Politecnico di Milano

Scuola di Ingegneria Industriale e dell'Informazione

Dipartimento di Elettronica, Informazione e Bioingegneria

Corso di laurea in

Computer Science and Engineering

Composite Convolution for 3D Point Clouds

Supervisor:

Prof. Giacomo Boracchi

Co-Supervisor:

Dott. Luca Frittoli

Master Graduation Thesis by:

Alberto Floris
Student Id n. 905900

A.A 2020

“Ах, высоко, высоко
Небо так близко и далеко
Не увернуться, не сдать назад
Больше не будет, как прежде, брат”

Ringraziamenti

Vorrei esprimere la mia profonda gratitudine a prof. Giacomo Boracchi, per avermi guidato nella stesura di questa tesi, ma più di tutto per avermi indirizzato verso lo studio di questo specifico settore, il 3D Deep Learning, stimolante quanto innovativo.
Allo stesso modo ringrazio il dott. Luca Frittoli, per avermi consigliato più e più volte come risolvere i numerosi problemi sorti nello sviluppo del progetto, e in particolare lo ringrazio per l'impegno e la pazienza da lui spese nel discutere con me le varie bozze prodotte dalla stesura di questo documento.
Ringrazio i miei genitori, senza il cui impegno pluriennale e continuo non avrei potuto arrivare a stendere questa tesi.
Infine ringrazio Stasya: non solo mi ha sopportato in questo difficile anno fatto di distanziamento sociale e lockdown, ma ha pure avuto la pazienza di ascoltarmi più e più volte nell'esporre i contenuti di questa tesi a lei e ad altri. Non paga di questo, ha anche avuto cura di rivedere la forma dell'inglese in diverse sezioni del documento.

Sommario

Nell'ultimo decennio il Deep Learning è divenuto un elemento pervasivo nell'ambito della tecnica e dell'ingegneria: di ciò è esempio lampante la diffusione delle Reti Neurali Convoluzionali in Computer Vision e Image Analysis. Un aspetto fin ora poco studiato, ma che negli ultimi anni ha guadagnato crescente interesse nella comunità scientifica, è lo sviluppo di modelli capaci di lavorare su dati 3-dimensionali. Un mezzo sempre più diffuso per rappresentare questo genere di dati è la così detta Point Cloud, un insieme di punti distribuiti nello spazio 3D e campionati direttamente dalla superficie di un oggetto o forma 3-dimensionale. La particolarità della Point Cloud è il suo essere "non strutturata": al suo interno, infatti, i punti non possiedono un ordinamento intrinseco nè giacciono in una struttura matriciale, come invece succede per i pixel di un'immagine. Al contrario, gli elementi appartenenti a una Point Cloud possono assumere qualsiasi configurazione nello spazio da cui sono campionati. Questo lavoro indaga sulla possibilità di estendere il paradigma proprio delle reti convoluzionali a questo nuovo tipo di dato, definendo un nuovo operatore convoluzionale, la *Convoluzione Composita*. Per dimostrare le potenzialità di tale operatore e della sua implementazione, vengono inoltre condotti due esperimenti: il primo, riguardante Multiclass Classification, permette di confrontare questo lavoro con altri presenti nello stato dell'arte; Il secondo, riguardante One-Class Classification, fa riferimento a un problema ancora non approfondito in ambito Point Clouds. Tramite questi due esperimenti, si vuole dimostrare che la nostra soluzione è in grado di ottenere performance comparabili allo stato dell'arte. Inoltre, si vuole mostrare come anche problemi quali OC classification possano essere affrontati tramite l'uso della *Convoluzione Composita*, in maniera simile a ciò che si fa nell'ambito delle immagini.

Abstract

In the last decade, Deep Learning has become a pervasive element in engineering. The spread of Convolutional Neural Networks in Computer Vision and Image Analysis is a major example. An aspect that has not been studied so far but which has gained increasing interest in the scientific community in recent years is the development of models designed for handling 3-dimensional data. An increasingly popular means of representing this kind of data is the so-called Point Cloud, a set of points distributed in 3D space and sampled directly from the surface of a 3-dimensional object or shape. The Point Cloud's peculiarity is its being " unstructured ": inside it, the points do not have any intrinsic ordering, nor do they lie in a matrix-like structure, as happens instead for the pixels of an image. On the contrary, the elements belonging to a Point Cloud can assume any configuration in the space from which they are sampled. This work investigates the possibility of extending the paradigm of convolutional networks to this new type of data, defining a new convolutional operator, the *Composite Convolution*. In order to show the potential of this operator and its implementation, two experiments are also conducted. The first, concerning Multiclass Classification, allows to compare this work with others present in State-of-the-Art; The second, concerning One-Class Classification, refers to a problem not yet studied in the Point Clouds area. In such experiments, we show that our solution can achieve comparable performances with the State-of-the-Art. Moreover, we prove that OC classification can be addressed through the use of *Composite Convolution*, in a similar way to what is done in the field of images.

