference).

Surface meshes are a common choice nowadays for 3D rendering, be it for scientific shape visualisation, the film industry, or video games. Colour textures and other surface properties can be stored easily for mesh models.

Surface meshes can be extended to volumetric meshes, such as tetrahedral or hexahedral meshes. Those have found use e.g. in materials engineering [13], for modelling tensions inside a physical solid.

### 2.1.3 Point Clouds

A *point cloud* is defined to be an unordered set of points $(x, y, z)$ in $\mathbb{R}^3$. It is the least structured representation amongst the ones mentioned in this section, as it contains no topological information, such as voxel adjacency, or mesh face adjacency present in the other representations.

If all points in a point cloud belong to the boundary of a certain 3D object, we call that cloud a representation of that object. This definition is motivated by the 3D object classification task – point clouds of real-life objects are usually obtained with scanners that detect the objects' surfaces. A point cloud representing a nightstand is shown in Fig. 2.2b.

One could also consider point clouds including points from the interior of the object, but it is important to make the distinction, and in this work, we fix our focus on the boundary-type representations.

Point clouds are commonly obtained using laser range finders. LiDAR (a portmanteau of 'light' and 'radar') sensors work by projecting laser beams into the environment and measuring the distance to the nearest objects along the beam trajectories, by analysing the returning reflected beams. Knowing the direction a beam was fired along, and the distance at which the collision occurred, a point in space can be inferred where a surface must have resided at the time of the measurement. Repeating the process rapidly for many different directions yields a point cloud scan of the surroundings.

LiDAR scanners are often used by mobile robots as part of their perception systems, possibly alongside video cameras or other sensors. The information sensed by LiDAR and video cameras is different and the two systems can be used to complement each other. However, successful LiDAR-only robot localisation has also been achieved [14], [15], [16].

LiDAR scanners can also be used to create point cloud scans of environments for other purposes, such as construction work [17] or archaeology [18].

Clouds produced by LiDAR scanners often have some structure, an example shown in Fig. 2.3, which shows a LiDAR scan of an urban environment, composed of a series of line scans, each comprised of measurements taken along a vertical plane. This extra domain knowledge could be incorporated into computer vision solutions designed to consume LiDAR data, but for the purposes of this work we do not make any assumptions about origins of the clouds. In

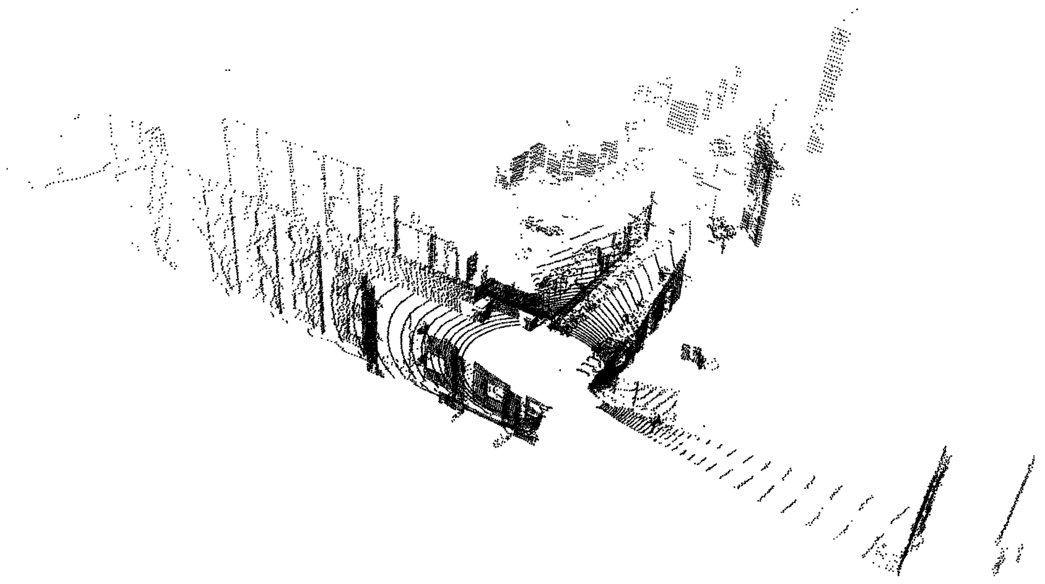


Figure 2.3: A point cloud obtained from a single LiDAR scan, taken from the Mathematics Institute dataset (see Section 4.1.2). The measurement was taken in between urban buildings.

fact, clouds in the ModelNet40 dataset we work with (introduced in Section 4.3) have been obtained artificially by sampling surface meshes.

### 2.1.4 Converting between representations

As mentioned previously, point clouds can be obtained from meshes by sampling their surfaces. One canonical choice of the sampling method would be area-uniform sampling. That is how the conversion illustrated in Fig. 2.2 was done.

Voxel grids can also be sampled to create point clouds. If one is interested in boundary-type point clouds, then before sampling, one has to solve the problem of determining which voxels belong to the boundary.

Point cloud data and surface meshes can both be discretised into voxels straightforwardly. Voxels can be converted into surface meshes, the usual approach being the aforementioned marching cubes algorithm [9].

Point cloud data can also be usefully aggregated into 2D images. One possible approach – *Bird's Eye View* – is to store height and density of a point cloud representation of terrain at each pixel of the ‘map’ image [19].

All approaches, that rely on quantising point clouds, require very high-resolution clouds in practise, which are not always available. Another common problem is handling the holes (gaps) in the quantised data, which arise from non-uniformity of the clouds.

## 2.2 Classification and Description

Classification and description are two related tasks, between which we like to distinguish for the purposes of this project. While classification demands that an input be mapped to one of predefined classes, description is a more general task, allowing for continuous output. The distinction is made more rigorous below.

### 2.2.1 Classification

*Classification* is a standard setting, which can be defined abstractly as follows:

- A set of inputs $X$.
- A (countable) set of classes $C$.
- Classifiers $f : X \to C$.

A famous example of a classification problem is the MNIST handwritten digits dataset [20], where inputs – $28 \times 28$ greyscale images – need to be classified as one of the 10 decimal digits.

### 2.2.2 Metric Space Description

Consider the robot localisation scenario first introduced in Section 1.1. A robot keeps moving around the world and it observes numerous objects in its environment. Its aim is to recognise objects it has seen before (for the purposes of determining its location in the world – i.e. localisation).

Having observed an object, the robot needs to map it to either one of previously seen objects, or a special *never seen this before* response. While one could technically frame it as a classification problem in different ways, there is another, more natural framing for this task, utilising the elementary mathematical concept of a *metric space*[1]. That is the *description task*:

- A set of inputs $X$.
- A metric space $\mathcal{D}$ of descriptions.
- Descriptors $f : X \to \mathcal{D}$.
- $\mathcal{D}$'s distance function $d : D \times D \to \mathbb{R}$.

The aim of description is to extract useful features that can be matched to features of previously seen objects. The robot, equipped with a descriptor $f$, would *describe* the newly observed object $x$, obtaining a description $f(x)$, and then make its decision based on distances between $f(x)$ and descriptions of previously seen objects. The assumption of the robot would be, informally, that similar objects have similar descriptions.

No simple answer exists to how to deal with those descriptions. While it is obvious which of the previously seen objects is most similar to the new one – being the one with least distant description – there is no ultimate way of deciding when to return the *never seen this before* response. That is usually decided using heuristics.

**Remark.** *To link classification and description, it is useful to think about classification, as implemented by most modern deep networks, as a description, followed by a block of fully-connected layers and an argmax.*

## 2.3 3D Point Cloud Description Methods

We have defined 3D point clouds as (unordered) sets of points in $\mathbb{R}^3$. Several design choices have to be made for any point cloud descriptor (be it based on neural networks or not):

[1]Informally, a *metric space* is a set of points equipped with a distance function that maps pairs of points to the distances between them. The distance function must satisfy some 'intuitive' assumptions, for the space to be a metric space – nonnegativity, symmetry, the 'triangle inequality', and returning 0 if and only if the two points are equal.

1. Whether to normalise the input cloud (to have mean 0 and variance 1) – this is dataset-dependent. Absolute size may convey useful information, which could occur if we have guarantees about sensor position relative to the object.

   In methods where normalisation of the input is essential, such as deep learning, one could process the normalised cloud, together with the original scale and mean inserted as separate inputs.

2. Whether to allow for an arbitrary number of points – some descriptor designs might require a fixed input size, e.g. a $256 \times 3$ array of `floats`, for 256-point clouds. In those cases, a bigger input cloud would usually be subsampled before being passed to such a descriptor.

3. Whether and how to utilise any features that might come associated with the points. Common such features include: intensity of the return LiDAR signal, surface normals, or RGB colours of the surface at the point.

### 2.3.1 Classical Methods

Point cloud descriptors that do not involve neural networks have been devised and used successfully. We refer to those as *classical* methods. Those can be looked at as geometric metrics, or 'handcrafted features' in contrast with the machine learning approaches. Well known classical descriptors used in robot localisation include:

- *Eigenvalue-based features.* Geometrically meaningful measures based on eigenvalues of the $3 \times 3$ covariance matrix of the point cloud. [21]

- *Shape histograms.* Ten histograms of various statistics, such as distances between randomly selected point pairs, were used in a successful robot localisation algorithm SegMatch [22], together with the eigenvalue-based features.

- The *3D Gestalt Descriptor,* based on aggregating point data in polar-coordinate bins around selected key-points [23]. The Gestalt Descriptor was used successfully in mapping of an underground mine terrain [24]. One of its key features is invariance to cloud density.

### 2.3.2 Deep Learning Methods

In recent years, many state of the art results have been achieved in point cloud tasks, by neural-network-based approaches, starting with the simple, foundational PointNet architecture [25], which has since served as a baseline for more advanced methods.

A key architectural requirement is order-invariance, imposed by the definition of point-clouds as *unordered* sets of points. Neural networks naturally receive inputs in ordered data structures (vectors, tensors) and thus order-invariance needs to be explicitly engineered.[2]

A literature review, covering both the PointNet and a multitude of more recent deep point cloud descriptors, is presented in detail in Chapter 3.

## 2.4 Rotation Invariance in Point Clouds Tasks

### 2.4.1 Motivation

Consider two LiDAR scans of the same surface, taken by a mobile robot at two occasions, from different viewpoints. (For now, we assume that the entire surface is visible both times.) The goal of the robot would be to recognise the surface as one it has previously seen before, in order to help the robot determine its current location. However, the two scans are not trivial to make an association between. In this Section, we first consider how the two scans may differ, and then describe how those differences can be eliminated in a principled way.

Firstly, there is the issue of differing coordinate systems – the robot would have likely been positioned and oriented differently in relation to the surface, when taking the two measurements. That would result in a translational and rotational difference between the scans. Secondly, the robot may have been at different distances, resulting in a difference in scales. Thirdly,

[2]Some architectures, such as PointCNN [4] (introduced in Section 3.2.1) are not order-invariant, but still perform well on standard point cloud tasks. Rigorously speaking, such descriptors are not deterministic functions of point clouds, but rather of point clouds with ordering.

on a related note, the scan taken from further away (with the same finite-resolution scanner) would have a lower density.

A simple normalisation step – setting the mean to 0 and variance to 1 (variance being the mean squared distance of points to origin) removes the translational and scale differences. Subsampling (deleting a subset of the points) can be used to alleviate the density disparity.

The only remaining difference left after applying the aforementioned steps is the difference in orientation. Keep in mind, that the two concrete point clouds do not only differ by a rotation. Different points on the surface were likely scanned at each attempt. For example, when scanning a person twice, one scan might miss the person's nose, while the other would not, just because the number of laser beams used is finite. Therefore, given just the point cloud data, there is no principled, exact way to recover the rotation by which the two underlying shapes differ.

This observation motivates interest in rotation-invariant descriptors, i.e. ones whose output does not change when a rotation is applied. Using such a descriptor would alleviate the final major remaining issue – the difference in orientation between the two scans.

### 2.4.2 Definitions

**Definition.** *$SO(3)$ denotes the group of all 3D rotations about the origin.*

**Definition.** *A 3D point-cloud descriptor $f : \mathcal{P} \to \mathcal{D}$, where $\mathcal{P}$ is the set of all 3D point-clouds, is said to be **rotation-invariant** if and only if, for all rotations $R \in SO(3)$ and all clouds $P \in \mathcal{P}$, there is $f(RP) = f(P)$.*

### 2.4.3 Remaining Difficulties

It must be noted that a rotation-invariant descriptor by itself does not resolve all problems encountered in practice by robots attempting localisation.

Firstly, and trivially, the descriptor still needs to be useful and extract the task-relevant information from its inputs – e.g. by putting two objects close to each other in the description space when they are ‘similar’ for the purposes of the task at hand. It needs to be robust to noise – small variations of the input point positions. Rotation-invariance without further guarantees is useless in itself.

Secondly, in some practical settings the objects of interest may be scanned from all the sides, but common scenarios, such as robot localisation, or vision for self-driving vehicles, involve only seeing objects from a single side. For example, a scan of a person from the front has a very different shape than a scan of the same person taken from the side.

Thirdly, in scenarios such as the two mentioned above, objects may be partly occluded by other things appearing between the object and the sensor.

Furthermore, in all of the above discussion, we have already assumed that we have a point cloud corresponding to an object we want to describe or classify. In practice, scans made by sensors contain many different objects in semantic sense, and need to be partitioned into ‘per-object clouds’ first. That is an area of active research, referred to as *segmentation*.

## 2.5 Supervised Machine Learning and Data Augmentation

### 2.5.1 Supervised Machine Learning in a Nutshell

In short, a supervised machine learning algorithm receives labelled examples of data and, based on those, produces a function ready to consume more data of that type.

In the point cloud classification setting, a *training set* of point clouds with their corresponding *true* classes would be provided. A source of the *truth* must exist – in classification, that source is usually humans manually labelling each input. Based on the information contained in the training set, the machine learning algorithm would produce a classifier function, which can later be applied to (previously unseen) inputs.

In the description setting for robot localisation, the ground truth labels are better interpreted as ‘object indices’, rather than classes. In general, ground truth for supervised machine learning can also take different forms than a collection of (input, label) pairs.

### 2.5.2 Training Neural Networks with Gradient Descent

The machine learning methods discussed in this dissertation are all based on neural networks – a family of functions parametrised by real-valued *weights*. We assume the reader's rudimentary familiarity with neural networks.

Recall that the basic way of training neural networks is backpropagation:

1. A batch[3] of inputs is fed into the neural network, which produces some outputs (e.g. descriptions, or class-probabilities which could be argmax-ed to predict a class).
2. A *loss* function computes a real-valued *loss* based on the labels and network's outputs.
3. The trainable parameters of the network are updated according to the gradient of the loss w.r.t. those parameters. While basic gradient descent just updates the weights along the current gradient ($w_{i+1} := w_i - \eta \cdot \nabla_{w_i} \mathcal{L}(w_i)$)[4], higher-order methods are usually used nowadays, with the commonly used Adam optimiser [26] being a prime example.

### 2.5.3 Relevant Loss Functions

This section presents some common loss functions for classification and description problems. Loss functions can be looked at as proxies between the mathematical formulation of our (neural network) function and the final goal we have in mind.

For example, while we are usually ultimately interested in optimising a classifier's accuracy, in order to get the gradients that we need for backpropagation, we perform a trick. We have the network produce intermediate values – *class probabilities* – from which the final prediction can be derived (using argmax). Those intermediate values are engineered to be differentiable functions of the input – as opposed to the value of classification accuracy – and are passed into a (differentiable) loss function. Finally, the gradient of the loss with respect to the weights of the network can be obtained and used to perform a training step.

[3]Often referred to as *mini-batch*, when only a subset of the training set is processed between training steps. In this work, we use the word *batch*, referring to mini-batches.

[4]Here, $w_t$ denote the network's weights at training step $t \in \mathbb{Z}$, $\eta$ denotes the *step size* parameter, $\mathcal{L}$ denotes the loss function and $\nabla_{w_i}$ is the gradient operator w.r.t. $w_i$.

**Definition.** ***Cross-entropy loss.*** *Classifier networks for a problem with $C$ classes are usually constructed to output $C$ class probabilities $p_c$ – non-negative numbers that sum to* 1. *When using a trained network to predict a single class, the argmax of those would be taken. During training, the numbers $p_c$ can be used to compute the differentiable cross-entropy loss, equal to $-\log p_y$, where $y$ is the true class.*

Consider the robot localisation setting for descriptors again – a robot may see an unbounded number of objects, some of them in multiple observations, in which case the robot's goal is to detect the repetition. That can be seen as a classification task with a dynamic, unbounded number of classes (objects). A different loss function is needed – the cross-entropy loss cannot be used in practice, as the number of class probabilities $p_c$ would keep growing without a bound.

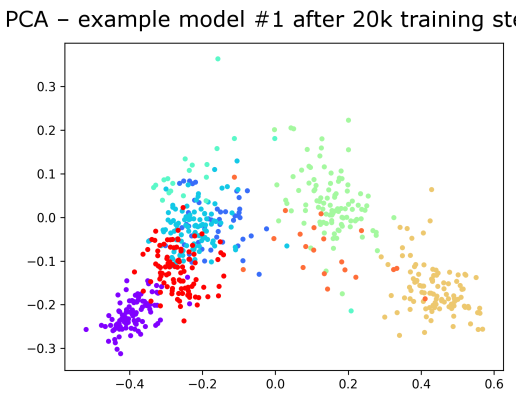
The idea behind the *pairwise* and *triplet* losses introduced below is to try to force the descriptions of objects to form *clusters* in the description space (also referred to as *feature space* further on). That means that any two descriptions of the same object should be close in feature space, while any two descriptions of different objects should be far away (relatively). See Fig. 2.4 for an example of clustering.

**Definition.** ***Pairwise loss*** *is defined to be the sum of distances of all pairs of inputs in the batch that belong to different classes.*

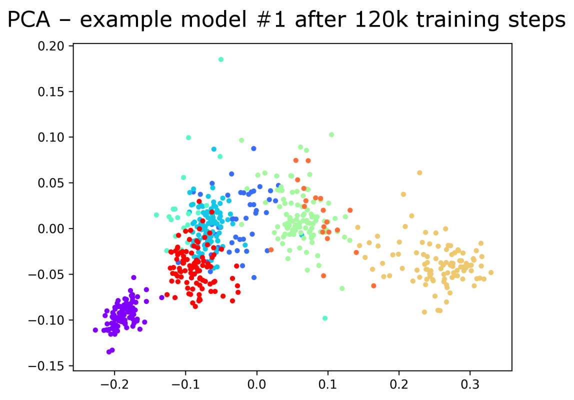
**Definition.** ***Triplet loss*** *is defined to be a sum over all triples of input clouds $(a, p, n)$ in the batch, for which the 'anchor' $a$ has the same class as the 'positive' example $p$ and a different class than the 'negative' example $n$. Given a distance metric $d$ over the feature space, the triplet loss is:*

$$\sum_{(a,p,n)} \max \left\{ 1 - \frac{d(a,n)}{d(a,p)+\epsilon},\ 0 \right\}. \tag{2.1}$$

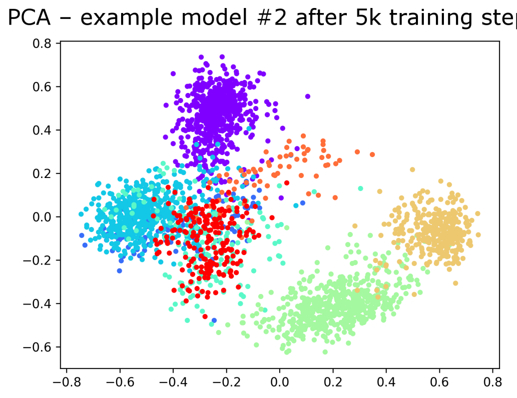
The purpose of the triplet loss as defined in Eq. 2.1 is to penalise all triplets, in which the negative example is closer to the anchor than the positive example. The effect of a non-zero margin variable $\epsilon$ is that the loss also penalises anchors $a$, for which $n$ is only slightly further



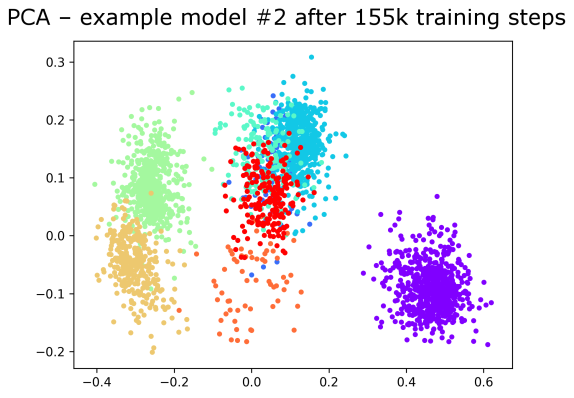
(a)



(b)



(c)



(d)

Figure 2.4: Class clustering visible in feature spaces of two different descriptors (top and bottom) trained using a combination of pairwise and triplet losses. Different colours correspond to different classes of input clouds. The high-dimensional feature space was mapped into two dimensions for this plot using standard Principal Component Analysis (PCA). One can see the clustering become stronger as training progresses (left to right). The fact that some classes overlap does not indicate a failure of the classifier, as this is merely a 2D projection of the feature space.

away from $a$, than $p$ is.