Contents

Chapter 1

Introduction

In the last decade, computer vision has become a key discipline in engineering, mainly due to the growth of its applications in everyday life. The birth of Deep Learning, particularly Convolutional Neural Networks [26][28], ignited interest in the discipline. Nowadays, CNNs can solve several different tasks concerning image or video analysis: classification, segmentation, object detection. Starting from around 2015 [55] [7], several research groups focused their attention on solving these same problems also for 3D shapes and objects, initiating the now-thriving field of Geometric Deep Learning. In this context, the variety of different solutions is impressive. From sparse convolution to graph-based neural networks, from Multiview CNNs to custom networks designed to handle 3D data, a significant development came in the form of Point Cloud Deep Learning.

With Point Cloud, we indicate an *unordered* set of points lying in a certain space, e.g. the 3D space. Usually, these points are sampled from surfaces they mean to represent. For example, sampled from a building's surface to represent it's geometrical shape. In many contexts, points are associated with different input features regarding Color information, Normal vectors, temperature and similar data obtained by the surface from which the point is sampled.

Point clouds are a popular way to represent 3-Dimensional data: they are commonly used in such fields as civil engineering, architecture, and heritage preservation [50] to capture the shape of specific structures. On the other hand, they are also flexible enough to be employed in various contexts with satisfying results. Research works, such as PointNet [37] [38], proved it possible to solve Deep Learning tasks over point

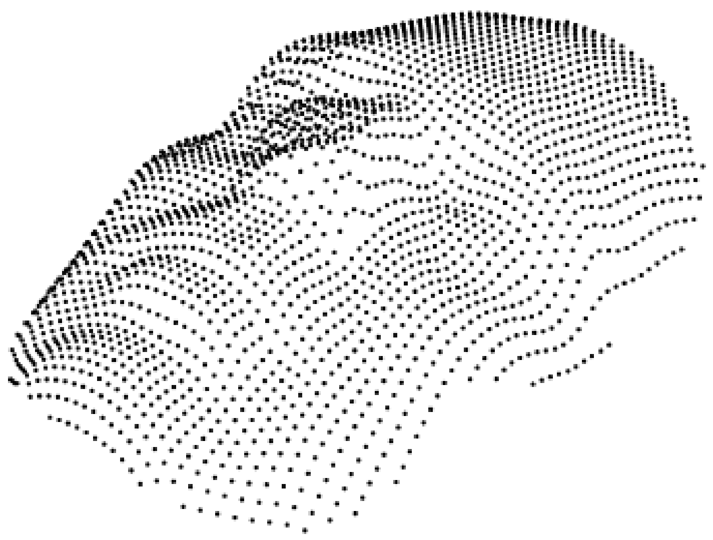

Figure 1.1: Image from [13] representing a human face with the use of a Point Cloud.

cloud datasets, clearing the way for the development of several competing methods that operate in the same scenario.

An important family of Point Cloud-based methods [51] [4] [2] [56] [14] [53] [21] [29] makes use of Convolution over Point Clouds, in a way similar to the 2D image convolution widely used in traditional CNNs. While point convolution may sound like a natural extension of image convolution, this is not the case. The points in Point Clouds are not placed in matrix-like structures, unlike the pixels of a traditional image. Indeed, the points are free to assume any position in the space from which they are sampled, and there are various modes of point sampling. For this reason, it is possible to say that Point Clouds are “unstructured”. This issue raises the question about defining such a point convolution operator in a powerful yet efficient way. A possible way to cope with the issue [56] [53] is to learn a *continuous* function, acting as a convolutional filter. Such filer is then employed in a discrete convolution, where features and points are aggregated by linear combination. Different ways to model the continuous filter have been proposed.

This thesis is embedded in this particular strand, as we are interested in investigating

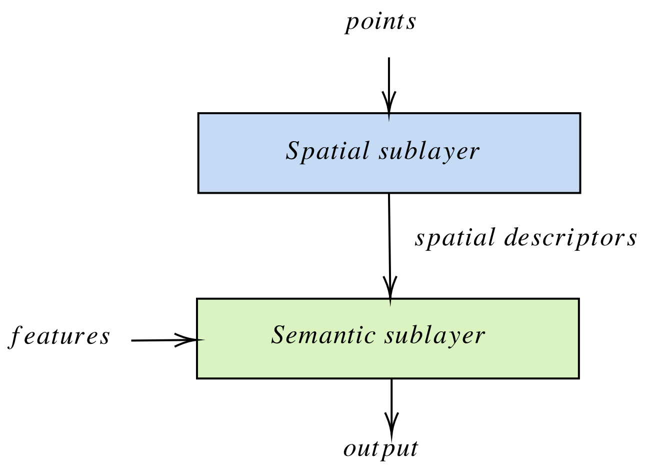

Figure 1.2: The high-level structure of our proposed layer. The spatial layers receives the input points participating in convolution and returns a *spatial descriptor* vector for each one of them. The semantic sublayer aggregates such spatial descriptors with each point's features and then computes the convolution's output.

3D Deep Learning with the use of Point Cloud CNNs. We do so by defining a novel point-convolutional operator and implementing a novel convolutional layer based on it. Our solution's peculiarity is that we define the convolution over points as a composition of two different operations. One is intended to give a spatial structure to the points involved in convolutions, and another one is intended to perform the actual aggregation of their semantic features. We call this operation *Composite Convolution*.

The implementation of these two operations is done by defining two corresponding modules:

- a *spatial sublayer*, intended to extract the neighborhood's spatial structure. In practice, this is done computing a so-called *spatial descriptor* vector describing each point's relative position.

- a *semantic sublayer*, intended to compute the actual convolution. This means to combine the spatial descriptor's components, obtaining a filter value for each neighboring point. Such filter is then used to compute convolution in a way similar to the Image case.

Our solution is structured so that these components are independent from one another. Ideally, it is possible to design independently both spatial and semantic layers and combine them in different ways. We show this by defining different possible spatial and semantic sublayers and by trying different combinations of them. In particular, we investigate the use of *Radial Basis Function Networks* (RBFNs) in defining the spatial layer. Concerning the semantic layer, we also propose a possible non-convolutional alternative, yielding interesting results in our experiments.

After defining our novel Composite Layer, we investigate its performance. In particular, we focus on two directions:

- *Multiclass Classification*, an already explored task in the field of Point Cloud Deep Learning. We focus on this task over two well-known datasets, *ModelNet40* [55] and *ShapeNetCore* [7]. In this setting, we investigate the performance of different combinations of spatial and semantic layers, together with different hyperparameter configurations. Finally, we compare our solution to the existing methods in literature. In these experiments, we show that our solution is capable of reaching comparable results with the State-of-the-Art. Moreover, we also show that the use of a non-convolutional semantic layer it is not only possible but also yield better results than some convolutional alternatives.

- Unsupervised *One-Class (OC) Classification* [45], in particular trying to recognize the classes composing the well-known *ShapeNetCore* [7] dataset. To perform this task, we employ a Deep Learning generalization of the well-known *Support Vector Data Description technique* (SVDD), presented by L. Ruff et al. in [42]. It is worth noticing that, to the best of our knowledge, this problem has not been tacked in Point Cloud Deep Learning yet. In this sense, the second relevant contribution of our work is to demonstrate that Point Cloud OC classification can by faced, in particular, by the use of our Composite Layer and Deep SVDD. The results show that the use of a non-convolutional semantic Layer allows the network to learn where convolutional solutions can't, though it is difficult to quantify the performance goodness due to the lack of competing methods. This is due to how Point Convolution and Deep SVDD are defined: for this reason, we also propose some ways to allow a convolutional layer to be trained in Deep SVDD.

Concerning the possible future extensions of this work, the second task is particularly interesting. Since there are no competing methods, a possible future development can be the definition of an alternative solution to the one presented here. Moreover, to the best of our knowledge, there are no datasets designed specifically for OC classification or Anomaly detection with Point Clouds. A beneficial development would be to define a dataset constructed purposefully for these tasks, allowing a better comparison of different models. Finally, a possible development is to apply our solution to the practical use-cases: while the work presented here is mainly theoretical, it is worth investigating its possible applications in fields such as Medical Imaging or Autonomous Navigation.

1.1 Document structure

Apart from this brief introduction, this thesis is structured in the following way:

- **Chapter 2** is dedicated to a broad introduction of the themes we will discuss in the rest of the document. In particular: we informally describe the field of Machine Learning and Deep Learning; we briefly recall the definition of the Convolution used in Image-CNNs such as [28] [26]; we introduce point clouds as a data structure that represents geometric entities.