The intuitive meaning of the margin is related to that in the Support Vector Machines (SVMs) [27], though many differences are present, e.g. the margin is a constant and not a subject of optimisation, and it represents a ratio of distances, as opposed to a feature-space distance, as in SVMs.

We use $\epsilon := 0.01$ as the margin value, found to work well empirically. It is important to note that Eq. 2.1 defines one particular variant of triplet loss. Slightly different formulations, all based on (anchor, negative, positive) triplets, can be found in the literature under the same umbrella name 'triplet loss'.

### 2.5.4 Data Augmentation

*Data augmentation* is a generic method of aiding supervised training by generating new labelled data from pre-existing data.

Domain knowledge can be used to systematically generate new data. For example, the MNIST handwritten digit classification problem is translation-invariant (as long as the translation does not shift the digit out of the image). What that means is that applying a small translation to the input should not change the output (class) of the classifier. Similarly, the 3D object classification problem is rotation invariant.

A data augmentation step could then involve taking existing MNIST input images and applying translations to them to create new training examples, and we know, by our domain knowledge, that the classes of those inputs would be unchanged. Similarly, one could take labelled 3D objects and rotate them while preserving the label.

Data augmentation by itself does not guarantee that the trained network will be invariant – it may cause the network to become more robust to the transformation at hand, and often works well in practice, as also observed in our experiments (see Chapter 4).

Undeniably, augmentation is a way of cheaply (i.e. without need for experts to provide labels) generating extra meaningful training data for the network to learn from, and thus generally improves generalisation of the trained models.

Finally, data augmentation can go beyond applying hand-crafted transformations to inputs and labels. Neural approaches to generating the augmented data have also been studied [28], including the use of the well-known Generative Adversarial Networks (GANs) [29].

## 2.6 The Spherical Fractal Convolutional Neural Network

### 2.6.1 Introduction

The Spherical Fractal Convolutional Neural Network (SFCNN) is a graph-convolution-based deep architecture for point cloud classification[5], introduced by Rao et al. [5]. We present its details in Chapter 5.

The SFCNN has set a new state of the art on the ModelNet40 object classification benchmark [5] (presented in more detail in Section 6.1). While it is not exactly rotation invariant, its design leads to strong robustness against rotations. It performs exceptionally well on the test set containing previously unseen rotations, compared to other methods.

### 2.6.2 Role of the SFCNN in Our Project

We implement, train, and evaluate the SFCNN model. Full reimplementation was necessary, as we have not obtained a reference implementation from the authors. We also discuss the architecture's rotation robustness and derive a version that is provably rotation invariant (under certain idealistic assumptions).

Furthermore, we spot and quantitatively confirm that the SFCNN model tends to disregard a large percentage of input points. As an original contribution, we address this by proposing two alternative modifications of the model. We implement and evaluate both.

[5]The authors also show how to use it for point cloud segmentation, using an encoder-decoder architecture. The segmentation task is outside the scope of this dissertation.

# Chapter 3

# Related Work

## 3.1 Point Clouds meet Deep Learning – The PointNet

### 3.1.1 Architecture

The PointNet architecture introduced by Qi et al. [25] is the first deep neural network specialised for point cloud processing. The authors demonstrate its usefulness for both classification and segmentation of point clouds.

Recall that a 3D point cloud is defined as an unordered set of points $(x_i, y_i, z_i)$. However, practical implementations of neural networks consume ordered data in the form of vectors or, more generally, tensors. Therefore, it is a non-trivial requirement that an architecture for point cloud processing be invariant to the order in which the points are input.

The key idea behind PointNet's order invariance is expressed by the following equation:

$$f(\{x_1, \ldots, x_n\}) = h(g(x_1), \ldots, g(x_n)), \tag{3.1}$$

where $h$ is a symmetric function, i.e. one invariant to permutations of its $n$ inputs.

The PointNet is based on a simple instantiation of Eq. 3.1. It implements $g$ with a shared

Multi-Layer Perceptron (MLP), and uses maxpooling for as the symmetric function $h$. In words, for each point $p_i$ of a cloud, the same MLP is applied, mapping its 3D coordinates $(x_i, y_i, z_i)$ onto a vector $f_i$ in a higher-dimensional space. Then, a point-wise maximum is taken over all $f_i$, yielding an order-invariant description of the point cloud.

In addition, the PointNet also incorporates an attempt at instilling invariance to rigid transformations of the input. The authors state, as motivation, the goal of achieving stable segmentation. An affine transformation matrix is predicted for each input cloud – with the goal of aligning the input into a canonical coordinate system – and applied before the description begins. An analogous step is also repeated in feature space, between two MLPs. The transformation matrices are performed using *Joint Alignment Networks*, which also utilise MLPs and maxpooling.

### 3.1.2 Results

The PointNet achieved state of the art in the ModelNet40 classification task [10], and outperformed a convolutional, voxel-based baseline, as well as previous classical solutions, on a ShapeNet [30] segmentation task proposed in [31].

In an ablation study, the authors demonstrate that the Joint Alignment Networks help improve classification accuracy of the network. However, no explicit comments are made on the level of invariance the trained models have to rigid transformations. The authors experimentally confirm robustness of the network to changes in point cloud density and random noise.

Furthermore, a universal approximation theorem about the PointNet architecture is proved, demonstrating the PointNet can learn to approximate any continuous function acting on point clouds. Such theorems are commonly proved in deep learning research to support newly introduced architectures.

Both the experimental and theoretical results affirm that the PointNet is a simple, yet robust and powerful, novel architecture. The good results served as motivation form further research of deep networks acting directly on point clouds.

### 3.1.3 PointNet++

While the PointNet outperformed a 3D-convolution-based voxel approach, its lack of regard for local structures is likely still a disadvantage. Combining the hierarchical idea of convolutional networks with the PointNet architecture, Qi et al. proposed the PointNet++ [32].

The PointNet++ model is composed of multiple layers. Each layer takes in a set of points in an $n$-dimensional space, each point possibly endowed with a feature vector. The initial layer would consume the original input cloud, and beyond that, each layer would consume the output of the previous layer. Each layer consists of the following stages:

1. A set of *centroid* points is chosen amongst the input points, using a *furthest-point sampling* method, which results in *centroids* that cover the entire input cloud approximately uniformly.

2. For each centroid $c_i$, a subcloud of the input cloud is chosen, by selecting nearest neighbours of $c_i$.

3. For each centroid, the corresponding subcloud is described using a PointNet, and the resulting feature vector $f_i$ is passed on to the next layer, at the position $c_i$.

Thus, the number of points processed at the next layer can be reduced, analogously to how 2D image-processing convolutional networks often reduce the image resolution across the consecutive layers.

The authors show that the PointNet++, with its hierarchical structure, is an improvement over the basic PointNet. It achieves significantly better results (having established new state of the art scores) on ModelNet40 [10] and MNIST[1] [20] classification tasks.

[1]The MNIST dataset can be used to generate 2D point clouds by sampling the surfaces of the handwritten digits.

## 3.2 Convolutions on Point Clouds

While the PointNet++ architecture provides an example of a point cloud architecture able to process local structures, it does not use learned convolutions, which have proved highly successful in many other applications of deep learning. However, other architectures exist, that employ convolutions for point cloud processing in various ways.

### 3.2.1 PointCNN

The key idea behind the PointCNN architecture [4], is to learn so-called $\mathcal{X}$-transformations, that map point clouds into ordered spaces $\mathbb{R}^n$, where usual convolution operators can be applied. The $\mathcal{X}$-transformations would ideally preserve the shape information of a cloud, while being order-invariant. They are implemented using Multi-Layer Perceptrons (MLPs) by Li et al., consuming both the point coordinates, and associated feature vectors. While the $\mathcal{X}$-transformations learned in practice have been found to not exhibit the aforementioned ideal properties [4], the PointCNN has, nevertheless, outperformed state of the art on numerous well-established classification and segmentation benchmarks.

The hierarchical structure of PointCNN resembles that of PointNet++ – a number of *representative* points are selected from the input cloud, and then the neighbourhoods around those key points are all processed with a shared transformation. In the case of PointCNN, the shared transformation is an $\mathcal{X}$-transformation, followed by a classical convolution – both steps together are referred to as an *$\mathcal{X}$-convolutional layer.*

Each $\mathcal{X}$-convolutional layer can be parametrised by the following 4 parameters, designed by Li et al.:

- $K$ – size of the subcloud selected for each representative point
- $D$ – a dilation factor – the $K$ points are selected at random out of the $K \cdot D$ nearest neighbours of the representative point (for $D = 1$, that reduces to the standard nearest neighbours approach)

- $P$ – number of output (representative) points
- $C$ – dimensionality of the output features

To apply a PointCNN network to a classification task, the authors compose a number of $\mathcal{X}$-convolutional layers, followed by a number of fully-connected layers.

### 3.2.2 Classical Grid Convolutions

Classical grid-based 2D and 3D convolutions can be applied to point cloud problems by converting them into other, grid-based, representations. As mentioned in Section 2.1.4, point clouds can be converted to a voxel representation by aggregating the point information into cells of a discrete voxel grid, or an octree (see Section 2.1.1). A convolutional approach for point cloud segmentation and classification, has been proposed by Riegler et al. [7].

Point cloud data can also be aggregated into 2D images, enabling the use of thoroughly studied image-processing convolutional neural networks. For example, Ku et al. [33] use the Bird's Eye View (BEV) representation [19] of LiDAR clouds, coupled with RGB camera images, for object detection in self-driving cars. The BEV representation projects a point cloud onto a horizontal plane partitioned into a pixel grid. Each pixel is then assigned features based on the maximum height, number of points in that pixel (density), and reflectance of the highest point (intensity). This lossy representation has been proven useful in conjunction with other representations.

### 3.2.3 Graph Convolutions

Ideas from Graph Convolutional Neural Networks, rising in popularity in recent years, have also been applied to point cloud processing. Wang et al. [34] introduced their Dynamic Graph Convolutional Neural Network (DGCNN) architecture, composed of *EdgeConv* operations. Those, at each step, aggregate features of a graph vertex' neighbours into a single feature vector, in an order-invariant way.

For use with the DGCNN, a point cloud is converted into a graph by introducing edges between nearest-neighbour points in the cloud. The other key idea of the DGCNN is that the neighbourhood graph be recomputed after each convolutional step, using feature-space neighbourhoods. This way, the DGCNN can exploit both spatial locality and semantically meaningful long-distance relations (feature-space locality).

In a theoretical observation, Wang et al. show that the PointNet [25] is a special case of DGCNN. Their DGCNN outperformed state of the art results on the ModelNet40 classification task [10], including the scores achieved by the PointNet [25], PointNet++ [32], and PointCNN [4] architectures mentioned previously in this Chapter.

Finally, Li et al. [35] show how to improve Graph CNNs, by transferring techniques used in other branches of deep learning to enable successful training of deeper networks. Those techniques are residual connections [36], dense connections [37], and dilated convolutions [38]. They use the DGCNN as their baseline, and improve upon its performance at a segmentation task, by constructing similar, but deeper, graph convolutional networks.

## 3.3 Handcrafted Rotation-Invariant Features

The previous Sections describe a number of successful point cloud architectures, not concerned with rotation invariance. The original PointNet's Joint Alignment Networks were posited to encourage invariance to affine transformations, which include rotations, and while the presence of those modules has helped improve the PointNet's performance, no tests of robustness or invariance were reported on.

The following Sections focus exclusively on architectures that are robust or invariant to rotations.

Consider the following toy representation: a cloud's centre of mass $m$ is computed, and then for each point $p_i$, its coordinates $(x_i, y_i, z_i)$ are replaced with a single number denoting the distance between $p_i$ and $m$. That representation of a cloud is fully invariant to rotations of the input, because rotations preserve distances. An arbitrary architecture can be applied to the resulting one-dimensional representation, and would always yield a rotation-invariant output.

While that toy representation likely discards too much information to compete with state of the art models, a stronger rotation-invariant representation, similarly hand-crafted, exists. It was discovered by Chen et al. [39] and named the Rigorously Rotation Invariant (RRI) representation. The key theoretical result proved about the RRI representation is stated in the following Theorem:

**Theorem 1.** *The RRI representation loses no information about the point cloud, other than its absolute orientation, under a mild assumption.*

This is saying that the original point cloud can always be reconstructed from its RRI representation (under the aforementioned mild assumptions, which we elaborate on later), up to a rotation – which is perfect, in the context of rotation-invariant representations.

The RRI representation of a 3-dimensional point $p_i$ is a $(3k+1)$-dimensional vector, where $k$ (a hyperparameter of the model) is the number of nearest neighbours selected for each point of the cloud. The vector consists of $r_i$, i.e. the magnitude of $p_i$ as a vector, and $k$ triples $(r_{ij}, \theta_{ij}, \phi_{ij})$, one per a neighbour $p_{ij}$ of $p_i$, where:

- $r_{ij}$ is the magnitude of $p_{ij}$, i.e. its distance from origin,
- $\theta_{ij}$ is the angle between the vectors $p_i$ and $p_{ij}$,
- $\phi_{ij}$ requires a longer explanation. Consider projections $q_{ij}$ of points $p_{ij}$ onto the plane that passes through the origin and is perpendicular to $p_i$. Ordering those projections clockwise (with $p_i$ pointing 'up'), $\phi_{ij}$ is the angle between $q_{ij}$ and whatever projection $q_{il}$ (for some $l \neq j$) precedes it.

As Theorem 1 states, that representation of a point cloud is lossless, under a *mild assumption*. That assumption is that the graph of $k$ nearest neighbours is strongly connected, i.e. one can get from any point $p_i$ of the cloud to any other point $p_j$ in a finite number of steps, each step leading from a point to one of that point's $k$ nearest neighbours.

This assumption becomes trivially true when $k$ is equal to the size of the point cloud, but can also hold for smaller values of $k$. Furthermore, the authors discovered that even decreasing the value of $k$ low enough, that 25% of the clouds from the ModelNet40 dataset do not satisfy the connectedness condition, does not decrease the model's performance drastically (although there is an observable decrease).

Chen et al. use RRI in conjunction with a custom hierarchical architecture (ClusterNet), and achieve state of the art on the variants of the ModelNet40 classification task that favour rotation-robust models most[2].

Another hand-crafted representation, weaker than the RRI, has also been used successfully. PPF-FoldNet [40] is an autoencoder architecture that learns rotation-robust representations of point clouds in an unsupervised manner. Before further processing (based on the FoldingNet architecture [41]), the architecture converts input clouds into Point-Pair Feature (PPF) representation, also encoding distances and angles amongst local sets of points, similarly to the RRI. The representations learned by the PPF-FoldNet are not exactly rotation-invariant but exhibit strong robustness to rotations and achieve a new state of the art on the 3DMatch benchmark [42].

Zhang et al. [43] have devised an architecture which applies 1D convolutions to data derived from another rotation-robust representation of points, based on distances and angles. They have defined a 'rotation invariant convolution' operator, which can be stacked, similarly to convolutions in classical Convolutional Neural Networks. Their model exhibits very high robustness to rotations, demonstrated on the ModelNet40 classification benchmark.

## 3.4 Rotation Equivariance

A mapping $\phi$ is *equivariant* to rotations, if applying a rotation $R$ to any input transforms the output in a predictable way (denoted $\mathcal{T}_R$), depending on that rotation $R$. This property is conveyed concisely by the following equation:

[2]We are referring to the SO(3)/SO(3) and $z$/SO(3) classification accuracy benchmarks, as introduced in Section 6.1.

$$\phi(Rx) = \mathcal{T}_R(\phi(x)) \tag{3.2}$$

Equivariance is a generalisation of invariance (which corresponds to $\mathcal{T}_R = \text{id}$ for all $R$). Equivariant operators can be used to construct architectures robust to rotations, as showcased by Worrall & Brostow's voxel-processing architecture – CubeNet [44].

The authors use the group convolution operator, which effectively introduces an extra (finite) dimension to be convolved over, corresponding to elements of a finite group. They prove mathematically that group convolution is equivariant to the actions of that group. Worrall & Brostow train their CubeNet – an architecture composed of multiple Group Convolution layers, followed by a fully-connected block – using a small group of 4 3D rotations, known as *Klein's Vierergruppe* $V$[3]. They found the four-group to outperform the larger tetrahedral group $T_4$ (12 rotations), and also to be preferable to the Cube Group (24 rotations), which was too large to be used in training effectively.

The CubeNet, using $V$-group convolutions, achieved state of the art results (amongst voxel architectures) on the ModelNet10[4] classification task [10], and comparable to state-of-the-art results on a biological volume segmentation dataset, the *ISBI 2012 Challenge* [45].

Another work by Worrall et al. [46] introduces Harmonic Networks – 2D image processing networks, equivariant to all (infinitely many) 2D rotations. Harmonic Networks are based on convolutions[5] with *complex harmonic spherical* filters of the form:

$$W_{m,\beta,R}(r,\theta) = R(r) \cdot e^{i(m\theta+\beta)}, \tag{3.3}$$

where $(r, \theta)$ are polar coordinates, $R$ is a real function of the radius known as the *radius profile*,

[3] Also known as the *four-group*, it is the smallest non-cyclic group of 3D rotations, meaning that it is not merely generated by composing a single rotation with itself a varying number of times.

[4] The ModelNet10 is a smaller dataset, containing a subset of the ModelNet40 models, with all shapes of a single class aligned with each other.

[5] Or more precisely, cross-correlations. The two operations have similar definitions and are often both referred to as convolutions in scientific writing.

$m \in \mathbb{Z}$ is known as the *rotation order*, and $\beta$ is a phase offset term. Convolutions with those harmonics are are provably rotation-equivariant. The authors note the equivariance alleviates need for data augmentation. Indeed, Harmonic Networks were shown to achieve state of the art on a rotated MNIST [20] benchmark, outperforming, amongst other approaches, data-augmented Convolutional Neural Networks.

A generalisation of the Harmonic Networks to three dimensions has been successfully implemented by Cohen et al. [47] in their Spherical CNNs, and later, in similar work by Thomas et al. [48] in Tensor Field Networks. Cohen et al. perform spherical correlations – an analog of classical 2D convolutions, that replaces 2D translations with 3D rotations. They implement an efficient way of computing the spherical correlations, based on the Fast Fourier Transform.

The resulting architecture is equivariant to all 3D rotations and achieves competitive results on object recognition benchmarks. To apply their architecture, Cohen et al. map input meshes onto spheres, and proceed applying spherical correlations.

## 3.5 An Alternative Approach – Alignment Preprocessing

An alternative to rotation-robust or -invariant architectures is to rotate the inputs into a canonical orientation, in a preprocessing step. For example, the SegMap robotic localisation pipeline [15] aligns the principal axis[6] of the input cloud to the $x$ axis, assumes the $z$ axis, corresponding to the direction of gravity, should remain unchanged[7], and resolves the remaining ambiguity using a hand-crafted heuristic, maximising the density of the part of cloud located in a certain halfspace of $\mathbb{R}^3$.

Preceding the description step with this kind of alignment was found to be a valid method for SegMap's authors' applications. However, used in conjunction with fast descriptors, the running time of the alignment step may become the architecture's bottleneck, slowing down

[6]Directions of the eigenvectors, found with 2D Principal Component Analysis, ordered by the magnitude of their eigenvalues.

[7]That is because SegMap is designed for mobile robots, and those can measure the gravity vector with hardware, and align the $z$ axis of their scans to that.

the forward-pass time, critical in real-time applications.

Additionally, the alignment step is not guaranteed to be stable – the rotation found by the alignment process may change abruptly due to small variations of the input. That results in lower robustness to input noise than that exhibited by rotation-invariant neural networks. As mentioned in Section 2.4.1, a perfect alignment method does not exist, and therefore rotation-invariant networks, robust to input noise, seem to be the preferable research direction.