- In **Chapter 3**, we present the state of the art in the field of 3D deep learning in general. After presenting some early approaches to 3D data, we focus our attention on the field of Point Clouds by introducing a simple taxonomy of possible methods working on them. We also state the properties an algorithm should possess when dealing with such data. Finally, we introduce a tool useful in the definition of our solution: the Radial Basis Function Network.

- In **Chapter 4** we delve into the topic of Point Convolution. Differently from 3, where we broadly present a variety of different paradigms, here we thoroughly present the point-convolutional Methods [2] [51] [4] that inspired our solution. We also discuss all the steps involved in the computation of a Point Convolution: with the aim of presenting "by example" problems and solutions encountered in defining a point-convolutional Layer. Since these works are closely related to one another, we present them in chronological order: in this way, the reader can understand the evolution of these techniques.

- In **Chapter 5** we present our solution to the problem of Point Convolution. In particular, we introduce a novel Convolutional Operator, called *Composite Convolution*, and propose a suitable *Composite Layer* implementing it. As we shall see, our *Composite Layer* is also capable of implementing non-convolutional operators: for this reason, we present a possible alternative to traditional convolution that proved to be quite useful in certain frameworks.

- **Chapter 6** is dedicated to testing our methods in two different tasks: supervised Multiclass Classification and unsupervised One-Class (OC) Classification. When testing on the first task, we are able to compare our models with the existing state of the art solutions; in OC classification we prove that it is possible, by using our approach, to perform this task over Point Clouds.
- Finally, in **Chapter 7** we draw some conclusions regarding the work done and we propose some possible future developments.

.

Chapter 2

Background

Before delving deeper into describing the state of the art in 3D Deep Learning we introduce the context in which this thesis is immersed. This introduction is meant to be quite general and informal: we chose not to spend time on the description of well-known machine learning tools, but instead, to focus on the history of the problem we are going to tackle. We discuss machine learning as a tool, useful in solving difficultly formalizable tasks; we introduce the Point Clouds as a helpful way to represent 3-Dimensional objects and shapes; we also briefly recall the definition of discrete Convolution we shall make extensive use of in the rest of the thesis. The aim is to give the reader a clear sight of the context in which this work is inserted, by leaving more formal and technical aspects to be treated in the following chapters.

2.1 Introduction to Machine Learning

Nowadays, artificial intelligence (AI) is a field with many practical applications and active research topics. When this scientific field was born, the original idea behind artificial intelligence was to develop software able to solve the problems difficult for humans but straightforward to formalise in a mathematical way. An example for this kind of problems may be the following: consider having a geographic map. The task is to assign a colour to each country in such a way that no country borders with another nation with the same colour, minimising the number of colours involved. This kind of problems can be challenging for humans, especially when having to deal with many different possible solutions that have to be explored (I.e., when the map contains many states with many common borders) nonetheless is relatively easy to develop an algorithm capable of solving the problem by relying on its mathematical formulation.

As the knowledge about how to solve the over cited problems grew, the scientific community focused its attention on the opposite problem: to develop software being able to automate tasks that are repetitive and intuitively simple for humans, but generally complex to formalise in traditional ways. An example of this kind of tasks may be to recognise objects from images, to compute the best route for a robot that needs to move, or to predict the variation of the house prices in a certain neighbourhood; when dealing with practical problems, each of these very general tasks can be grounded in many substantially different settings: recognition of bears in outdoor photos is very different from recognition of different people from their faces, but we can consider both problems to be image classification tasks. We expect the intelligent software to automate routine labor, understand speech or images, make diagnoses in medicine and support basic scientific research. In the early days of artificial intelligence, the field rapidly tackled and solved problems that are intellectually difficult for human beings but relatively straightforward for computers: problems that can be described by a list of formal, mathematical rules. The true challenge to artificial intelligence proved to be solving tasks that are easy for people to perform but hard for people to describe formally: problems that we solve intuitively, that feel automatic, like recognizing spoken words or faces in images.

2.1.1 Machine Learning as Data Driven AI

We define a Machine Learning (ML) algorithm as an algorithm that is able to *learn* from experience, in order to solve a given task. An interesting definition of a Machine Learning algorithm is the following, given by Tom Mitchel in [33]:

> *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.*

This definition contains multiple concepts that are worth explanation:

- As **Experience**, we generally mean a data collection sampled from the environment in which the ML algorithm should work. This collection is called *dataset*, ad it is used to *train* our algorithm. For example, if a certain ML algorithm is needed to recognize human faces from photos, the dataset used for training will contain a number of said faces. Vice versa, If the algorithm is needed to estimate the price of a stock option, a suitable dataset may be the past history, in terms of pricing, of said stock option.

- as **Task**, we can intend a number of different problems that can be solved with the use of Machine Learning. A number of examples were mentioned above, others include data denoising, data description, anomaly detection, text or image generation, image segmentation or text-to-speech synthesis.
- as **Performance**, we mean a certain metric which is used by the algorithm to evaluate its ability to solve the given task. Usually, this metric comes in the form of an *Objective Function* which describe the given problem and should be minimized or maximized by the algorithm; In particular, if said Objective function should be minimized, it may also be called *"loss function"* or *"error function"*.

2.1.2 Supervised and unsupervised Learning

As stated before, there are many categories of tasks that can be solved by the use of Machine Learning. In particular, this thesis will aim to solve two different tasks that fall in two different categories. The first task here discussed is called *Multi-class Classification*, and belongs to the *supervised learning* tasks' family: we can informally describe it as the problem of classifying different samples as instances of a given class of data. The second task here discussed is called *One-Class Classification (OC)*, a term used for which could also be *Anomaly Detection*. As stated by [42], OC is the task of discerning unusual samples in data, and typically belongs to the *Unsupervised Learning* tasks' family. It is now important to give an intuitive definition of what supervised and unsupervised learning is.

- In **Supervised Learning**, we associate a label to each sample fed to the machine learning algorithm. This label is known as *target*, and it represents the result that the algorithm is expected to produce for that given sample: for example, if we are dealing with image classification, we would associate to each sample (i.e., each image) a label telling us that the given sample belongs to the given class; if we want to predict house prices, we associate to each house a label containing it's true price, which the algorithm should be able to predict. Supervised learning is useful because it allows data scientists to explicitly describe the result that the

ML algorithm is expected to return. This also allows to define objective functions that directly depend on the algorithm performances in the given task.

- in **Unsupervised Learning** instead, data from the training set is fed directly to the ML algorithm, without having any labels associated to them. As described in [17], it is possible to say that the aim of unsupervised learning is to learn useful properties that depend on the training set structure. For example, a typical unsupervised learning task is to learn a compact representation for the data fed through the ML algorithm; other examples may be to learn the probability distribution from which the training data is sampled, data denoising, data clustering.

The distinction between supervised and unsupervised algorithms is not formally and rigidly defined: there is no formal way to distinguish whether a certain value is to be considered a sample's feature or the target associated to it, but is nonetheless important to intuitively understand how these two broad families of algorithms work.

2.1.3 Machine Learning and Deep Learning

Until now, we defined Machine Learning inside the broader field of Artificial Intelligence. We are then interested in describing a sub-field of ML, the area this thesis is part of: Deep Learning.

In Machine Learning we employ data to solve different tasks. However, the performance of many simple ML algorithms depends heavily on the representation of the data they are given. Such data representation can be decomposed in different atomic units, usually known as *features*. As stated above, such features can dramatically influence the performances of an ML algorithm, which raises the question of how to define them before they are employed.

In traditional Machine Learning, experts give structure to data using handcrafted features. They were tasked with finding ways to extract information before presenting the data to the algorithms. A field later developed, called *Representation Learning*, was precisely focused on producing such intermediate data representation directly with machine learning tools. This is also a form of machine learning, aimed at replacing the feature extraction effort done by experts.

In this sense, it is possible to use Representation Learning for feature extraction and machine learning for task solving. Also Representation Learning, however, is a Machine Learning task: as with other Machine Learning tasks, it requires data that is more or less structured. By iterating the reasoning described before, it is reasonable to stack different Representation Learning together: in other words, to introduce representations that are expressed in terms of other, simpler representations. By Following this idea, it is possible to build "*deep*" models composed of many different sub-units, shaping out a representation hierarchy that starts from raw data and ends with the solution of a given task. This is precisely what Deep Learning is, and its most representative algorithm is the Neural Network (NN).

Of the many sub-fields of Deep Learning, we are most interested in Deep Computer Vision. Here, the most widely known model is the *Convolutional Neural Network* (CNN), a paradigm originally designed for image-based deep learning. In this thesis we are interested in extending this specific paradigm, thought to be applied on images, to the cutting-edge field of 3D Deep Learning.