# Chapter 4

# Data Augmentation for Rotation Invariance

## 4.1 Issues with Non-Rotation-Robust Models

### 4.1.1 Natural Segmentation and Matching (NSM)

The motivation for our study of data augmentation comes from problems observed by the authors of the Natural Segmentation and Matching (NSM) robot localisation algorithm [14].

NSM is used by a robot with the purpose of localising its current pose (position and orientation) in a prior map, based on the robot's recent LiDAR readings. The map consists of a number of scans (or more concise descriptions of the scans) taken at previous times, each with an associated robot pose, at which the scan was taken. The task of NSM is to predict the robot's current pose, relative to the map, given its most recent observation.

The NSM pipeline is composed of multiple stages, listed below. Also see Fig. 4.1 for a visualisation.

1. A pre-filtering and segmentation of the point cloud representing the surroundings.
2. Description of the segments using a trained PointCNN network [4].

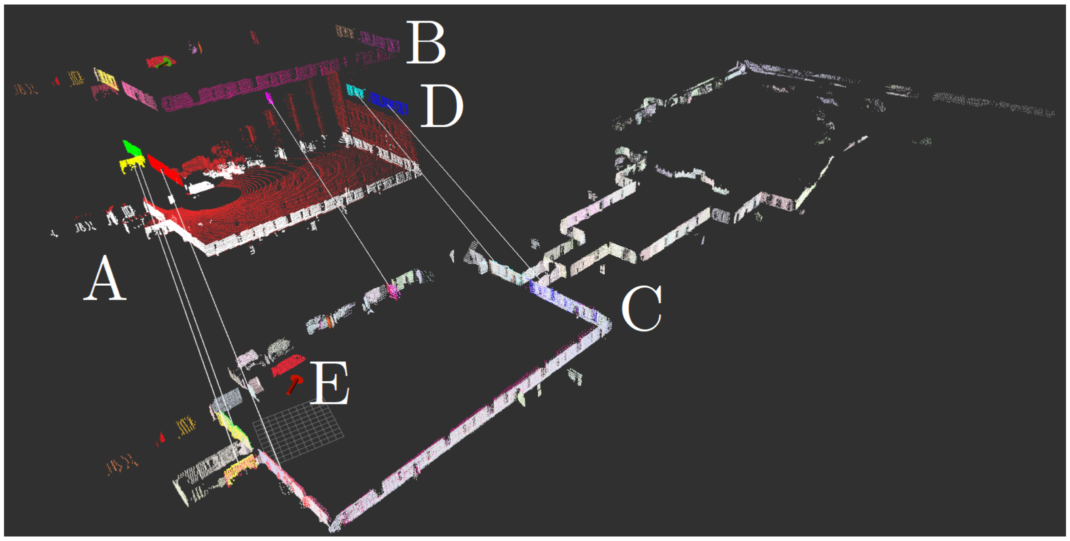


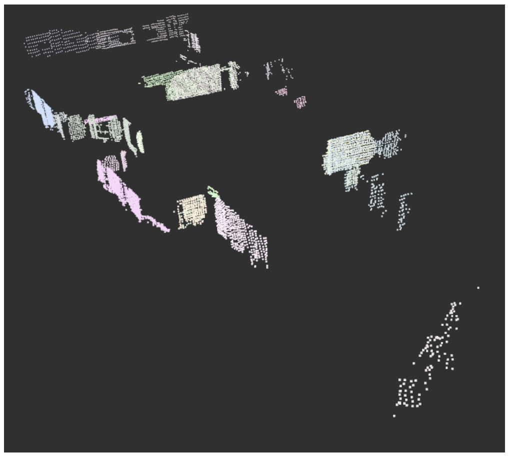
Figure 4.1: The NSM pipeline illustrated. A – in red: an example input cloud; in white: the input cloud truncated along the $z$-axis to remove the ground and ensure consistent segment height. B – the segmented input cloud. C – the (segmented) map to localise in. D – six input segments that were matched to segments of the map in a geometrically consistent way. The matchings are illustrated with the white lines. E – the red arrow to the left of the letter 'E' marks the estimated pose of the robot in the map's coordinate frame.
This visualisation features the Newer College Dataset [1] and was made using RViz software [2].

3. A geometric consistency test, which receives candidate matches of the form (map segment, observed segment), based on the PointCNN features, and attempts to find a large enough number of matches that are all geometrically consistent. That means there should exist an isometric[1] transform that maps the coordinate frame of the robot onto the coordinate frame of the map, so that each (matched) observed segment is mapped onto its matching map segment.

### 4.1.2 Dataset

NSM has been tested on a LiDAR dataset collected by the Dynamic Robot Systems Group at the Mathematics Institute campus in Oxford. The dataset consists of a map, composed of 791 LiDAR scans, covering the entire area (see Fig. 4.2a), and test observations (example in Fig. 4.2b) from another round of scanning (the scans were obtained by a sensor traversing the area in loops). The objective is to determine the correct pose in the map's coordinate frame,

[1]An isometry is a distance-preserving transformation.


(a) The map included as part of the Mathematics Institute dataset. The aim of a localisation algorithm would be to determine the pose in this map that a given LiDAR measurement has been taken from.

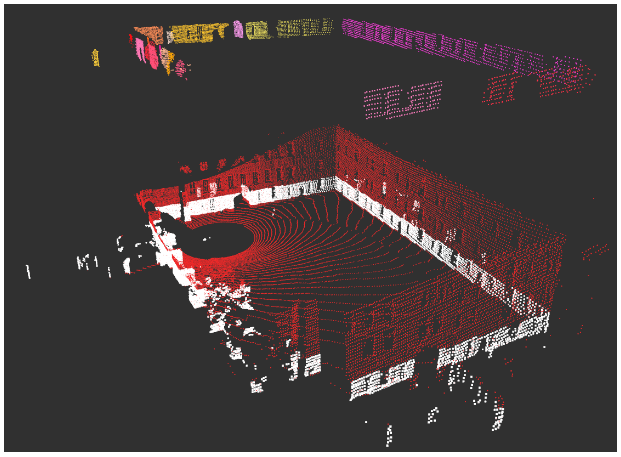

(b) In red: an input cloud given to NSM. In white: the input cloud trimmed to remove the ground and ensure consistent height of the segments as a normalisation step. In colour, above: the result of segmentation performed by NSM.

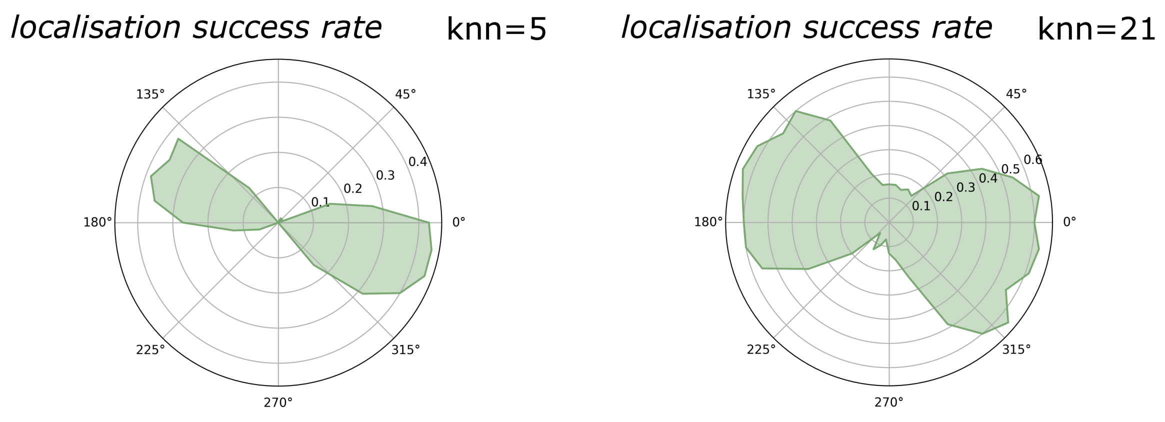
Figure 4.2: The NSM localisation algorithm running on the Mathematics Institute dataset. Visualisation made in robotics software RViz [2].

from which a given test observation was taken.

### 4.1.3 Quantitative Analysis

It has been noticed that NSM struggles for test observations taken at a significantly different angle than the map scans. At the beginning of this project, we have implemented a benchmark to see how well exactly NSM handles rotations of the test segments. The data collected in Fig. 4.3 shows that NSM's rate of successful localisations deteriorates drastically when all the test segments are rotated about the $z$ (vertical) axis by the same angle, except for rotation angles close to $0°$ or $180°$.

These results motivate the investigation of data augmentation, as the NSM is implemented in a modular fashion, which allows us to swap out the non-rotation-robust PointCNN descriptor for a different one – e.g. a version of the same descriptor trained with data augmentation.



(a) Success rate for `knn=5`.
(b) Success rate for `knn=21`.

Figure 4.3: Success rate of NSM for rotated inputs on the Mathematics Institute LiDAR dataset, plotted as a function of the rotation angle.
The data is collected over 70 test clouds. Two different values were provided for the `knn` parameter of the geometric consistency module, which determines how many candidate matches are considered. Larger parameter values trade off the increase in running time for an improved success rate.
180° rotations are handled well – that is because the segments in the dataset tend to be flat pieces of walls, and thus do not change much when a 180° rotation is applied.

## 4.2 The Subject of our Study

Data augmentation, as introduced in Section 2.5.4, means automatically generating new labelled training examples based on existing data. By adding rotated versions of inputs into the training set, with the same labels as the original clouds, one can encourage deep models to learn to be robust against rotations.

It is important to understand that, in general, data augmentation gives no hard guarantees about properties of the learned model.

The natural way of using data augmentation for rotation robustness is to sample and apply a random rotation each time a training cloud is fed into the model. An alternative paradigm would be to generate a finite number of new inputs at the start, and then iterate many times over that augmented training set.

For our study, we stick to the first paradigm, with one slight generalisation, discussed later,

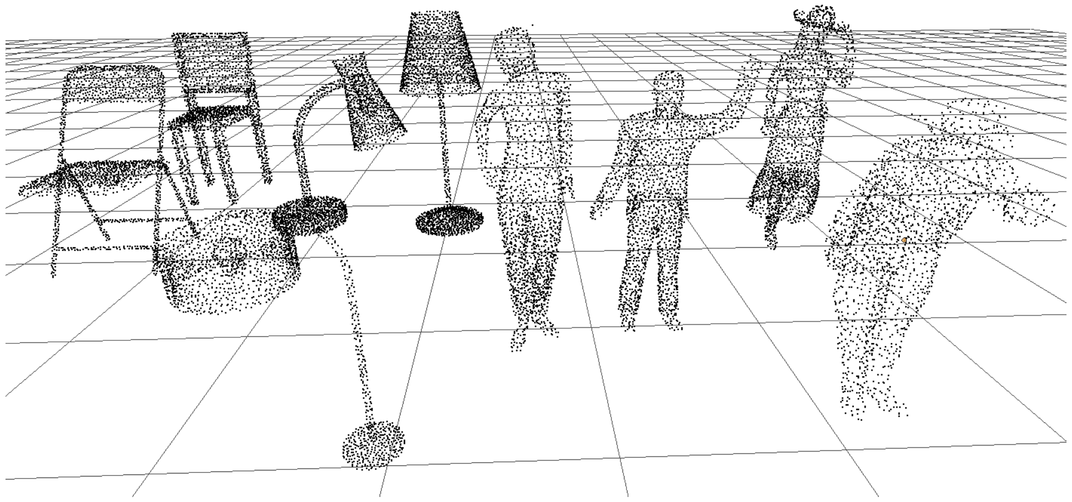


Figure 4.4: Nine example point clouds from three different classes in the ModelNet40 dataset, in their original orientation and scale. Left to right: two chairs, three lamps, and four humans. Considerable in-class variability is exhibited – the humans are in different poses, the lamps and chairs have significantly different shapes. Each of the clouds contains 2048 points and comes with the centre of mass shifted to the origin and variance (i.e. mean distance of points to the origin) normalised to 1.

where the rotations used in a single training batch are not all pairwise independent. We test multiple different distributions of rotations, and compare the reaction of the trained models to input rotations, using custom benchmarks.

## 4.3 The ModelNet40 Dataset

The primary dataset we use in this project (both for the study of data augmentation, and the SFCNN) is the ModelNet40 dataset [10] – one of the most common benchmarks for 3D object classification methods. The dataset contains 12308 three-dimensional meshes representing 'everyday life' objects from 40 different classes, such as *airplane*, *keyboard*, *plant*, or *table*. The dataset contains different numbers of objects of different classes.

For our project, we have used a publicly available[2] version of ModelNet40 with point clouds of size 2048 sampled from the meshes. Some example clouds are shown in Fig. 4.4.

[2] `https://shapenet.cs.stanford.edu/media/modelnet40_ply_hdf5_2048.zip`

| Layer | $K$ | $D$ | $P$ | $C$ |
|---|---|---|---|---|
| 1st | 8 | 1 | $-1$[p] | 48 |
| 2nd | 12 | 2 | 384 | 96 |
| 3rd | 16 | 2 | 128 | 192 |
| 4th | 16 | 3 | 128 | 384 |

Table 4.1: Configuration of the PointCNN model we use in our study of augmentation methods. For the meaning of $K, D, P, C$, refer to Section 3.2.1.
[p]The $-1$ given as value of $P$ indicates that the number of output points of a layer is the same as the number of its input points – in other words, all points are selected as representatives.

The dataset was used in a 9840:2468 (80%:20%) training/testing split, following the original SFCNN paper [5].

## 4.4 The Descriptor

We apply the various data augmentation routines to the training of a PointCNN [4] model (see Section 3.2.1) on the ModelNet40 dataset.

Our PointCNN descriptor is trained with a sum of triplet loss and pairwise loss (see Section 2.5.3), which encourage clustering of the input clouds in feature space according to their true classes. We use the same PointCNN model configuration, as previously used by the authors of the PointCNN paper [4] for their ModelNet40 classification experiment. The configuration can be found on the PointCNN author's public repository[3]. The model contains 4 $\mathcal{X}$-convolutional layers, with parameters listed in Table 4.1. The only difference between our model and the one used by Li et al. is that, instead of a 40-class classifier, we are training a 16-dimensional descriptor – that is achieved by changing the shape of the final fully-connected layer.

The network has approximately 600k parameters and training it with a batch size of 128 requires 8954MB of GPU memory.

[3]`https://github.com/yangyanli/PointCNN/blob/master/pointcnn_cls/modelnet_x3_l4.py`

### 4.4.1 Computational Resources

Our research group, the Dynamic Robot Systems Group at the Oxford Robotics Institute, has provided us with a work laptop, as well as remote access to one of the institute's computers with an NVIDIA Titan V GPU (12GB of RAM). The development and evaluation has primarily taken place on that machine (via `ssh`).

The Oxford Robotics Institute has also provided us with access to the JADE (Joint Academic Data Science Endeavour) high-performance computing cluster[4], where some of the training for our project was run using JADE's job scheduling system `slurm` [49].

## 4.5 Rotation Robustness Benchmarks for Descriptors

To evaluate the effect of data augmentation on rotation robustness, we have designed and implemented three generic benchmarks for descriptors. We limit both the augmentation and our benchmarks to only use rotations about a single axis ($z$).

Each of the benchmarks maps angles $\theta \in [0, 2\pi]$ onto real-valued scores, corresponding to how *well* the descriptor handles $z$-rotations by $\theta$ being applied to its inputs. The meaning of *well* depends on the benchmark.

While all the benchmarks could have an arbitrary point cloud dataset plugged in, we consistently use the ModelNet40's test set.

All three benchmarks are constructed similarly, therefore we first introduce some notation to unify all three:

- Let $C$ denote a set of point clouds used by the benchmark (we used the ModelNet40's test set).
- Let $R^z_\theta$ denote the $z$-rotation by an angle $\theta$.
- Let $f : C \to \mathcal{D}$ denote the descriptor being benchmarked, where $\mathcal{D}$ is the metric feature

[4]`https://www.jade.ac.uk/`

space equipped with a distance metric $d : \mathcal{D} \times \mathcal{D} \to \mathbb{R}$.

- Let $c_i \in C$ denote a single input point cloud.
- Let $s$ denote the *score function* of the benchmark, which maps a pair $(R, c_i)$ onto a real-valued score. The three benchmarks will differ by their definitions of $s$.

  The final output of each benchmark is the function $S$ defined as:

$$S(\theta) = \frac{1}{|C|} \sum_{i=1}^{|C|} s(R^z_\theta, c_i), \tag{4.1}$$

  i.e. each rotation angle is mapped onto the average score over all input clouds. We find this mapping is best visualised as a polar plot, such as the ones seen in Figures 4.5, 4.6, 4.7.

### 4.5.1 The `eknn` benchmark

This benchmark's name stands for 'Exclusive $k$ Nearest Neighbours'.

Let $N \subseteq C \setminus \{c_i\}$ of size $k$ be the set of point clouds from $C$ that are $k$ nearest neighbours of $R^z_\theta c_i$ (excluding $c_i$ itself).

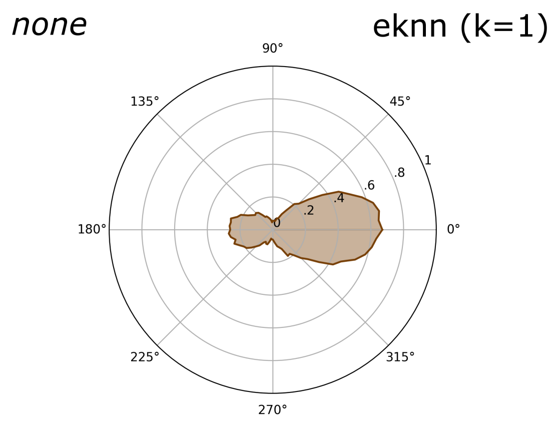
Then the score is:

$$s(R^z_\theta, c_i) := \begin{cases} 1.0 & \text{if } N \text{ contains a cloud of the same class as } c_i \\ 0.0 & \text{otherwise.} \end{cases} \tag{4.2}$$

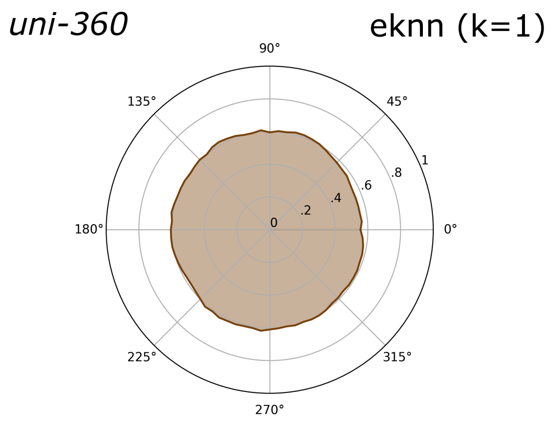
$S(\theta)$ is therefore a real number in $[0, 1]$, where **larger** values are better.

For example, the `eknn` benchmark with $k = 1$ maps an angle $\theta$ to the percentage of clouds $c_i$ for which the nearest-neighbour of the rotated cloud $R^z_\theta c_i$ in feature space is of the same class as $c_i$. Example plots of $S(\theta)$ can be found in Fig. 4.5.

Thus the `eknn` score (with $k = 1$) of a descriptor $f$ at the angle 0 gives a theoretical lower



(a) A basic PointCNN model, not robust to rotations.



(b) A PointCNN model trained with data augmentation, and thus robust to rotations.

Figure 4.5: Example `eknn` scores ($k = 1$) for two different models. The basic model has the highest score at $0°$, and the score deteriorates away from $0°$, being worst at $\pm 90°$. For the robust model, the score stays approximately constant at all angles (equal to the $0°$-score of the basic model), indicating strong robustness to rotations.

bound on classification accuracy we can obtain – one could always use the descriptor $f$ and turn it into a classifier by picking the class of the nearest neighbour in the training set. This may not be a practical approach for too large datasets as it requires storing the entire training set worth of descriptions.