2.2 A brief recall of Image Convolution

In 1998, Ian LeCun et al. [28] introduced the first Convolutional Neural Network (CNN) to recognize handwritten digits for the banking and postal industries. This work is seminal for starting a compelling ML branch, the so-called Deep Learning. While this technology was difficult to exploit in its early days, it gained significant momentum in 2012, following the work of A. Krizhevsky et al. [26]: nowadays CNNs are a crucial tool in the field of Deep Learning, specifically when dealing with ordered data like in many Computer Vision applications. With "ordered data" we mean data whose composing elements (e.g. pixels for images) are ordered and inserted into a specific structure (in the case of pixels, the matrix representing the image). The novelty of CNNs resided in the fact that they could extract information directly from such ordered data in a very effective way, without an expert needing to define handcrafted features. In contrast to the previously existing methods the efficacy of CNNs is much higher.

In particular, such feature extraction is done by means of the discrete convolution operator:

$$(\phi * g)(y) = \sum_{x_i \in X} \phi(x_i) g(x_i - y), \tag{2.1}$$

where x_i is a pixel, $\phi(x_i)$ the features associated to it. Here $g(\cdot)$ is a function called filter, that depends on the distance between the output pixel y and each pixel $x_i \in X$ participating in convolution.

Let us consider the case of images: in this setting, This operator is used to aggregate neighboring pixels together and extract information from their spatial distribution. This aggregating process is precisely the one extracting our image's features. The elements composing it are the following:

- the function $\phi(x)$ identifies the features associated with a given pixel x: for example, the color.
- the set X, called neighbourhood, is composed pixels surrounding the output pixel y. In practice, how many pixels are part of the neighbourhood is decided based on how the filter's function $g(\cdot)$ is defined.
- $g(x - y)$ is a discrete, compact support function that associates a weight to each pixel inside a given neighborhood of the output. In this sense, the argument of the function $g(\cdot)$ is the relative position between the output pixel y and a neighboring pixel x_i. In practice, this function is defined as an enumeration: to each possible argument, a specific output value is defined.

The output $(\phi * g)(y)$ can be interpreted as a pixel of a new image, having a specific position y and a specific feature (or set of features) $(\phi * g)(y)$. Having an image as output, it is possible to then apply another convolution to it: in other words, we can stack together different *convolutional layers*, in order to build a powerful feature extractor able to catch local as well as global details.

In this thesis, we shall not delve too much in discussing about CNN architectures and their general properties: we are more interested in discussing the peculiarities of the Image-convolution operator. In the case of images, the definition of convolution heavily relies on the fact that pixels' positions are discrete and distributed in a grid-like structure: the image itself. In the last years, many different works had the goal of extending such operator to context in which such grid-like structure is not present: the

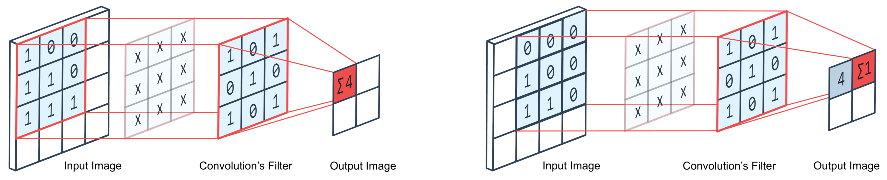

Figure 2.1: Convolution over images. In particular, we see how two pixel values of a (2×2) output image are computer.

case of Convolution over Point Clouds is one of these. In this sense, the problem we are trying to solve is easy to explain: redefine discrete convolution for unstructured and unordered data.

2.3 Introduction to Point clouds

The last thirty years saw the rise of automatic computation in many different disciplines. An important innovation was the rise of CAD modeling in many engineering fields, such as mechanical engineering or civil engineering. We can relate the origin of such that we now call "Point Cloud" to these early days.

As stated in [22], a typical task in CAD modeling is to construct a model starting from a physical object or part: for example, to produce a CAD model starting from a physical (e.g., clay, wooden) mock-up. To accomplish this, it is necessary to:

- Produce an abstract, easily readable, sufficiently precise and reliable representation of the physical object.
- Decode this representation of the given object and transform it to a 3D mesh or similar kind of data, readable and modifiable using CAD software.

Point clouds (PCs) proved to be an incredibly successful representation for this kind of problem: they not only can be easily produced starting from a physical object, but it is also easy to convert them to 3D meshes. They are generally produced by the means of a 3D scanner, or by the use of photogrammetry techniques. For example, LiDAR scanners are often employed to produce point clouds representing architectural objects or landscapes.