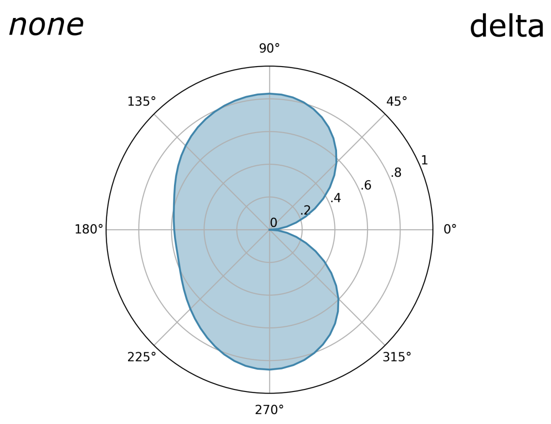
One could also hope to further improve classification accuracy by, for example, training a Multi-Layer Perceptron that takes the description as input and outputs classification logits.

### 4.5.2 The `delta` benchmark

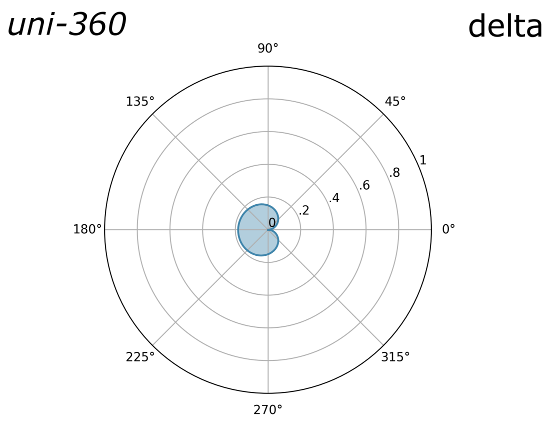
In short, this benchmark looks at the distance by which a rotation shifts descriptions in the feature space (normalised by dividing by the average feature space distance between $R^z_\theta c_i$ and everything in $C$):

$$s(R^z_\theta, c_i) := \frac{d\left(f\left(R^z_\theta c_i\right), f(c_i)\right)}{\frac{1}{|C|}\sum_{j=1}^{|C|} d\left(f\left(R^z_\theta c_i\right), f(c_j)\right)}. \tag{4.3}$$

$S(\theta)$ is therefore a real number in $[0, \infty)$, where **smaller** values are better. A perfectly



(a) A basic PointCNN model, not robust to rotations.



(b) A PointCNN model trained with data augmentation, and thus robust to rotations.

Figure 4.6: Example `delta` scores for two different models. The score at 0° is 0 by definition. The score deteriorates at angles further from 0°. The performance reduction is much greater for the non-robust model, but the scores do stay below 1.0, indicating that the descriptor is better than random. The scores are non-zero for the model trained with data-augmentation, indicating it is not exactly invariant to rotations.

rotation-invariant model would have a `delta` score of 0.0. Example plots of $S(\theta)$ can be found in Fig. 4.6.

### 4.5.3 The `mean-dist` benchmark

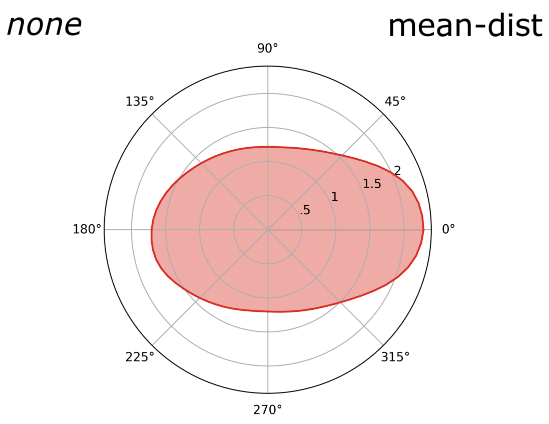
In short, this benchmark looks at how well the classes (including the rotated representatives of those classes) are clustered in feature space.

Let SAME $\subseteq C$ denote the set of clouds from the same class as $c_i$, and DIFF $:= C \setminus$ SAME.

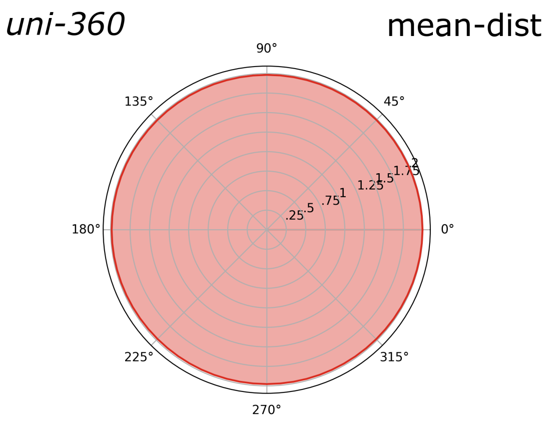
In words, the score function computes the ratio of mean feature-space distance of $R^z_\theta c_i$ to different-class objects to the mean feature-space distance of $R^z_\theta c_i$ to same-class objects:

$$s(R^z_\theta, c_i) := \frac{\frac{1}{|\mathrm{DIFF}|}\sum_{j=1}^{|\mathrm{DIFF}|} d(f(R^z_\theta c_i), f(\mathrm{DIFF}_j))}{\frac{1}{|\mathrm{SAME}|}\sum_{j=1}^{|\mathrm{SAME}|} d(f(R^z_\theta c_i), f(\mathrm{SAME}_j))} \tag{4.4}$$

$S(\theta)$ is therefore a real number in $[0, \infty)$, where **larger** values are better (different-class objects should be further away than same-class objects). Example plots of $S(\theta)$ can be found



(a) A basic PointCNN model, not robust to rotations.



(b) A PointCNN model trained with data augmentation, and thus robust to rotations.

Figure 4.7: Example `mean-dist` scores for two different models. The non-robust model has the highest score at 0°, and the score deteriorates away from 0°, being worst at ±90°. The score stays above 1.0, however, indicating the model is still better than random. For the robust model, the score stays approximately constant at all angles, indicating strong robustness to rotations.

| Benchmark | Range of $S(\theta)$ | Better values | $S(\theta)$ for random descriptor | $S(\theta)$ for a rot. invariant descriptor |
|---|---|---|---|---|
| `(k=1) eknn` | $[0, 1]$ | larger | $1/\|C\|$ [a] | $[0, 1]$ |
| `delta` | $[0, \infty)$ | smaller | 1.0 | 0.0 |
| `mean-dist` | $[0, \infty)$ | larger | 1.0 | $[0, \infty)$ |

Table 4.2: List of our rotation robustness benchmarks. [a]Assuming each class has the same number of examples in $C$.

in Fig. 4.7.

### 4.5.4 Summary

Table 4.2 juxtaposes the three rotation-robustness metrics. As an extra intuition for how to interpret benchmark scores, the table also states what value the benchmark would take, in expectation, for a 'random descriptor'. 'Random descriptor' here is defined as one that has a constant number of fixed descriptions and picks uniformly a random injective mapping which determines how it assigns descriptions to objects and all rotated objects (assuming only finitely many different values of $\theta$ are considered, as is the case in practice).

| Filename | Purpose |
|---|---|
| `benchmark_plotting.py` | Polar plots for rotation robustness benchmarks |
| `eval_mn40.py` | Rotation robustness benchmarks |
| `illustrate.py` | Visualising failure and success cases in `eknn` benchmark |
| `pca_plot.py` | Visualising descriptor feature space |

Table 4.3: Overview of the code written for the data augmentation study. All files written fully by the dissertation's author.

## 4.6 Implementation

Our primary development framework was Python 3 and Tensorflow 1.8 [50]. Data plots were made with matplotlib [51].

The essential code that we have authored for the project is attached with this dissertation and an overview of the files is presented in Table 4.3.

## 4.7 Results

The augmentation methods and their results are introduced in the following Sections. All models have been trained on the ModelNet40 dataset for 20k steps, which corresponds to 260 epochs. This takes about an hour on a single NVIDIA Titan V GPU. A summary of all results can be found in Table 4.4.

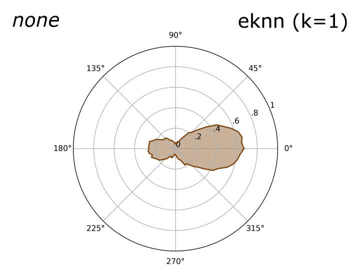
The benchmarks scores were computed using a 200-cloud subset of ModelNet40's test set.

### 4.7.1 No Augmentation (`none`)

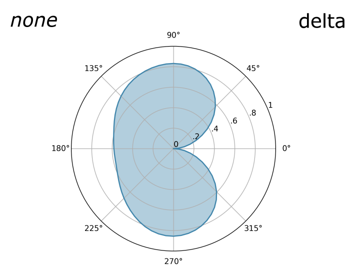
Referred to as `none`, this baseline model is trained with no data augmentation. Its scores on the three rotation robustness benchmarks are pictured in Fig. 4.8.

The model is demonstrated to not be robust to rotation, as the `delta` scores are far from 0.0 and the `eknn` and `mean-dist` scores vary greatly for different angles.

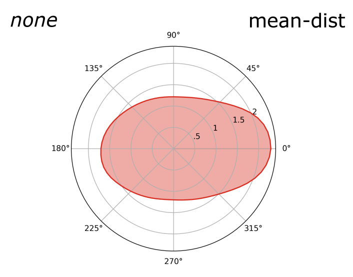
The scores of all benchmarks are worst at $\pm 90°$ angles. Our explanation is that some of the ModelNet40 models are approximately symmetric to 180° rotations about the $z$-axis, which



(a) The `eknn` scores.



(b) The `delta` scores.



(c) The `mean-dist` scores.

Figure 4.8: Rotation robustness benchmark scores for the PointCNN model '`none`'.

makes the scores better closer to 180°. For example, a guitar model has a relatively similar shape when rotated by 180° about its longest axis (along the guitar neck).

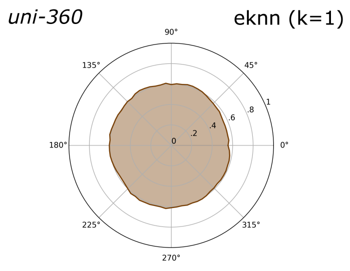
The `eknn` score of about 0.6 at 0° indicates that the model is capable of obtaining at least 60% classification accuracy. The score drops drastically to about 0.1, when the inputs are rotated by 90° compared to the training set.

The `delta` score reaches 0.8 for 90° rotations. That means the distance in feature space between a model and its 90°-rotated version is only 20% lower than the distance between that rotated model and the entire dataset on average. The score being lower than 1.0 suggests that the model performs better than a random descriptor, but the score being so high indicates low robustness to rotations.

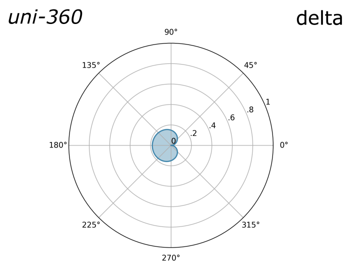
### 4.7.2 Uniform sampling (`uni-360`)

This model, referred to as `uni-360`, was augmented with $z$-rotations sampled uniformly from $[-\pi, \pi]$ for each cloud used in training. The model's scores on the three rotation robustness benchmarks are pictured in Fig. 4.9.

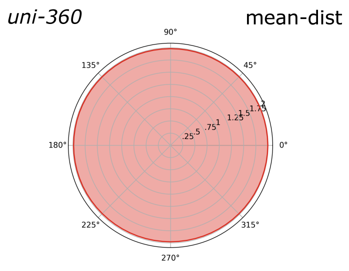
The scores prove that augmentation helped make the model robust to $z$-rotations. The `eknn` score is approximately constant for all angles, meaning that inputs are, on average, equally likely to be classified correctly, no matter what rotation may be applied to them. The score is contained in the range $[0.56, 0.62]$ for all angles. Notably, this is significantly lower than the 0.67 score at angle 0° for the non-augmented model. This shows there is a trade-off between



(a) The `eknn` scores.



(b) The `delta` scores.



(c) The `mean-dist` scores.

Figure 4.9: Rotation robustness benchmark scores for the PointCNN model '`uni-360`'.

robustness and peak accuracy.

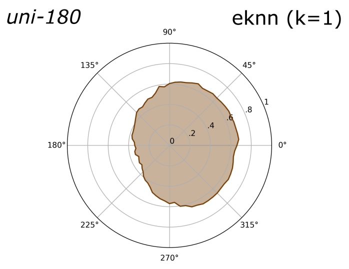
The `delta` score reaches its worst value of 0.18 at 180°. It is interesting to see that the 90° rotation is no longer the worst case – the score is worst at 180°. The `delta` score being non-zero proves that the model is not exactly rotation-invariant. At the same time, the scores are much lower than for the non-augmented model, indicating high robustness of the descriptor to $z$-rotations.

The `mean-dist` score is approximately equal for all rotations, at the value of approximately 2.0. This tells us that, even though the rotations do affect the descriptions (as shown by the `delta` score), the rotated inputs from a given class still belong to the class' cluster equally strongly as their non-rotated versions.

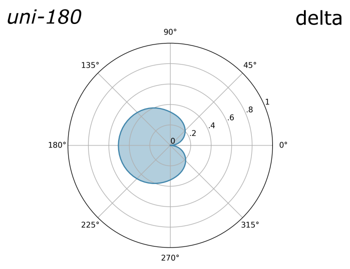
When investigating the `mean-dist` plots of descriptors, the level of their uniformity over all angles informs us about the rotation robustness of the model — as long as the scores are well above 1.0 (the random descriptor score).

The absolute value of the `mean-dist` score itself indicates the strength of separation of class clusters in the feature space – the score is conceptually closely correlated with the objective that the triplet and pairwise losses are designed to optimise.

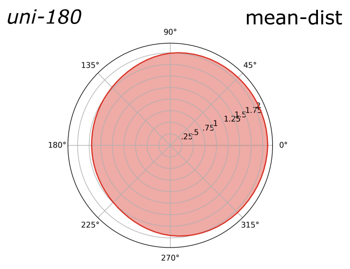
A higher `mean-dist` score is desirable and the range of scores has no theoretical upper bound. However, it is important to note that a **perfect** classifier (`eknn` score of 1.0) could have a finite `mean-dist` score and – conversely – a sequence of descriptors could have unbounded



(a) The `eknn` scores.



(b) The `delta` scores.



(c) The `mean-dist` scores.

Figure 4.10: Rotation robustness benchmark scores for the PointCNN model '`uni-180`'.

`mean-dist` scores at 0° but `eknn` scores at 0° bounded from above by a constant value strictly less than 100% (because the clusters can be very far apart but have a constant percentage of 'impostor' points from other classes).

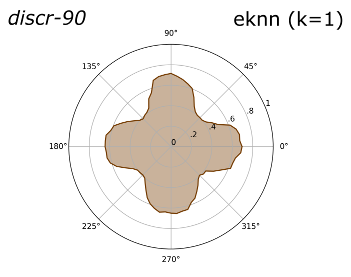
### 4.7.3 Uniform sampling on half the domain (`uni-180`)

This model, referred to as `uni-180`, was augmented with $z$-rotations sampled uniformly from $[-\pi/2, \pi/2]$ for each cloud used in training. That is, only half of the space of $z$-rotations was used. The model's scores on the three rotation robustness benchmarks are pictured in Fig. 4.10.

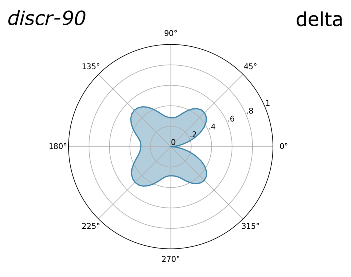
On the `eknn` benchmark, this model outperforms `uni-360` at 0° rotation, and matches it at 90° rotation, while performing much worse in the range of rotations not used during augmentation, i.e. $[\pi/2, 3\pi/2]$. This further demonstrates the trade-off (first noted in the discussion of `uni-360`) between peak `eknn` scores and their uniformity over all rotations.

The `delta` score is consistently worse than for the `uni-360` model, at all angles. That indicates the descriptor is less robust to even the small rotations, when the range used for augmentation is limited.

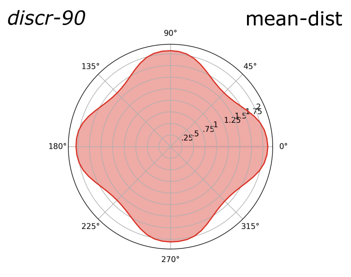
The `mean-dist` scores follow those of `uni-360` in the augmentation range $[-\pi/2, \pi/2]$ and drop off elsewhere.



(a) The `eknn` scores.

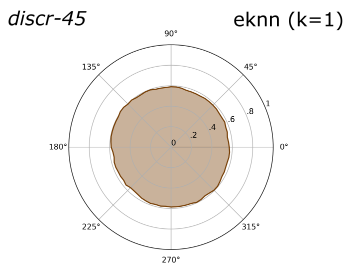


(b) The `delta` scores.

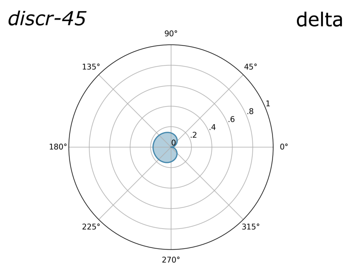


(c) The `mean-dist` scores.

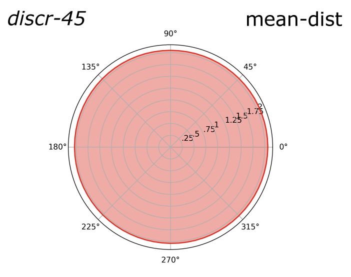
Figure 4.11: Rotation robustness benchmark scores for the PointCNN model ‘`discr-90`’.



(a) The `eknn` scores.



(b) The `delta` scores.



(c) The `mean-dist` scores.

Figure 4.12: Rotation robustness benchmark scores for the PointCNN model ‘`discr-45`’.

### 4.7.4 Discrete sampling (`discr-90`)

This model, referred to as `discr-90`, was augmented with $z$-rotations sampled randomly from a discrete list of values: $[0, \pi/2, \pi, 3\pi/2]$. The model's scores on the three rotation robustness benchmarks are pictured in Fig. 4.11.

This model's scores on all three benchmarks tell the same story – the model handles all four rotations seen during training well, but its performance deteriorates rapidly for rotations farther away from those four.

### 4.7.5 Denser discrete sampling (`discr-45`)

This model, referred to as `discr-45`, was augmented with $z$-rotations sampled randomly from a discrete list of values: $[0, \pi/4, 2\pi/4, \ldots, 7\pi/4]$. The model's scores on the three rotation robustness benchmarks are pictured in Fig. 4.12.

The effect of discretely sampling rotations every 45°, as opposed to every 90° is qualitatively different. The model's performance does not deteriorate for the in-between angles, e.g. 22.5°. The scores on all three benchmarks closely resemble the uniform sampling method `uni-360`. That shows that there is a limit as to how finely the rotation space needs to be sampled for the model to become robust to the entire space of rotations.

### 4.7.6 Pair augmentations (`pairs`)

This augmentation method is more involved, in that it doesn't simply sample a rotation independently for each cloud. Instead, in each training batch, we include two copies of every cloud – one of them rotated by a uniformly-sampled rotation (as in `uni-360`), the other one rotated by 90° *with respect to the former.*

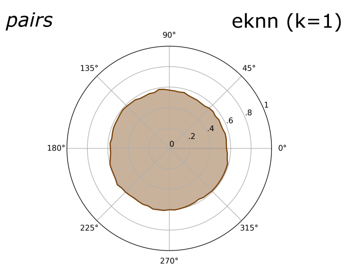
This is motivated by the idea that 90° rotations are the ones the baseline model is least robust to. Given that both copies of a cloud are in the same batch, the pairwise/triplet loss is guaranteed to receive plenty of examples to punish, as long as the model doesn't handle 90° rotations well and keeps misclassifying one of the two copies.

The pair augmentation idea is our original contribution, as far as we are aware, it has not been documented. It is generic, in that both the uniform sampling and the fixed 90° rotation can be replaced by other distributions (we treat the 90° rotation as a trivial single-valued distribution).

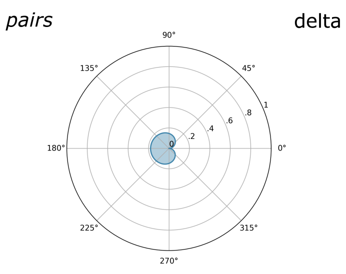
Further, note that this augmentation method is designed for losses which are computed over an entire batch, rather than being computed independently for each input of the batch and then added. In the latter setting, we would not expect the difference between pair augmentation and a per-cloud method using the same effective distribution. Technically, the ordering of the input examples enforced by the pair augmentation method would still make a difference to the training process.

The model's scores on the three rotation robustness benchmarks are pictured in Fig. 4.13.

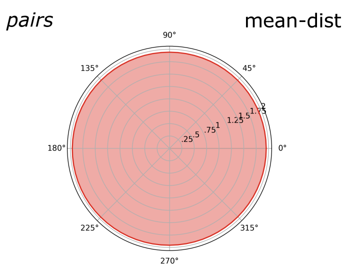
The model trained with pair augmentation has, as expected, successfully learned robustness



(a) The **`eknn`** scores.

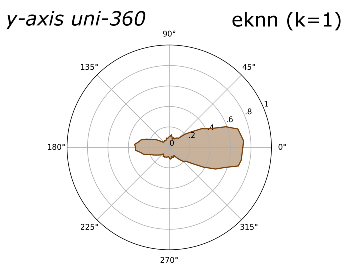


(b) The **`delta`** scores.

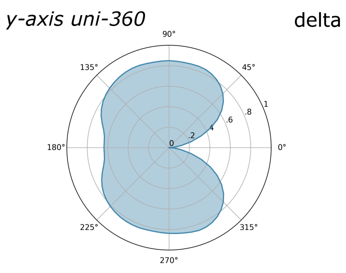


(c) The **`mean-dist`** scores.

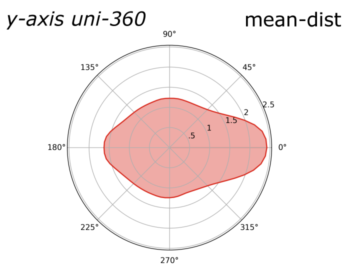
Figure 4.13: Rotation robustness benchmark scores for the PointCNN model '`pairs`'.



(a) The **`eknn`** scores.



(b) The **`delta`** scores.



(c) The **`mean-dist`** scores.

Figure 4.14: Rotation robustness benchmark scores for the PointCNN model '`y-axis`'.

to rotations, as indicated by the uniformity of its `eknn` and `mean-dist` plots, and the low `delta` scores.

The model performs very comparably to `uni-360`, but the small differences in performance are consistently in favour of `uni-360`.

### 4.7.7 Orthogonal axis (`y-axis`)

This model, referred to as `discr-45`, was augmented with rotations about the $y$ axis, sampled uniformly from $[-\pi, \pi]$. Note this is an axis orthogonal to the one used for evaluation of the robustness benchmark scores, which use $z$-rotations. The model's scores on the three rotation robustness benchmarks are pictured in Fig. 4.12.

The `y-axis` model is trained with data augmented with rotations orthogonal to those described for the benchmarks. One might expect that model to perform at least as well as the non-augmented model. Further, it is reasonable to expect it to perform better, as it effectively

| **method** | `eknn` **scores**[1] | | | | `delta` **scores** | | | | `mean-dist` **scores** | | | |
|---:|---|---|---|---|---|---|---|---|---|---|---|---|
| `none` | 0.67 | *0.06* | *0.26* | *0.04* | 0.0 | 0.83 | 0.58 | *0.86* | 2.29 | *1.22* | 1.70 | *1.20* |
| `y-axis` | **0.72** | 0.10 | 0.33 | 0.07 | 0.0 | *0.85* | *0.64* | *0.86* | **2.42** | *1.22* | *1.63* | *1.20* |
| `pairs` | *0.56* | 0.57 | 0.57 | 0.54 | 0.0 | 0.15 | **0.18** | **0.18** | *1.95* | 1.94 | 1.96 | 1.94 |
| `uni-360` | *0.56* | 0.59 | 0.60 | **0.56** | 0.0 | 0.15 | **0.18** | **0.18** | 2.00 | 1.98 | 1.99 | 1.98 |
| `uni-180` | 0.66 | 0.60 | 0.34 | 0.32 | 0.0 | 0.33 | 0.51 | 0.51 | 2.10 | 1.99 | 1.70 | 1.70 |
| `discr-90` | 0.69 | **0.71** | **0.65** | 0.39 | 0.0 | 0.28 | 0.29 | 0.50 | 2.10 | **2.07** | **2.04** | 1.63 |
| `discr-45` | 0.57 | 0.59 | 0.58 | **0.56** | 0.0 | **0.14** | **0.18** | **0.18** | 2.06 | 2.05 | **2.04** | **2.03** |

Table 4.4: Summary of scores of all evaluated augmentation methods. [1]The four entries for each benchmark are, in order: score at 0°, at 90°, at 180°, worst score. All scores are rounded to two decimal digits. Bold font indicates the **best** score among the seven models, italics indicate the *worst*.

sees more training data – over the course of training, 260 epochs of data are processed, and therefore a model with augmentation, which generates 'new' inputs every time, sees more unique data overall. That is indeed observed in the consistently superior `eknn` scores of the `y-axis` model. The `delta` and `mean-dist` scores are comparable for both models.

The profiles (shapes) of both `eknn` and `delta` score plots for the `none` and `y-axis` models are quite similar. Contrary to that, the `mean-dist` scores of the `y-axis` model deteriorate at the four angles $k \cdot \pi/2 + \pi/4, k \in \mathbb{Z}$ – a phenomenon not present for the `none` model.

### 4.7.8 Comparative Summary

The benchmark results are aggregated in Table 4.4.

`uni-360`, `pairs`, and `discr-45` are the three models that exhibit strong robustness to all $z$-rotations. The small differences in the benchmark scores suggest that `discr-45` is preferable to `uni-360`, and more so to `pairs`.

The non-augmented baseline and `y-axis` perform poorest in virtually all benchmarks, indicating that all reasonable methods of augmentation, that we evaluated, have benefited the model's robustness to rotations.

The `y-axis` model outperforms all other models with respect to the `eknn` score at 0°. This score being significantly better than the `eknn` score at 0° for the analogous `uni-360` model,

| **method** | none | y-axis | pairs | uni-360 | uni-180 | discr-90 | discr-45 |
|---|---|---|---|---|---|---|---|
| **mean** | 0.23 | 0.23 | 0.58 | 0.60 | 0.54 | 0.55 | 0.58 |
| **at** 0° | 0.67 | 0.72 | 0.56 | 0.56 | 0.66 | 0.69 | 0.57 |
| **worst** | 0.04 | 0.07 | 0.54 | 0.56 | 0.32 | 0.39 | 0.56 |

Table 4.5: Average eknn results over all angles. The results at 0° are approximately the best results of all the models. The bottom two rows give evidence that the profiles of the eknn scores among the rightmost five models vary much more than the mean.

demonstrates an anisotropy of the dataset. While the models in ModelNet40 are, in general, not all aligned, there must be an average tendency for the $z$ and $y$ axes to relate differently to symmetries of the clouds in the dataset.

### 4.7.9 General Observations

Some general observations, discussed below, can be made, based on the multitude of benchmark scores obtained.

The mean-dist score can be observed to be strongly correlated with 1/delta for the non-robust models – this property is not implied by the definitions of the scores, and thus is interesting to see. For the two robust models, the mean-dist score is approximately constant, while the delta score still varies greatly between 0.0 at 0° and approximately 0.18 at 180°.

A trade-off can be noticed between the peak eknn score and its uniformity over all angles. More robust models have lower peaks – e.g. uni-360 performs poorer at 0° than the non-augmented model. The discr-90 model outperforms all other models at 90° and 180°, while performing worse than uni-360 at in-between angles such as 45°.

The trade-off mentioned above would be expected to promote different models to have more similar average eknn score across all angles. To validate that idea, we have computed the means, presented in Table 4.5. The table demonstrates that – while the shapes of the polar plots of eknn scores of augmented models such as uni-360, uni-180, and discr-90 are very different – their mean eknn scores (averaged over all angles) are very similar – all between 0.55 and 0.60. The non-augmented model and the one augmented with rotations about an orthogonal axis both have a mean eknn score of 0.23 – much lower than the other models.

Chapter 5

# The Spherical Fractal Convolutional Neural Network (SFCNN)

## 5.1 Brief Overview of the SFCNN

What follows is an overview of the architecture. An accompanying diagram of the architecture can be seen in Fig. 5.1. All details are fully defined in later Sections of this Chapter.

1. A 3D point cloud $P$ of an arbitrary size (further denoted by $N$) is accepted as an input. It is scaled and translated so that it has mean 0 and, under that condition, fits tightly in a radius-1 sphere centred in the origin.

2. A spherical lattice with radius 1 and origin 0 is put in the same space as the input cloud. The lattice can be considered an undirected graph with finitely many vertices (the number of vertices further denoted by $V$) that come with a position in $\mathbb{R}^3$.

3. The *projection step* follows. That is, for each vertex $v$ of the lattice, a feature vector is computed based on the positions of the $k$ (a fixed hyperparameter) points of $P$ that are closest to $v$.

4. The input point cloud is not used any more after the projection step. The architecture

proceeds by processing the lattice with features at its vertices. That is referred to as the *convolutional stage*, where, at each step, new features for each vertex are computed based on its previous features, as well as the features of its neighbours in the lattice.

5. The convolutional stage is split into substages. At the end of each substage, the lattice is made coarser by dropping every other vertex. At the same time, the dimensionality of the features is increased. This is analogous to decreasing the image dimensions but increasing the number of channels in an image-processing Convolutional Neural Network (CNN). At the end of each substage, max-pooling over all lattice vertices is performed to obtain a single feature vector for the entire lattice[1].

6. Finally, those 'global' feature vectors from all substages are all concatenated and fed into a Multi-Layer Perceptron (MLP) that outputs classification logits.

## 5.2 Subdivided Regular Icosahedron as a Lattice

A key concept used in the SFCNN architecture is the spherical lattice. That lattice is made progressively coarser as the convolutional stage proceeds. We begin by capturing the abstract properties that a lattice needs to have in order to be compatible with the SFCNN. Any structure that matches the definition of the *subdivided lattice* could be plugged into the SFCNN.

**Definition.** *Let a* ***lattice*** *be a finite undirected graph with vertices in* $\mathbb{R}^3$.

**Definition.** *We define a* ***subdivided lattice*** *with* $s$ *levels of subdivision to be a sequence of* $s$ *lattices* $L_0, \ldots, L_{s-1}$, *such that* $L_i$ *is a subgraph of* $L_{i+1}$ *for all* $i$. *That means all vertices (resp. edges) of* $L_i$ *are also vertices (resp. edges) of* $L_{i+1}$.

Further, we define the exact lattice used by the authors of the original paper, which we also use in our implementation. The lattice is based on the Platonic solid known as *regular icosahedron*.

[1]The term 'max-pooling' refers to taking a point-wise maximum over the feature vectors at all vertices.

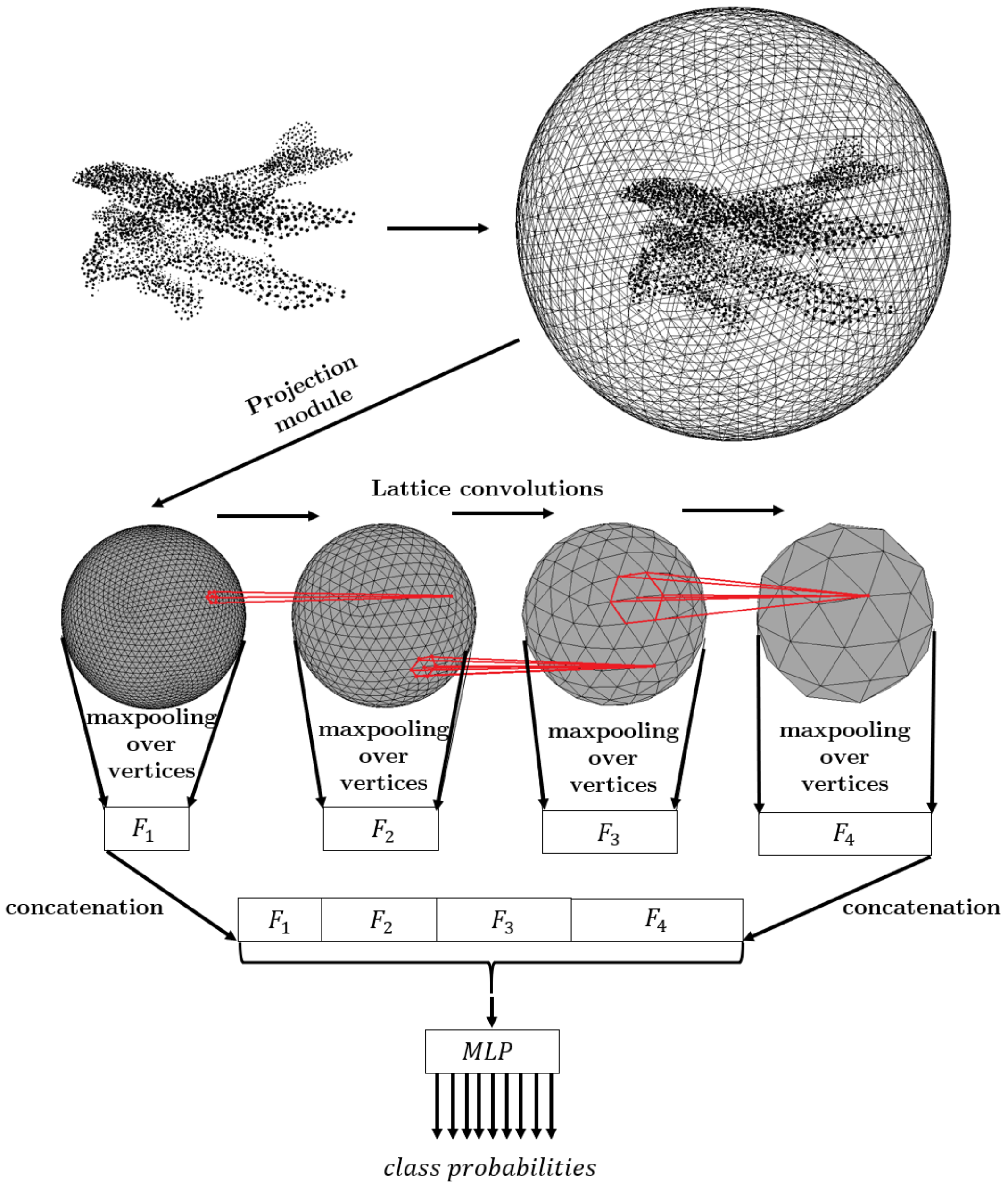


Figure 5.1: The SFCNN architecture. The input cloud is embedded inside a spherical lattice. The projection module computes a feature vector for every vertex on the initial lattice. Lattice convolutions are then performed, making the lattice progressively coarser. The lattice features are maxpooled over all its vertices at the end of each convolutional substage, and the results are concatenated, and then input into a final MLP, which returns the class probabilities corresponding to the input cloud.

The regular icosahedron is a regular polyhedron (informally, *regular* means that it looks the same from the perspective of each of its vertices, each of its edges, and each of its faces) that has 20 triangular faces. Its illustration can be found in Fig. 5.2. Its relevant properties are:

- Each of its 20 faces is a triangle.
- All 12 vertices have degree 5.
- It has a *circumsphere*, i.e. a sphere that passes through all of its vertices.

The regular icosahedron can be iteratively subdivided to create a sequence of progressively finer polyhedra[2]. Each subdivision step works as follows:

1. On each existing edge, create a new vertex in the middle of that edge.
2. For each existing face (they are all guaranteed to be triangular), create three edges between the newly created vertices, so that the original face is subdivided into four triangular faces.
3. Finally, scale all the new vertices away from the origin (the centre of the circumsphere) so that they land on the circumsphere.

This step can be applied an arbitrary number of times iteratively. The polyhedron obtained after $n$ subdivision steps will have the following properties:

- Each of its $20 \cdot 4^n$ faces is a triangle.
- 12 vertices (being the original ones) have degree 5 and all other $30 \cdot \left(\sum_{k=0}^{n-1} 4^k\right)$ vertices have degree 6.
- It has a circumsphere.

Let $I_n$ denote the regular icosahedron subdivided $n$ times, as defined above. Then the subdivided lattice that we use in our SFCNN implementation is $(I_1, I_2, I_3, I_4)$. See Fig. 5.2 for an illustration. Note that the non-subdivided icosahedron $I_0$ is not used. Rao et al. [5] note

[2]The spherical polyhedra obtained by subdividing the icosahedron, as described, are referred to as *icospheres*, e.g. in the Blender 3D editing software [52].

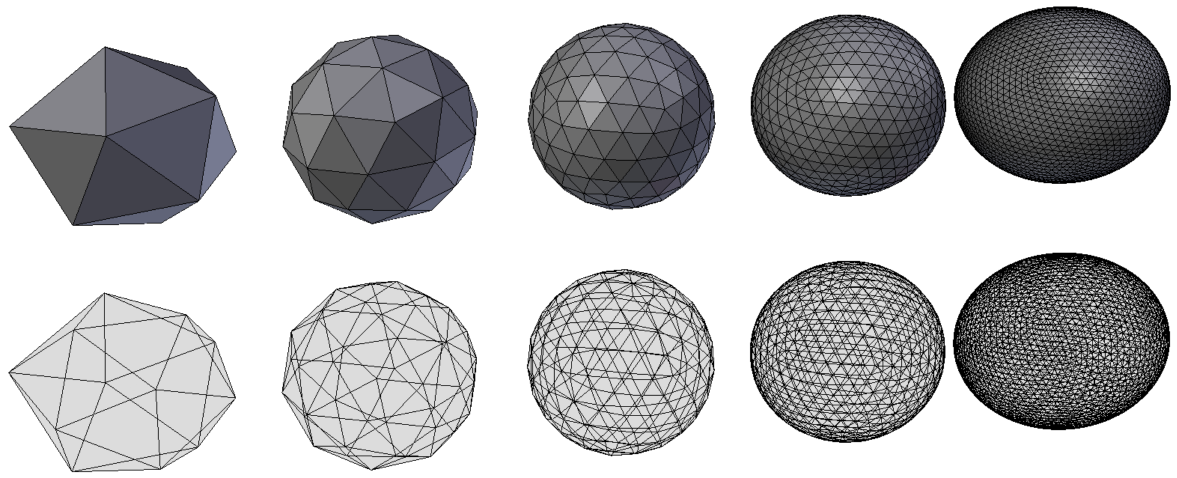


Figure 5.2: From left to right: the regular icosahedron $I_0$ and its four consecutive subdivisions $I_1, I_2, I_3, I_4$.

that it is too coarse.

## 5.3 The Projection Module

For a fixed lattice $L$ with $V$ vertices, the *projection module* consumes the input point cloud $P$ of size $N$ (i.e. with $N$ points), and produces a $d$-dimensional feature vector $f_v$ for each lattice vertex $v$. The lattice that is projected onto is the one at the highest subdivision level (i.e. the most subdivided).

Before defining the projection module, we need to introduce the reader to Non-Local layers [53].

### 5.3.1 Non-Local Layers

Non-Local layers [53] were proposed in the context of deep video processing, to provide a means for extracting long-distance relations in the data, be it in pixel space, or time. A non-local layer computes a response at a point as a weighted sum of features at all points. This is in contrast to convolutional and recurrent modules which favour local information transfer – it usually takes many layers of convolutions for the data from one corner of an image to interact with data from the opposite corner in any way.

A Non-Local layer's response has the simple generic definition, where $x$ and $y$ are the input and output vectors, respectively:

$$y_i = \frac{1}{\mathcal{C}(x)} \sum_{\forall j} f(x_i, x_j) g(x_j) \tag{5.1}$$

where $\mathcal{C}(x)$ is a normalisation factor, $g$ is a learnt transformation of the inputs, and $f$ is a learnt attention-like function. The sum is taken over **all** points $j$, as emphasised by the '$\forall$' symbol in the sum.

Wang et al. [53] propose four concrete instantiations of Eq. 5.1, all with $g$ being a simple linear transformation: $g(x_j) = W_g x_j$. The four different choices of function $f$ (together with a corresponding $\mathcal{C}$) are referred to as *Gaussian*, *Embedded Gaussian*, *Dot Product*, and *Concatenation* [53].

### 5.3.2 Implementation of the Projection Module

The projection module works in 3 stages (illustrated in Fig. 5.3):

1. For a lattice vertex $v$, the $k$ nearest neighbours of $v$ amongst the points of the cloud $P$ are selected. ($k$ is a fixed hyperparameter of the network.) Let $P_v$ denote this $k$-point subcloud of $P$, with points $P_v^{(1)}, \ldots, P_v^{(k)}$.