We can describe a Point cloud as an *unordered set* of points, lying in a n-dimensional space. Usually, though not always, these points are taken from a given surface (as by using the over cited 3D scanning and photogrammetry techniques) and for this reason they are easily converted to polygon meshes, NURBS (Non-uniform rational B-spline) surfaces or similar data. Nowadays, Point Clouds are not only used by mechanical engineers, but they find application also in biomedical imaging, in Geographic Information Systems (GIS) and many other fields.

While initially meant to represent real-world objects, PCs are nowadays also used for the opposite goal: to represent ideal objects, like meshes or CAD models, in a compact and computation-friendly way. In particular, Point Clouds can be helpful in

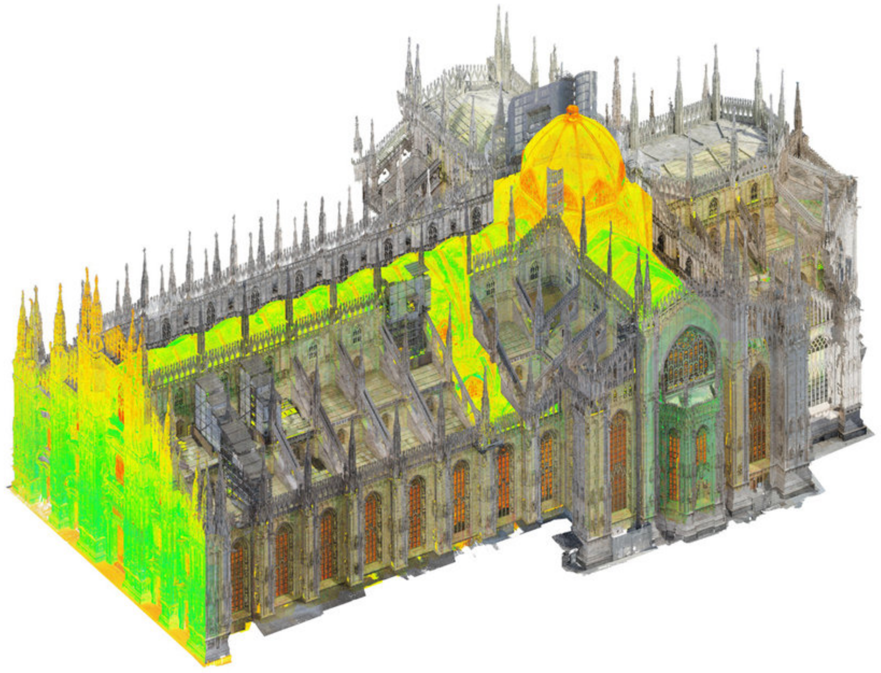

Figure 2.2: Image from [50] representing an 3D Point Cloud (PC) of the entire Milan Cathedral (more than 3 billion points). In this case, the PC is obtained by merging two different types of points: TLS data, that are shown with their intensity colours (green, yellow, orange) and photogrammetric data with RGB color information.

tasks where a standard 3D mesh would be difficult to handle. This later use is more recent than the one described before and gained momentum specifically in the last decade. In particular, PCs raised particular interest in the field of 3D Deep Learning: they are an uncommonly flexible data type, capable of representing 3D shapes both precisely and in a compact fashion. In this field, the pioneering work was *PointNet* by C. Qi et al., which proved that Deep Learning could be applied to PCs to solve several supervised learning tasks.

For this reason, we can trace a first broad categorization for Point Clouds:

- Synthetic PCs, designed starting from synthetic data, are used to perform tasks over 3D shapes. In this case, using PCs as data structure is owed to the fact that Point Clouds are easier to handle than other representations.
- Real-world PCs are sampled directly from real-world objects and are oriented to capture the object's features, often in a very detailed manner. In this case,

the use of PCs is due to the fact that most of the scanning techniques produce Point Cloud representations. Depending on the framework, real-world point clouds can be prone to issues. Some of these problems are related to the point's distribution over the represented shape or to the inability to sample particular details accurately; some others are related to the features associated with each point, like its color or the normal vectors to the sampled surface;

Other than this broad classification, it is possible to identify several possible differences between Point Clouds used in different contexts. For example, the PCs used in architecture and heritage preservation (like the one shown in figure 2.2) are certainly richer in terms of features and amount of points than the Point Clouds used by autonomous agents for navigation. In this sense, PCs are undoubtedly flexible instruments in representing the 3-Dimensional world in quite diverse settings. This fact is the main reason behind the thriving of Point Cloud-based methods in different computer vision fields, Deep Learning being one of these.