2. A rotation $\mathcal{R}_v$ is applied to $P_v$. It is a rotation about the origin, that maps $v$ to a globally fixed vector $u$, fixed for the purposes of this work to be $u = (0, 0, 1) \in \mathbb{R}^3$. There exists more than one such rotation; this topic is discussed in detail in Section 5.6. This rotation can be thought of as transforming the subcloud to a coordinate system corresponding to $v$'s 'point of view'.

3. The rotated subcloud $\mathcal{R}_v P_v$ is fed into a three-layer point-wise Multi-Layer Perceptron (MLP). That MLP is shared for all $v$ and also for all points in the subcloud. A Non-Local layer is also applied before the final layer of the MLP.

   The MLP* (asterisk to indicate presence of the Non-Local layer) maps each 3-dimensional point $(x, y, z)$ onto a higher-dimensional feature vector. At the end, max-pooling over all $k$ points in the subcloud is applied to obtain the feature vector $f_v$.

$$f_v := \max_{i\in\{1,\ldots,k\}} \mathrm{MLP}^* \left(\mathcal{R}_v P_v^{(i)}\right) \tag{5.2}$$

The authors of SFCNN do not mention which variant of Non-Local layer they use. However, one of the key observations made in the original paper on Non-Local networks [53], is that the choice of Non-Local layers' instantiation doesn't seem to much affect performance of models containing those layers. For the purpose of this project, we have chosen to use the *Gaussian* instantiation:

$$f(x_i, x_j) := \exp(x_i^T x_j), \qquad \mathcal{C}(x) := \sum_{\forall j} f(x_i, x_j). \tag{5.3}$$

The $\forall j$ non-locality ranges over the $k$ points of each subcloud.

## 5.4 Lattice Convolutions

### 5.4.1 The Basic Block

The purpose of lattice convolutions is to combine features of each lattice vertex with those of its neighbours. The contributions of all neighbours should be symmetrical. This motivates the following formula:

$$f'_v = \mathrm{conv}\left(\max_{i\in \mathrm{nei}(v)} (\mathrm{conv}(f_v||f_i))\right), \tag{5.4}$$

where nei$(v)$ denotes the set of lattice-neighbours of $v$, $f_v||f_i$ denotes concatenation of the two feature vectors along the channel dimension, conv denotes a convolution with kernel size 1, and the max is taken pointwise.

In words, for each neighbour $i$ of $v$, a convolution is applied to $f_v||f_i$. Vectors obtained this

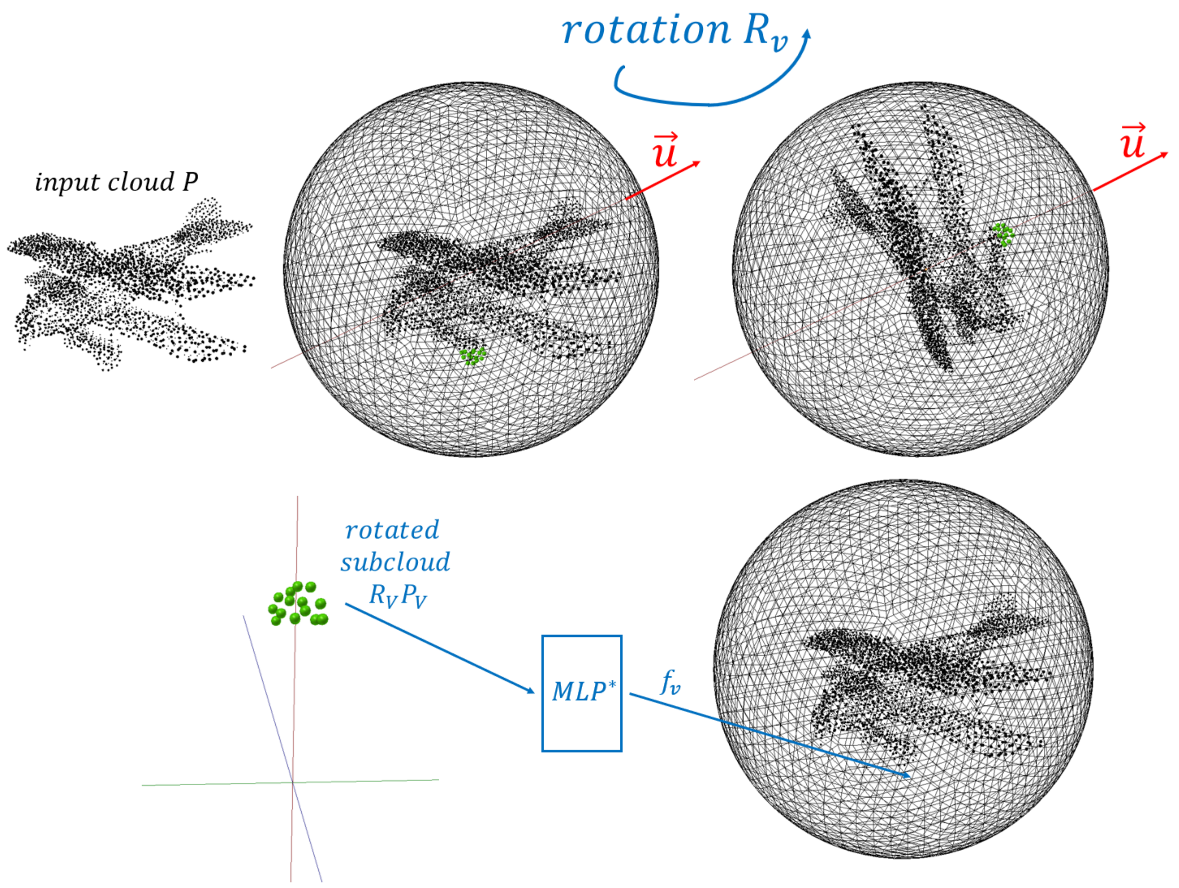


Figure 5.3: An illustration of the Projection Module.
Top row: the input cloud is embedded in a sphere. For each vertex $v$, the nearest neighbour subcloud is extracted (marked green in the image). The $\mathcal{R}_v$ rotation that aligns the vertex $v$ with the fixed vector $u$ is applied to the cloud.
Bottom row: the rotated subcloud $\mathcal{R}_v P_v$ is input into the MLP$^*$, which returns the feature vector $f_v$ for the vertex $v$.

way for all neighbours are maxpooled, and then the result undergoes a second convolution to yield the new feature vector $f'_v$.

A few common deep learning tricks are further used to enhance Eq. 5.4, see Fig. 5.4 for an illustration:

1. A ReLU activation[3] followed by a *batch normalisation* [54] step is applied before the max-pooling.

   Applying batch normalisation after a certain layer means that the $b$ outputs of that layer (where $b$ is the size of the training batch) are jointly normalised (each dimension independently) to have mean 0 and variance 1. While positive influence of batch normalisation has been empirically noticed in many different settings, the reasons for that are still a subject of investigation [55].

2. Batch normalisation is also applied after the second convolution.

3. That second batch normalisation is followed by an incoming residual connection [36], i.e. the original input $f_v$ is (pointwise) added to the current output.

   While the increasing capability to train ever deeper and wider networks has been directly responsible for the recent successes of deep learning, increasing network depth does also create some problems. A prime example is the *vanishing gradient* problem [56]. In short, the partial derivatives of loss with respect to weights of the network are computed using *chain rule*, which involves a number of gradient multiplications that grows linearly with the network's depth. Thus, in deep networks, gradients can become very prohibitively small, not allowing the parameters to change after any sensible amount of training.

   The purpose of residual connections is to introduce shallower paths in the network, which *skip* some layers. This helps alleviate the issues inherent to deep networks. Residual connections are not the only solution to the vanishing gradient problem, other approaches

[3]A commonly used activation function, defined to map negative $x$ to 0, and non-negative $x$ to $x$. Activation functions are used in neural networks to introduce non-linearity – without those, a large network composed of linear transformations would not be more expressive than a single linear transformation.

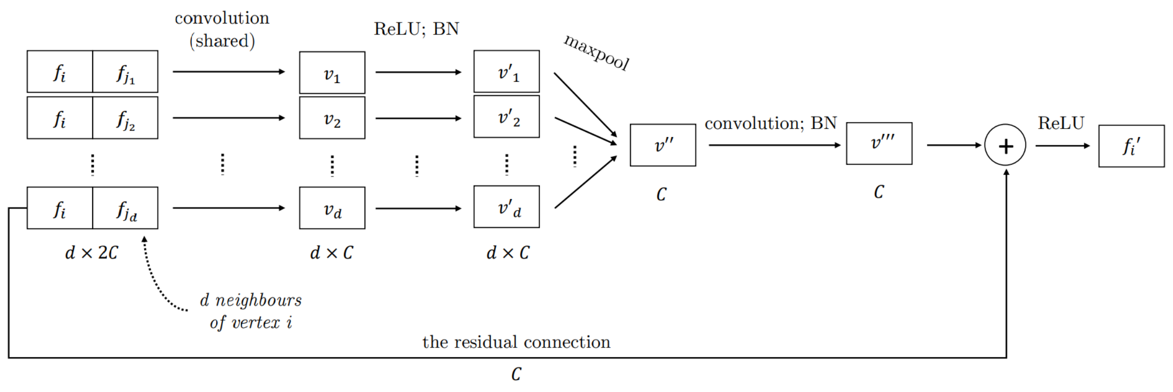


Figure 5.4: The basic block of the SFCNN's convolutional stage, as described in Section 5.4.1. A mapping of a single vertex' feature vector $f_i$ to its next value $f_i'$ is shown. That operation is performed in parallel for the entire lattice, in practice, adding an extra dimension of size $V$ (vertex count of the lattice) to all tensors. $d$ is equal to 5 or 6 for the icosahedral lattice.

including use of persistent state such as in the Long Short Term Memory (LSTM) architecture [57], or adapting the activation functions (the vanishing gradient problem being one of main reasons for the popularity of the ReLUactivation function, and decreased use of the once ubiquitous sigmoid and tanh functions).

4. The residual connection is followed by another ReLU activation.

The Eq. 5.4 with the aforementioned enhancements is further referred to as the *basic block* of the convolution stage of SFCNN. Fig. 5.4 provides an illustration.

### 5.4.2 The Convolutional Stage

The convolutional stage, i.e. the part of the SFCNN architecture that follows the projection module, is easiest understood as a sequence of *substages*.

A single *convolutional substage* consists of a number of basic blocks (see Section 5.4.1) applied sequentially, followed by dropping out the vertices from the finest subdivision level. That is, the next substage shall operate on a coarser lattice. In exchange, the next substage shall use a higher channel count, i.e. each vertex has a higher-dimensional feature vector associated with it.

A single substage $i$ uses a constant number of $C_i$ channels. The convolution $\text{conv}(f_v||f_i)$ maps $2C_i$ channels onto $C_i$ channels, while the second convolution in a basic block maps $C_i$ channels onto $C_i$ channels. Exceptions occur at boundaries between two substages, where the channel count needs to change ($C_i \to C_{i+1}$).

At the end of each substage $i$, the entire lattice is maxpooled, and the vector $F_i := \max_v f_v^{(i)}$ is stored, where $f_v^{(i)}$ denotes the feature vector at $v$ at the end of substage $i$.

The final output of the convolutional stage of SFCNN is $(F_1||F_2||\ldots||F_n)$, where $n$ is the number of convolutional substages used, and $||$ denotes concatenation. Its dimensionality is $C_1 + C_2 + \ldots + C_n$.

## 5.5 Full Model

Putting it all together, the SFCNN architecture is composed of three stages performed in sequence:

1. Projection of the point cloud onto the lattice.
2. The convolutional stage.
3. An MLP that outputs classification logits.

A three-layer MLP is used in the final step, with 512, 128, and $C$ outputs in consecutive layers, where $C$ is the number of classes for the classification task at hand.

When describing the architecture, we have not specified the exact number of channels, or basic blocks used. In [5], the authors condense those into two hyperparameters, $B$ for depth, and $K$ for width. See Table 5.1 for details. Those, together with $k$ (number of points in the nearest-neighbour subclouds in the projection step) are the three hyperparameters of SFCNN. For reference, the best-performing model found by the authors of [5] used $k = 16, B = 3, K = 8$.

This concludes the presentation of the original SFCNN architecture, as introduced in [5]. What follows is an analysis and discussion of that model.

| Convolutional Substage | # Lattice Vertices | # Channels | # Basic Blocks Sequentially |
|---:|:---:|:---:|:---:|
| 1st | 2562 | $16K$ | $2B$ |
| 2nd | 642 | $32K$ | $2B$ |
| 3rd | 162 | $64K$ | $2B$ |
| 4th | 42 | $128K$ | 4 |

Table 5.1: The convolutional stage in detail.

## 5.6 Analysis of Rotation Invariance

Recall the Projection Module, as defined in Section 5.3. The $k$-point subcloud $P_v$ of the input cloud $P$ was extracted for each vertex $v$. Then, a rotation $\mathcal{R}_v$ was applied to $P_v$. We stated that $\mathcal{R}_v$ is a rotation such that $\mathcal{R}_v v = u$, where $u$ is some globally fixed vector, e.g. $(0, 0, 1) \in \mathbb{R}^3$ in our implementation. This turns out to be ambiguous. We examine the rotations $\mathcal{R}_v$ and their purpose in greater detail now.

The main purpose of using those rotations and choosing a spherical and close-to-uniform lattice, is to introduce some robustness against input rotations. Robustness is all that the model provides – it is not exactly rotation invariant, as we discuss later.

It is easiest to illustrate the motivation for the SFCNN architecture by showing a modified version of it, which is **exactly** rotation-invariant, up to an error introduced by *imperfection* of the lattice (we explain, what we mean by this later). As we remove the modification again, to restore the original SFCNN model, we lose the exact invariance property, but leave in place most of the mechanisms that helped achieve it — this is our justification for why the model handles rotations well.

### 5.6.1 The (Lossy) Rotation Invariant SFCNN

After applying the rotations $\mathcal{R}_v$ (step 2 in Section 5.3) to the subclouds, the original SFCNN would feed their $(x, y, z)$ coordinates into the MLP* (step 3 in Section 5.3).

An extra step that we add to the original SFCNN, in order to obtain an exactly rotation invariant model, is replacing the 3D $(x, y, z)$ coordinates of the points with 2D coordinates

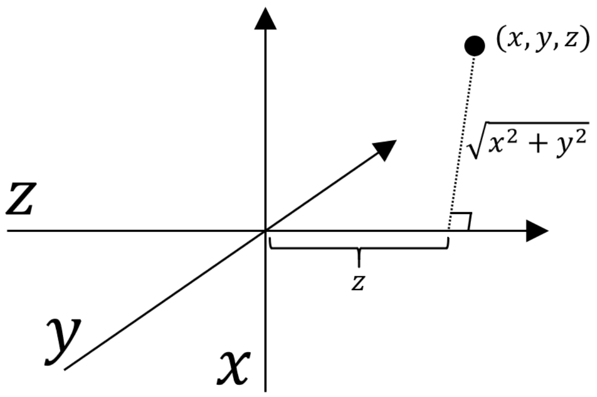


Figure 5.5: The $z$-rotation-invariant 2D feature illustrated. The two parameters $\sqrt{x^2+y^2}$ and $z$ are equal to the distance away from the $z$ axis, and distance from origin along the $z$ axis, respectively. Crucially, both are invariant to $z$-rotations.

(see Fig. 5.5 for illustration):

$$(\sqrt{x^2+y^2}, z). \tag{5.5}$$

Recall that the we have chosen $(0, 0, 1)$ to be the fixed vector $u$, referred to in the definition of rotations $\mathcal{R}_v$. The key property of the chosen representation $(\sqrt{x^2+y^2}, z)$ is that those new features are invariant to rotations about $u$.

**Remark.** *This extra step sacrifices some information, though this is, strictly speaking, not necessary. For example, the information-lossless Rigorously Rotation Invariant features introduced by Chen et al. [39] (and discussed in Section 3.3) could be used instead of* $(\sqrt{x^2+y^2}, z)$.

We now proceed to prove that the modified SFCNN is rotation invariant (up to the error introduced by imperfections of the lattice). Recall that we use the notation $R^z_\alpha$ to denote a rotation by angle $\alpha$ about the $z$-axis.

We start by proving a useful lemma about a particular unique representation of 3D rotations. Informally, the Lemma states that any rotation $\mathcal{R}_v$ that rotates $v$ onto $u$ – and there is infinitely many such rotations for each $v$ – can be decomposed into a fixed rotation $Q(v)$ which rotates $v$ onto $u$, followed by a rotation about $u$ (the $z$-axis).

**Lemma 1.** *Let $S^2$ denote the unit sphere. There exists a function $Q : S^2 \to SO(3)$, such that every rotation $\mathcal{R}_v \in SO(3)$ that rotates $v$ to $u = (0, 0, 1)$ can be represented as $R^z_\alpha \circ Q(v)$ for a unique $\alpha \in [0, 2\pi)$.*

*Proof.* We start by defining $Q$. Consider the usual spherical coordinates[4] $(\phi, \theta) \in [0, 2\pi] \times [-\pi/2, \pi/2]$, built around the global coordinate frame $(x, y, z)$, so that $x$-rotations change the longitude $\phi$.

The spherical coordinates are unique everywhere on $S^2$, other than the poles of the $x$-axis, i.e. $\theta = \pm\pi/2$, where any longitude $\phi$ maps to the same point. Written formally, that well-known fact reads:

**Claim.** *For any point $p$ on the unit sphere, other than the poles* $\mathrm{pole}_N := (1, 0, 0)$ *and* $\mathrm{pole}_S := (-1, 0, 0)$*, there is a unique rotation $\kappa(p)$ that maps $u = (0, 0, 1)$ onto $p$ and is of the form:*

$$\kappa(p) = R^{(\Lambda y)}_\theta \circ \Lambda, \qquad \text{where } \Lambda := R^x_\theta, \quad \theta \in (-\pi/2, \pi/2), \quad \phi \in [0, 2\pi). \tag{5.6}$$

In words, $\Lambda$ is an $x$-rotation by some $\phi$, that should be pictured as acting on both the point $p$ and the $y$-axis. After $p$ is rotated by $\Lambda$, it is then rotated by the (latitudal) angle $\theta \in (-\pi/2, \pi/2)$ about the $\Lambda$-rotated $y$-axis obtained from the $y$-axis by $\phi$. $\Lambda$ followed by the latitudal rotation is the definition of $\kappa(p)$.

Additionally, let $\kappa(\mathrm{pole}_N) := R^y_{\pi/2}$ and $\kappa(\mathrm{pole}_S) := R^y_{-\pi/2}$. This choice is arbitrary, we could have used any spherical-coordinate-type rotation with latitude $\pm\pi/2$.

We define $Q(p) := \kappa_p^{-1}$ for every $p \in S^2$.

To prove that every rotation $\mathcal{R}_v$ that rotates $v$ to $u$ can be represented as $R^z_\alpha \circ Q(v)$ in **at most** one way, note that there must be $R^z_\alpha = \mathcal{R}_v \circ Q(v)^{-1}$ and thus $\alpha$ is unique.

[4]With radius $r$ implicitly fixed to 1.

To prove that every rotation $\mathcal{R}_v$ that rotates $v$ to $u$ can be represented as $R^z_\alpha \circ Q(v)$ in **at least** one way, we note that the rotation $\mathcal{R}_v \circ Q(v)^{-1}$ must be a rotation about the $u$-axis, as it preserves the origin (because both $\mathcal{R}_v$ and $Q(v)$ do) and it preserves $u$:

$$(\mathcal{R}_v \circ Q(v)^{-1})u = \mathcal{R}_v(Q(v)^{-1}u) = \mathcal{R}_v(v) = u.$$

□

Having proven Lemma 1, we now proceed to prove invariance of the modified SFCNN.

Consider an input cloud $P$, an arbitrary rotation $R \in \mathrm{SO}(3)$, and the rotated cloud $RP$. We want to show that the lattice of feature vectors will be the same for both $P$ and $RP$, up to a rotation (equal to $R$). That would result in the SFCNN producing the same outputs for $P$ and $RP$, because of the max-pooling performed over all lattice vertices (see Section 5.4.2), as the max-pooling provides invariance to lattice rotations (i.e. only the lattice connectivity matters, and the positions of its vertices in space, or any kind of ordering assigned to them, does not).

Consider an arbitrary lattice vertex $v$, and the lattice vertex $Rv$. We want to show that $f_v(P) = f_{Rv}(RP)$, i.e. the feature vector at $v$ for the input cloud $P$ is the same as the feature vector at $Rv$ for the input cloud $RP$.

**Remark.** *This is where the aforementioned 'lattice imperfection' comes into play – for most rotations $R$ there would not actually exist a lattice vertex exactly at $Rv$, as the lattice has finitely many vertices. On top of that, the lattice is not completely uniform, because of the 12 degree-5 vertices it has amongst all its other degree-6 vertices. Thus the vertices don't all play a symmetric role in the convolutional stage.*

Because the MLP* is shared for all lattice vertices, it is sufficient to prove that the features fed into the MLP* are the same, i.e. $f_v(P) = f_{Rv}(RP)$.

Note that the $k$-point nearest-neighbour subcloud $P_v$ consists of the same points as the $k$-point subcloud $(RP)_{Rv}$. Formally: $R(P_v) = (RP)_{Rv}$. This is easy to see – as the lattice and cloud are rotated together by $R$, the distances between lattice vertices and input points do not change.

Recall that we have chosen to convert the $(x, y, z)$ positions to features that are invariant to rotations about $u$. Therefore, it is sufficient to prove that: $(a)$ applying $\mathcal{R}_v$ to the whole cloud $P$, and $(b)$ applying $\mathcal{R}_{Rv}$ to the rotated cloud $RP$, both give outputs (rotated subclouds) identical up to a $u$-rotation. Recalling that $u = (0, 0, 1)$ is aligned with the $z$-axis, the succinct way to restate this is Lemma 2 below. Proving it shall conclude the proof of invariance of this modified SFCNN.

**Lemma 2.** *$\mathcal{R}_{Rv} \circ R = R^z_\gamma \circ \mathcal{R}_v$, for some $\gamma \in [0, 2\pi)$.*

*Proof.* Let $\alpha, \beta \in [0, 2\pi)$ be such that $\mathcal{R}_v = R^z_\alpha \circ Q(v)$ and $\mathcal{R}_{Rv} = R^z_\beta \circ Q(Rv)$, the form discussed in Lemma 1. Because $Q(Rv)$ rotates $Rv$ onto $u$, it follows that $Q(Rv) \circ R$ rotates $v$ onto $u$. Therefore, by Lemma 1, there exists a unique $\delta$ such that:

$$Q(Rv) \circ R = R^z_\delta \circ Q(v) \tag{5.7}$$

We finish the proof of this Lemma as follows:

$$\begin{aligned}
\mathcal{R}_{Rv} \circ R &= (R^z_\beta \circ Q(Rv)) \circ R && \text{(def. of } \mathcal{R}_{Rv}) \\
&= R^z_\beta \circ (Q(Rv) \circ R) && \text{(associativity)} \\
&= R^z_\beta \circ (R^z_\delta \circ Q(v)) && \text{(by Eq. 5.7)} \\
&= R^z_{\beta+\delta-\alpha} \circ (R^z_\alpha \circ Q(v)) && \text{(algebra of } z\text{-rotations)} \\
&= R^z_{\beta+\delta-\alpha} \circ \mathcal{R}_v && \text{(def. of } \mathcal{R}_v) \\
&= R^z_\gamma \circ \mathcal{R}_v. && \text{(setting } \gamma := \beta + \delta - \alpha)
\end{aligned}$$

The uniqueness of $\gamma$ is implied by the uniqueness of $\alpha, \beta, \delta$, guaranteed by Lemma 1. $\square$

We have implemented the rotation-invariant version of SFCNN and confirmed a much higher invariance (presumably only inexact due to the imperfection of the lattice). The experiment is described in Chapter 6.

### 5.6.2 Impossibility of Invariance in the Basic SFCNN

In Section 5.6.1, we have proven that one can obtain rotation invariance by replacing the $(x, y, z)$ point features, fed into the MLP in the projection module, with 2D features invariant to $z$-rotations.

We have also seen that there is infinitely many choices of the rotation $\mathcal{R}_v$, if the only constraint is that $\mathcal{R}_v v = u$. The original paper makes the concrete choice of $\mathcal{R}_v$ by specifying a closed algebraic formula for $\mathcal{R}_v$ as a function of $u$ and $v$, derived from the Rodrigues' formula:

$$\mathcal{R}_v := 2\frac{(v+u)^T(v+u)}{(v+u)(v+u)^T} - I. \tag{5.8}$$

To conclude our discussion of SFCNN's rotation invariance, we prove below that there is no way to choose the rotations $\mathcal{R}_v$, so that the SFCNN is invariant to all SO(3) rotations. That would require that the subclouds fed into the MLP*s in the projection step are themselves invariant, which would in turn require that $\mathcal{R}_{Rv} \cdot R = \mathcal{R}_v$ for all $R \in \mathrm{SO}(3)$ and $v$. We present a quick proof by contradiction:

$$\begin{aligned}
&\forall R, v.\ \mathcal{R}_{Rv} \cdot R = \mathcal{R}_v \iff \\
&\forall R, v.\ \mathcal{R}_{R^{-1}v} = \mathcal{R}_v \cdot R \iff \\
&\forall R, v.\ \mathcal{R}_v^{-1} \cdot \mathcal{R}_{R^{-1}v} = R \qquad (\star)
\end{aligned}$$

Where the last statement is impossible, because (not only for some $v$, but even for all $v$) there exist rotations $R_1 \neq R_2 \in \mathrm{SO}(3)$, such that $R_1^{-1}v = R_2^{-1}v$. (Those are simply the rotations about the $v$-axis.) This quickly gives a contradiction:

$$R_2 \overset{(\star)}{=} \mathcal{R}_v^{-1} \cdot \mathcal{R}_{R_2^{-1}v} = \mathcal{R}_v^{-1} \cdot \mathcal{R}_{R_1^{-1}v} \overset{(\star)}{=} R_1 \neq R_2.$$

### 5.6.3 Intuitive Summary

The rotations $\mathcal{R}_v$ are applied so that each lattice vertex sees the subcloud from 'its point of view'. For example, when describing an airplane, the lattice vertex near the tip of the plane will always receive the same point cloud, independent of the global orientation of the plane. The concrete lattice vertex will be a different one depending on the plane's orientation, but that does not matter, as the lattice vertices all contribute symmetrically to the SFCNN's final output.

The rotations $\mathcal{R}_v$ seem to help, but do not provide full invariance because they only eliminate two degrees of freedom: the vertex $v$ may see the same subcloud of $P$ rotated differently about the 'line of sight' axis, compared to what the vertex $Rv$ sees for the rotated input $RP$.

Removing one degree of freedom by replacing the 3D features $(x, y, z)$ of the subcloud points with 2D features is a simple way to obtain a provably (approximately) rotation-invariant architecture.

## 5.7 Quantitative Analysis of the Projection Step

The projection step begins by extracting nearest-neighbour subclouds $P_v$ of the input cloud $P$. Points that do not belong to any of the $P_v$ subclouds make no contribution to the output of the SFCNN.

The points ignored by the model are ones that are not the among the nearest points to any lattice vertex. For concave surfaces, the points in the concavities are most susceptible to being dropped out – and that in order changes the shape effectively processed by the SFCNN.

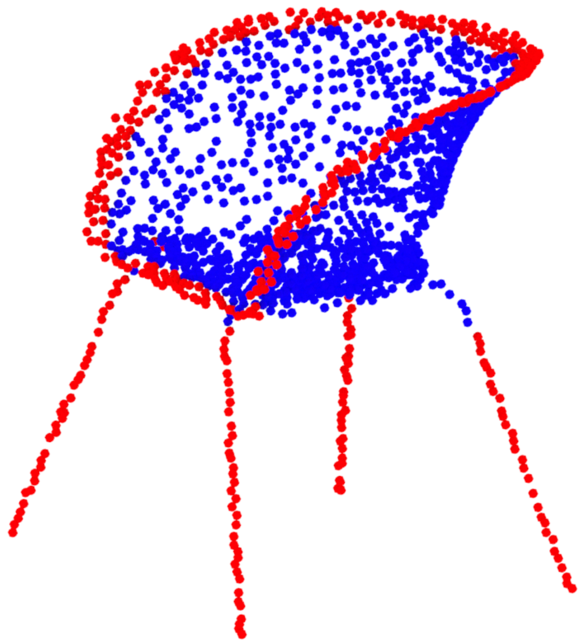


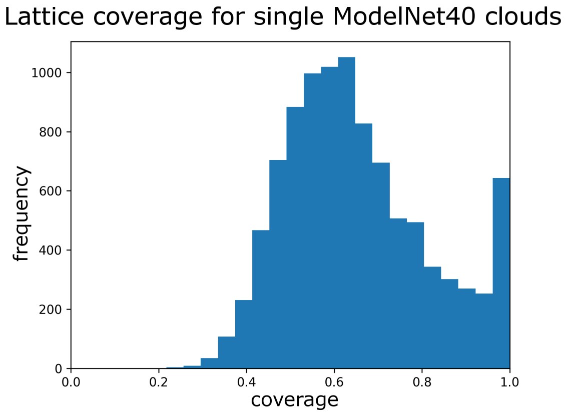
Figure 5.6: An example cloud from the ModelNet40 dataset (`chair_0083.ply`). In red: vertices that contribute to the SFCNN output. In blue: vertices that **do not** contribute to the output.

To understand the extent of this effect in a real dataset, we checked what percentage of the points contribute to the output of the SFCNN, across ModelNet40's training set. We shall refer to that percentage as the *coverage* of a point cloud. A histogram can be seen in Fig. 5.7a.

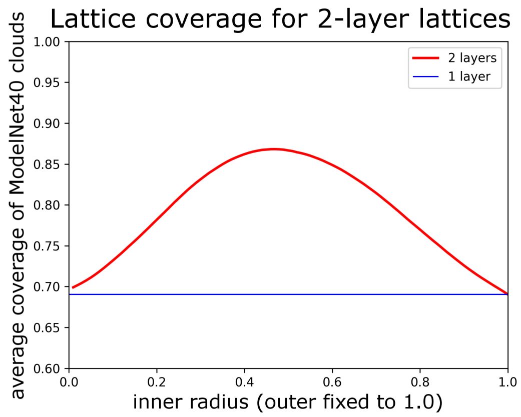
Convex surfaces can be proven to not be subject to that issue. Points may still be dropped out in practice, but only due to the finite nature of the lattice. This is expressed rigorously in the following Lemma with an informal proof:

**Lemma 3.** *For any point $p$ on a convex surface $S$ that is contained in the interior of the unit sphere, there exists a point $v$ on the unit sphere, such that $|p - v| = \min_{s \in S} |s - v|$, i.e. $p$ is the nearest point to $v$ on $S$.*

*Proof.* Consider the plane $T$ tangent to $p$. As $S$ is convex, all of $S$ is on one side of $T$. Project a ray $r$ going out of the surface $S$ at point $p$, in the direction perpendicular to $T$. $r$ intersects the unit sphere at exactly one point $v$.



(a) Coverage of the default SFCNN lattice for point clouds in ModelNet40's training set. The average is 69.0%.



(b) The average coverage of the ModelNet40 training clouds by 2-layer lattices, as a function of the inner radius (while the outer radius is kept at 1.0).

Figure 5.7: Coverage analysis for multi-layer icosahedral lattices.

Then any segment $\overline{vs}$ for any point $s$ on the surface must intersect the plane $T$ and thus be at least as long as the segment $\overline{vp}$. $\square$

## 5.8 Original Modifications to SFCNN

In addition to the rotation-invariant version of SFCNN discussed in Section 5.6.1, we introduce two modifications of SFCNN that address the low lattice coverage, as discussed in Section 5.7.

Both modifications, referred to as *3d* and *onion*, keep all the key ideas of the basic SFCNN. They both introduce lattice points inside the unit sphere, in order to capture the input points usually dropped by the basic SFCNN (which tend to lie close to the origin). The difference between the *3d* and *onion* models lies in how the features from lattice points at different depths are aggregated.

Both approaches introduce the new lattice points by reusing the subdivided icosahedra at smaller scales. This choice was made to preserve the (approximate) uniformity and (approximate) rotation invariance guaranteed by the subdivided icosahedra – the reason why that lattice shape was chosen over others in the first place by the authors of SFCNN [5].

### 5.8.1 The onion SFCNN

The *onion* approach introduces multiple identical lattices of different radiuses. The projection step is performed as usual for each of the lattices separately. Then the feature vectors are concatenated over all lattices and assigned to the primary lattice, while all other lattices are discarded. The architecture proceeds as usual from there.

#### 5.8.1.1 Further Work

Having identical layers of different radii stands in contradiction with the uniformity goals that the SFCNN tries to accomplish by choosing to work with the icosahedral lattice. The inner layers cover the 3D space more densely, as the same number of vertices is compressed into a smaller surface. The way to fix this is to have smaller layers be subdivided less finely.

For simplicity, let us assume we have two layers, where the inner one has been subdivided one fewer time. How do we aggregate the information from the two layers? Two natural solutions, not evaluated in this project, are:

1. Concatenate the features from the inner layer after performing one substage of the convolution stage (after which the outer layer becomes as coarse as the inner one has been to begin with).

2. Concatenate the features from the inner layer to the corresponding vertices in the outer layer, and for the high-subdivision vertices in the outer layer, instead use the averaged feature vectors from the two adjacent low-subdivision vertices. This can be generalised to when the difference in the number of subdivisions between the two layers is higher than 1 – the average will just have to be taken over more than two vertices.

### 5.8.2 The 3d SFCNN

The *3d* approach introduces a **single** '3d' lattice with radially-directed edges in addition to the original surface-directed edges.

The lattice we use is composed of identical copies of the original icosahedral lattice ('layers'), with edges connecting corresponding vertices of any pair of adjacent layers. The subdivision levels are set as in the original lattice, i.e. the progression from fine to coarse occurs only inside each layer, and not in the radial dimension.

What we mean by ‘progression from fine to coarse in the radial dimension’ is the idea of every other icosahedral layer being dropped out at the end of each substage. We choose not to do this because having too many layers would prohibitively increase the size of the model in memory. At 2 layers, we were already limited to using a batch size of 8 for training.

#### 5.8.2.1 Further Work

The same uniformity issue can be raised as for the *onion* SFCNN (see Section 5.8.1.1). We have come up with two ways to combine adjacent layers of different subdivision levels, which roughly correspond to the two solutions suggested for the onion SFCNN:

1. The coarser inner layer could have edges going only to the corresponding vertices in the finer outer layer.

2. The coarser inner layer’s vertex $v$ could have edges going to the corresponding $v'$ in the outer layer, but also all vertices $u$ of the finer outer layer, for which $v'$ is the nearest vertex at its subdivision level. Edges of the latter kind would gradually disappear as the whole lattice undergoes sparsification.

Again, we leave evaluating those ideas as future work.

### 5.8.3 Choice of Layer Radii

For both the 3d and onion extensions, we decided to use two layers, the outer with radius 1.0 (i.e. the same lattice as used in the basic SFCNN), and the inner one with radius optimised for maximising the total lattice coverage of the ModelNet40 training clouds.

We decided to limit ourselves to two layers. If we kept adding more layers, we would be observing quickly diminishing returns in terms of point cloud coverage, while the training batch size would have to be decreased significantly.

The optimisation of the inner layer's radius was a simple exhaustive search for the best inner radius, with step size 0.01. A set of 400 ModelNet40 clouds (10 per class) was used. The optimisation returns an inner layer radius of 0.47, which yields $86.8\% \pm 9.8\%$ average coverage as compared to the single-layer lattice's average coverage of $69.0\% \pm 16.5\%$. The optimisation is illustrated in Fig. 5.7b.

# Chapter 6

# Experiments and Results

## 6.1 Rotated ModelNet40 classification benchmarks

A popular choice in studies of rotation-robust models is to use not only the basic ModelNet40 dataset, but also rotated copies of its models, both for training and testing.

In the original SFCNN paper [5], the authors compute the accuracy of their classifier in three settings: $z/z$, $z/\mathrm{SO}(3)$, $\mathrm{SO}(3)/\mathrm{SO}(3)$. $z$ denotes the set of rotations about the $z$ axis, $\mathrm{SO}(3)$ denotes the set of all 3D rotations, and the $X/Y$ notation indicates that the set $X$ was used to augment the training set of clouds, while the set $Y$ was used to augment the test set.

It is straightforward to see that $z/\mathrm{SO}(3)$ is an indicator of rotation-robustness, as it tests a model on data rotated in an unseen way, compared to data in the training distribution. Alternatively, one could look at metrics such as $\mathrm{id}/\mathrm{SO}(3)$ or $\mathrm{id}/z$, where id (identity) represents no augmentation. We choose to stick with the metrics used in [5].

Models robust to rotations are also more likely to perform well at the $z/z$ and $\mathrm{SO}(3)/\mathrm{SO}(3)$ benchmarks, because the test-time rotations are still not exactly ones that have been previously seen. Also, robustness or invariance to rotations effectively shrinks the input space, which generally makes the learning easier.

The $z$-rotations are generated by sampling the rotation angle uniformly from $[0, 2\pi)$. For the SO(3) rotations, three angles are sampled, $\alpha, \beta, \gamma$, and the final rotation is taken to be $R^z_\gamma \circ R^y_\beta \circ R^x_\alpha$, where $R^x_\alpha$ denotes the rotation about the $x$-axis by angle $\alpha$, etc. Such a representation of rotations is known as *extrinsic Euler Angles*.

## 6.2 Hyperparameter Search for the SFCNN

There is a multitude of hyperparameters to be set when training an SFCNN model, both for the optimiser – we used the Adam optimiser [26], following the original paper [5] – and for the SFCNN model itself. In the original SFCNN paper, Rao et al. provide values they had used for most of the hyperparameters.

To determine the best hyperparameter values, we have implemented and run a *partial* grid search. *Partial* indicates that, if $D_i$ denotes the finite set of values we wanted to try for the $i^{\text{th}}$ hyperparameter, then our search has only covered a subset of the Cartesian product $\prod_i D_i$. Smaller groups of related hyperparameters (e.g. base learning rate and learning rate decay) would be subjected to an exhaustive grid search and then fixed, whereafter further hyperparameters would be optimised. Every $D_i$ had each of its values evaluated at least once.

The hyperparameter values were selected by optimising the $z/z$ classification accuracy (see Section 6.1 for meaning of $z/z$). Each model was trained for a fixed time duration (2 hours). Training each model until convergence would have been preferable but was not feasible for this project (as was the case with a complete grid search, as opposed to partial).

Our hyperparameter optimisation process and outcomes are summarised in Table 6.1. We have opted to always use largest batch size allowed by our hardware, which was 32 for the small model ($K = 4, B = 2$) and 12 for the large model ($K = 8, B = 3$).

## 6.3 Implementation Tools and Details

Our primary development framework was Python 3 and Tensorflow 1.8 [50].

| **name** | **meaning** | **reported** | **used** | **grid-search tested** |
|---:|---:|---:|---:|---:|
| `K` | SFCNN width | $4^b, 8^t$ | 4 | 8 |
| `B` | SFCNN depth | $2^b, 3^t$ | 2 | 3 |
| `k` | SFCNN subcloud size | 16 | 16 | — |
| non-local layer | instantiation of n.l. layer | — | *Gaussian* | — |
| `epsilon` | Adam hyperparam[1] | — | $10^{-4}$ | $10^{-3}, 10^{-2}$ |
| `lr_base` | initial learning rate | $10^{-3}$ | $10^{-4}$ | $10^{-7}, 10^{-6}, 10^{-5}, 10^{-3}$ |
| `lr_min` | minimum learning rate | — | $10^{-8}$ | — |
| `lr_decay_rate` | learning rate decay rate | 0.8 | 0.9 | $0.6, 0.8$ |
| `lr_decay_steps` | steps between l.r. decays | $20^e$ | 20 | — |
| `weight_decay` | Adam hyperparam[2] | $10^{-5}$ | $10^{-5}$ | $10^{-4}$ |
| `jitter` | scale of input noise | — | 0.0 | $0.01, 0.05$ |

Table 6.1: List of hyperparameters for SFCNN training. 'reported' indicates what value, if any, was reported to be used in the original paper [5]. 'used' indicates the value that we used for our evaluation. 'grid-search tested' indicates the set of values we have tried in our hyperparameter grid search, other than the value in the 'used' column.

[b,t] The [b] and [t] annotations, next to reported values for `K` and `B`, correspond to 'baseline' and 'top-performing', respectively.

[e] The value of `lr_decay_steps` is set in epochs.

[1] The *epsilon* is used by Adam to avoid division by zero in its formula for weight update. Its magnitude affects the rate of weight updates.

[2] The *weight decay* parameter controls the strength of $L^2$ regularisation, a standard deep-learning method that helps avoiding overfitting. [58]

| Filename | Purpose |
|---:|---|
| `sfcnn.py` | The SFCNN model definition |
| `subdivided_lattice.py` | SFCNN model – helper methods related to the lattice |
| `setting_baseline.py`[1] | An example SFCNN configuration |
| `lattice_coverage.py` | Lattice coverage computation |
| `create_jobs.py` | Hyperparameter grid search |
| `create_setting.py` | Hyperparameter grid search |
| `grid_search.py` | Hyperparameter grid search |
| `subdivided_lattice_test.py` | Tests for debugging the SFCNN implementation |

Table 6.2: Overview of the code written for the SFCNN study. All files written fully by the dissertation's author, [1]except `setting_baseline.py` adapted from pre-existing PointCNN code.

We have implemented the SFCNN model entirely from scratch, based on the paper by Rao et al. [5], as we have not succeeded at obtaining a reference implementation from the original authors. We have adapted pre-existing PointCNN training and evaluation code (from our research group's repository) for use with the SFCNN.

The essential code that we have authored for the project is attached with this dissertation and an overview of the files is presented in Table 6.2.

## 6.4 Results

We trained our implementation of SFCNN on the ModelNet40 dataset, along with the *onion* and *3d* versions of it. The model with stronger rotation invariance described in Section 5.6.1, referred to as *invariant*, was also trained.

The Adam optimiser was used with cross-entropy loss. The hyperparameter values listed in the 'used' column of Table 6.1 were applied for all models. Each of the SFCNN models has approximately 4.4M parameters, and takes about 180k steps to converge, which corresponds to 585 epochs. This takes about 14 hours on a single NVIDIA Titan V GPU.

Training the basic small model with batch size of 32, or the *onion*/*3d* model with batch size 16, takes under 12GB of GPU RAM. The memory bottleneck comes from the size of the most subdivided lattice – for the *onion*/*3d* models, the lattice is twice as big, and thus the batch

| **model** | $z/z$ | SO(3)/SO(3) | $z$/SO(3) |
|---|---|---|---|
| SFCNN ($K = 8, B = 3$) – reported in [5] | 91.4% | 90.1% | 84.8% |
| SFCNN ($K = 4, B = 2$) – reported in [5] | 90.2% | *unknown* | 83.2% |
| SFCNN ($K = 4, B = 2$) – ours, *baseline* | 85.1% | 84.2% | 80.8% |
| SFCNN ($K = 4, B = 2$) – ours, *invariant* | **86.0%** | 84.3% | **80.9**% |
| SFCNN ($K = 4, B = 2$) – ours, *onion* | 85.3% | **84.8**% | 79.2% |
| SFCNN ($K = 4, B = 2$) – ours, *3d* | 85.0% | 83.7% | 79.2% |

Table 6.3: Classification accuracy of SFCNN variants on ModelNet40. The top two rows present values reported by Rao et al. [5]. The remaining rows are results obtained by our implementations, where $z$/SO(3) results were measured after 180k training steps, while $z/z$ and SO(3)/SO(3) results were cherry-picked after the models converged.

size needs to be decreased by a factor of two, for the model to fit in memory.

The classification accuracy of our models on ModelNet40 is summarised in Table 6.3 and compared to the values reported by Rao et al. [5].

The results obtained by our implementation of the SFCNN are below the accuracies reported by Rao et al. [5]. This is, likely, a result of architectural and hyperparameter differences, caused either by our simplifications, or the details of the original implementation missing from the paper:

- The original paper reports use of 'voting trick' to boost performance. It is unclear what exactly this refers to, and we have not been able to reproduce the 'voting trick'.
- The original paper reports use of an additional 'invertibility' loss to improve training of the projection module. Their ablation tests, attribute an improvement of about 1% to the use of that loss, which we have omitted.
- The original paper does not specify what loss was used to train the model. Possibly, combining cross-entropy loss with other losses, such as triplet or pairwise, could improve performance.
- The original paper does not specify the type of Non-Local layer used. While the paper on Non-Local Networks [53] reports that the instantiation choice does not have much effect on performance of Non-Local-Networks (composed of multiple Non-Local layers),

perhaps the choice is more influential when only a single Non-Local layer is present, as in the SFCNN.

- The original paper does not specify all training hyperparameters, for example the initial learning rate of Adam, or the Adam epsilon parameter. It is likely that a more exhaustive hyperparameter search would find a better configuration than our partial grid search did.

The *invariant* model outperforms our baseline on all benchmarks, with most significant improvement at the $z/z$ category. Performance of the *onion* and *3d* models is comparable to that of the basic SFCNN. The *onion* model slightly outperforms the baseline model on the $z/z$ and SO(3)/SO(3) benchmarks.

The scores of the *invariant* model demonstrate that its higher robustness is an advantage that outweighs the loss of information that occurs when the $(x, y, z)$ coordinates are replaced with the lossy 2D features (refer to Section 5.6.1). Furthermore, the model is confirmed to not be exactly invariant – otherwise, its scores at all benchmarks would be equal, as has been observed e.g. for the Rigorously Rotation Invariant model by Chen et al. [39].

The *3d* model consistently performs slightly worse than the baseline, even though it consumes more points from the input clouds, thanks to its added icosahedral layer. That suggests that the way in which the data is aggregated is not optimal. We note that the convolutional step treats all lattice neighbours of a vertex symmetrically – regardless of whether the edge to that neighbour is oriented radially, or perpendicularly to the sphere's radius. We conjecture that the network would benefit from introducing a separate convolution kernel for the radial connections. The cost of this modification would be an increased number of trainable parameters.

The *onion* model performing comparably to the baseline indicates that the additional information it obtains is either not useful for classification, or it is wasted due to the architecture's design.

# Chapter 7

# Review of the Project

## 7.1 Contributions

As part of the project, we have contributed the following original ideas, recalled below.

**Rotation robustness benchmarks.** We have designed three rotation robustness benchmarks for point cloud descriptors (or indeed any 3D object descriptors). Each provides a different kind of insight into how the descriptor reacts to rotations. The benchmark scores can be visualised on easy-to-read polar plots. The details can be found in Section 4.5.

**The exactly rotation-invariant version of SFCNN.** We have designed a simple modification for the SFCNN model, that makes it exactly invariant to all rotations in SO(3) (under some idealistic assumptions about the lattice). We provide details and a proof of that invariance in Section 5.6.1, and an evaluation of the model in Chapter 6.

***3d* and *onion* modifications to the SFCNN.** We have noticed that the SFCNN's output only depends on some of the input points. The phenomenon was confirmed quantitatively on the ModelNet40 dataset, and we proposed two related modifications to the model in Section 5.8. The modifications were implemented, and their performance compared to the baseline SFCNN in Chapter 6.

**Pair augmentation.** A generic rotation augmentation method, designed for losses such as triplet loss, which rate an entire input batch as a whole. Pair augmentation is described in Section 4.7.6.

We plan to open-source our implementation of SFCNN to help the scientific community.

## 7.2 Directions for Future Work

Several relevant opportunities for research, detailed below, remain unexplored by our project.

**Generalisation of augmentation methods to SO(3).** Our investigation of data augmentation was limited to $z$-axis rotations only. All the methods can be easily generalised to cover all of SO(3) instead. The rotation robustness benchmarks (Section 4.5) can also be generalised. It is not obvious whether the augmentation methods would rank the same way if generalised to SO(3).

Recalling the key difference between `discr-90` and `discr-45`, it would also be interesting to see how finely SO(3) needs to be sampled in a discrete distribution for it yield robustness comparable to that obtained using the continuous uniform distribution.

**Evaluating pair augmentation further.** The pair augmentation idea describes a general framework with two parameters: the base distribution and the distribution used to obtain the second rotation in a pair. We have only tested pair augmentation with a uniform base distribution, and a deterministic $90°$ rotation used to generate the second rotation. To determine the validity of pair augmentation with more certainty, more tests should be performed using different distribution pairs.

**More datasets and architectures.** The augmentation methods could be compared for other architectures than the PointCNN, and both the augmented models and the SFCNN could be applied to more datasets, to better understand their strengths and weaknesses.

To understand how the rotation-robustness benchmark scores translate to performance on real tasks, the PointCNN models trained with augmentation could be subjected to the same evaluation as the basic model was, testing the localisation success rate of the NSM localisation algorithm (see Section 4.1.3).

Applying the SFCNN to robotics datasets, such as The Newer College Dataset [1], where the segments to be described tend to have different shapes from those encountered in ModelNet40, would be interesting. Furthermore, other standard 3D object processing benchmarks than the ModelNet40 classification task, could also be used for evaluation of the SFCNN, such as the SHREC'17 retrieval task [59]. contest [59].

**Reproducing the SFCNN classification accuracy reported in the original paper [5].** Two elements mentioned in the original paper, but omitted in our project, should be investigated: the voting trick to boost classification accuracy, and the 'invertibility constraint' loss. Additionally, a broader hyperparameter search could be performed. Those actions should provide more certainty whether the results from [5] can be reliably reproduced.

**Further experiments on the *invariant* variant of the SFCNN.** Instead of using the lossy 2D features $(\sqrt{x^2 + y^2}, z)$ to ensure invariance, other choices of features could be evaluated, such as the lossless *Rigorously Rotation Invariant* features [39].

**Further experiments on the *onion* and *3d* modifications.** As detailed in Sections 5.8.1.1 and 5.8.2.1, implementations with coarser inner layers seem to have theoretical advantages (uniformity), and would be interesting to test.

Furthermore, models with more than two icosahedral layers could be investigated to understand the relationship between lattice coverage and model performance.

As discussed in Section 6.4, use of a separate convolutional kernel for the radially-oriented edges of the *3d* lattice, seems to be a promising idea.

Another change that could be tried is increasing the channel count of the *onion* network, as currently the network pushes the information from both lattices into the number of channels usually used for a single lattice. This is a possible reason for why the extra information captured by the *onion* network's projection module does not yield more benefit.

Finally, the *onion/3d* multi-layer design could be combined with the 2D features used by the *invariant* model. This seems promising, as both the *onion* and *invariant* modifications, on their own, showed improvements over the baseline.

# References

[1] Milad Ramezani et al. *The Newer College Dataset: Handheld LiDAR, Inertial and Vision with Ground Truth.* 2020. arXiv: `2003.05691 [cs.RO]`.

[2] Hyeong Ryeol Kam et al. “RViz: A Toolkit for Real Domain Data Visualization”. In: *Telecommun. Syst.* 60.2 (Oct. 2015), pp. 337–345. URL: `https://doi.org/10.1007/s11235-015-0034-5`.

[3] Yann LeCun et al. “Deep learning”. In: *Nature* 521.7553 (May 2015), pp. 436–444. URL: `https://doi.org/10.1038/nature14539`.

[4] Yangyan Li et al. “PointCNN: Convolution On X-Transformed Points”. In: *Advances in Neural Information Processing Systems 31.* Ed. by S. Bengio et al. Curran Associates, Inc., 2018, pp. 820–830. URL: `http://papers.nips.cc/paper/7362-pointcnn-convolution-on-x-transformed-points.pdf`.

[5] Y. Rao et al. “Spherical Fractal Convolutional Neural Networks for Point Cloud Recognition”. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).* 2019, pp. 452–460.

[6] Donald Meagher. “Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer”. In: (Oct. 1980).

[7] Gernot Riegler et al. “OctNet: Learning Deep 3D Representations at High Resolutions”. In: *CoRR* abs/1611.05009 (2016). arXiv: `1611.05009`. URL: `http://arxiv.org/abs/1611.05009`.

[8] Arie E. Kaufman. “Voxels as a Computational Representation of Geometry”. In: 1994.

[9] William E. Lorensen et al. “Marching Cubes: A High Resolution 3D Surface Construction Algorithm”. In: *SIGGRAPH Comput. Graph.* 21.4 (Aug. 1987), pp. 163–169. URL: `https://doi.org/10.1145/37402.37422`.

[10] Zhirong Wu et al. “3D ShapeNets for 2.5D Object Recognition and Next-Best-View Prediction”. In: *CoRR* abs/1406.5670 (2014). arXiv: `1406.5670`. URL: `http://arxiv.org/abs/1406.5670`.

[11] The CGAL Project. *CGAL User and Reference Manual*. 5.0.3. CGAL Editorial Board, 2020. URL: `https://doc.cgal.org/5.0.3/Manual/packages.html`.

[12] K. Weiler. “Edge-Based Data Structures for Solid Modeling in Curved-Surface Environments”. In: *IEEE Computer Graphics and Applications* 5.1 (1985), pp. 21–40.

[13] Mamadou Tahirou Bah et al. “Mesh morphing for finite element analysis of implant positioning in cementless total hip replacements.” In: *Medical engineering & physics* 31 10 (2009), pp. 1235–43.

[14] Georgi Tinchev et al. *Seeing the Wood for the Trees: Reliable Localization in Urban and Natural Environments*. Sept. 2018.

[15] Renaud Dube et al. “SegMap: 3D Segment Mapping using Data-Driven Descriptors”. In: *Robotics: Science and Systems (RSS)*. 2018.

[16] Aleksandr Segal et al. “Generalized-ICP”. In: *Proc. of Robotics: Science and Systems*. June 2009.

[17] Jun Wang et al. “Integrating BIM and LiDAR for Real-Time Construction Quality Control”. In: *Journal of Intelligent & Robotic Systems* 79 (Sept. 2014), pp. 1–16.

[18] Arlen Chase et al. “Airborne LiDAR, archaeology, and the ancient Maya landscape at Caracol, Belize”. In: *Journal of Archaeological Science* 38 (Feb. 2011), pp. 387–398.

[19] Xiaozhi Chen et al. *Multi-View 3D Object Detection Network for Autonomous Driving*. 2016. arXiv: `1611.07759 [cs.CV]`.

[20] Y. Lecun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[21] Martin Weinmann et al. “Semantic 3D scene interpretation: A framework combining optimal neighborhood size selection with relevant features”. In: *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences* II-3 (Sept. 2014), pp. 181–188.

[22] Renaud Dubé et al. “SegMatch: Segment based loop-closure for 3D point clouds”. In: *CoRR* abs/1609.07720 (2016). arXiv: `1609.07720`. URL: `http://arxiv.org/abs/1609.07720`.

[23] M. Bosse et al. “Place recognition using keypoint voting in large 3D lidar datasets”. In: *2013 IEEE International Conference on Robotics and Automation*. 2013, pp. 2677–2684.

[24] Robert Zlot et al. "Efficient Large-Scale 3D Mobile Mapping and Surface Reconstruction of an Underground Mine". In: vol. 92. July 2012.

[25] Charles Ruizhongtai Qi et al. "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation". In: *CoRR* abs/1612.00593 (2016). arXiv: `1612.00593`. URL: `http://arxiv.org/abs/1612.00593`.

[26] Diederik P. Kingma et al. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: `1412.6980 [cs.LG]`.

[27] Ingo Steinwart et al. *Support Vector Machines*. 1st. Springer Publishing Company, Incorporated, 2008.

[28] Connor Shorten et al. "A survey on Image Data Augmentation for Deep Learning". In: *Journal of Big Data* 6 (2019), pp. 1–48.

[29] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. arXiv: `1406.2661 [stat.ML]`.

[30] Angel X. Chang et al. *ShapeNet: An Information-Rich 3D Model Repository*. 2015. arXiv: `1512.03012 [cs.GR]`.

[31] Li Yi et al. "A scalable active framework for region annotation in 3D shape collections". In: *ACM Transactions on Graphics* 35 (Nov. 2016), pp. 1–12.

[32] Charles R. Qi et al. *PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space*. 2017. arXiv: `1706.02413 [cs.CV]`.

[33] Jason Ku et al. *Joint 3D Proposal Generation and Object Detection from View Aggregation*. 2017. arXiv: `1712.02294 [cs.CV]`.

[34] Yue Wang et al. "Dynamic Graph CNN for Learning on Point Clouds". In: *CoRR* abs/1801.07829 (2018). arXiv: `1801.07829`. URL: `http://arxiv.org/abs/1801.07829`.

[35] Guohao Li et al. "Can GCNs Go as Deep as CNNs?" In: *CoRR* abs/1904.03751 (2019). arXiv: `1904.03751`. URL: `http://arxiv.org/abs/1904.03751`.

[36] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: `1512.03385`. URL: `http://arxiv.org/abs/1512.03385`.

[37] Gao Huang et al. *Densely Connected Convolutional Networks*. 2016. arXiv: `1608.06993 [cs.CV]`.

[38] Fisher Yu et al. *Multi-Scale Context Aggregation by Dilated Convolutions*. 2015. arXiv: `1511.07122 [cs.CV]`.

[39] C. Chen et al. "ClusterNet: Deep Hierarchical Cluster Network With Rigorously Rotation-Invariant Representation for Point Cloud Analysis". In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, pp. 4989–4997.

[40] Haowen Deng et al. *PPF-FoldNet: Unsupervised Learning of Rotation Invariant 3D Local Descriptors*. 2018. arXiv: `1808.10322 [cs.CV]`.

[41] Yaoqing Yang et al. *FoldingNet: Point Cloud Auto-encoder via Deep Grid Deformation*. 2017. arXiv: `1712.07262 [cs.CV]`.

[42] Andy Zeng et al. *3DMatch: Learning Local Geometric Descriptors from RGB-D Reconstructions*. 2016. arXiv: `1603.08182 [cs.CV]`.

[43] Zhiyuan Zhang et al. *Rotation Invariant Convolutions for 3D Point Clouds Deep Learning*. 2019. arXiv: `1908.06297 [cs.CV]`.

[44] Daniel Worrall et al. *CubeNet: Equivariance to 3D Rotation and Translation*. 2018. arXiv: `1804.04458 [cs.CV]`.

[45] Ignacio Arganda-Carreras et al. "Crowdsourcing the creation of image segmentation algorithms for connectomics". In: *Frontiers in Neuroanatomy* 9 (2015), p. 142. URL: `https://www.frontiersin.org/article/10.3389/fnana.2015.00142`.

[46] Daniel E. Worrall et al. *Harmonic Networks: Deep Translation and Rotation Equivariance*. 2016. arXiv: `1612.04642 [cs.CV]`.

[47] Taco S. Cohen et al. *Spherical CNNs*. 2018. arXiv: `1801.10130 [cs.LG]`.

[48] Nathaniel Thomas et al. *Tensor field networks: Rotation- and translation-equivariant neural networks for 3D point clouds*. 2018. arXiv: `1802.08219 [cs.LG]`.

[49] Morris A. Jette et al. "SLURM: Simple Linux Utility for Resource Management". In: *Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*. Springer-Verlag, 2002, pp. 44–60.

[50] Martin Abadi et al. "TensorFlow: A system for large-scale machine learning". In: *CoRR* abs/1605.08695 (2016). arXiv: `1605.08695`. URL: `http://arxiv.org/abs/1605.08695`.

[51] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95.

[52] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation. Stichting Blender Foundation, Amsterdam, 2018. URL: `http://www.blender.org`.

[53] Xiaolong Wang et al. "Non-local Neural Networks". In: *CoRR* abs/1711.07971 (2017). arXiv: `1711.07971`. URL: `http://arxiv.org/abs/1711.07971`.

[54] Sergey Ioffe et al. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *CoRR* abs/1502.03167 (2015). arXiv: `1502.03167`. URL: `http://arxiv.org/abs/1502.03167`.

[55] Shibani Santurkar et al. *How Does Batch Normalization Help Optimization?* 2018. arXiv: `1805.11604 [stat.ML]`.

[56] Fakultit Informatik et al. “Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies”. In: *A Field Guide to Dynamical Recurrent Neural Networks* (Mar. 2003).

[57] Sepp Hochreiter et al. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. URL: `https://doi.org/10.1162/neco.1997.9.8.1735`.

[58] Andrew Y. Ng. “Feature Selection, L1 vs L2 Regularization, and Rotational Invariance”. In: *Proceedings of the Twenty-First International Conference on Machine Learning.* ICML ’04. Banff, Alberta, Canada: Association for Computing Machinery, 2004, p. 78. URL: `https://doi.org/10.1145/1015330.1015435`.

[59] Manolis Savva et al. *SHREC’16 Track: Large-Scale 3D Shape Retrieval from ShapeNet Core55.*