Chapter 3

State of the art

As discussed in chapter 1, this thesis contributes in two different directions: the first consists in the definition of a novel point-convolutional layer designed to operate on point clouds. The second direction concerns testing our novel convolutional layer in two different tasks: supervised Multiclass Classification and unsupervised One-Class (OC) Classification. This chapter treats all the fundamental topics needed to understand both the approach proposed by this thesis and other related works on deep learning with point clouds.

First, we introduce some useful topics in understanding the goal of this thesis and the design of our proposed solution. In particular, section 3.1 is dedicated to discussing a simple alternative to the traditional Multilayer perceptron (MLP) of which we shall make extensive use: the Radial Basis Function Network (RBFN). Furthermore, in section 3.2 we shall spend a few more words about what point clouds are and what properties are expected from machine learning algorithms dealing with them;

In the second part of the chapter, we describe the other related methods involving deep learning on 3D data and point clouds. In particular, section 3.3 introduces some early approaches to Geometric Deep Learning; section 3.4 will be dedicated to present a simple taxonomy for methods involving supervised Deep Learning tasks with point clouds. Finally, in section 3.5 we discuss unsupervised techniques on point clouds.

3.1 Radial Basis Function Networks

In the field of deep learning, it has been proved that Multi-Layer Perceptrons (MLPs) possess universal approximation capabilities [17]. For this reason, MLPs are widely used in system identification, prediction, regression, classification, control, feature extraction, and associative memory. Here, we present a different family of neural networks that, like MLPs, can approximate an arbitrary function. This meta-algorithm is called Radial Basis Function Network (RBF): in defining a convolution over point clouds, we shall make extensive use of it.

3.1.1 Introduction

Firstly introduced by Broomhead and Lowe in [5], Radial Basis Function Networks possess the same approximation capabilities of an MLP, with the significant advantage of faster convergence[60]. Historically, RBFs were introduced for exact function interpolation [3]: having a set of points $x_1, x_2, x_3, ...$ lying in a n-dimensional space, each one associated to a target vector $t_1, t_2, t_3, ...$, the purpose was to find a function $f(x_i)$ able to fit the target exactly. This result was reached using the same principle of many basic regression models: via a linear combination of fixed Basis Functions, each having as argument a certain number of input features. In the case of Radial Basis Function Networks, as suggested by the name, this linear combination is performed between basis functions of the type:

$$h(||x_i - c_l||).$$

In other words, each basis function h_l depend only by the distance between an input point and a certain vector *center* c_l, associated with it; moreover, this means that all bases are obtained by translating the same function and thus their values depend only on the center c_l. The RBFN output is then computed as:

$$f(x_i) = \sum w_l h(x_i - c_l).$$

, where the coefficients w_l can be learnt by Gradient Descent or Ordinary Least Squares as with other simple regression methods. The only difference between this exact interpolation method and the use of RBFNs in machine learning is given by the fact that, whereas in exact interpolation the goal is to fit exactly each input point, this is undesirable when doing machine learning in order to avoid overfitting and allow the model to generalize better.

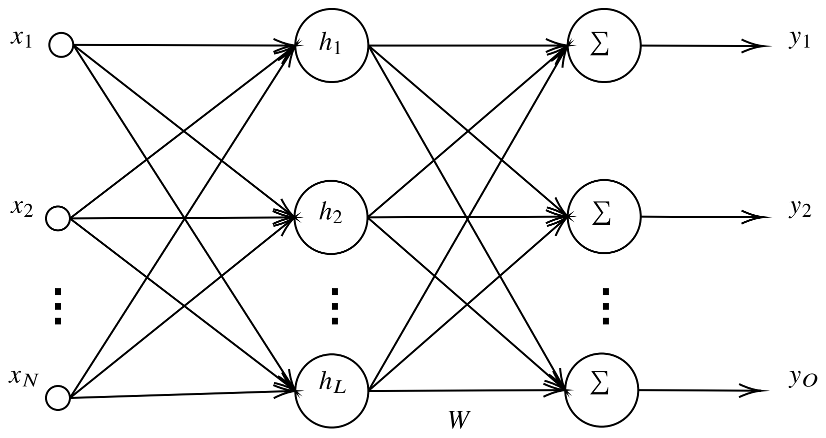

Figure 3.1: Architecture of an unbiased Radial basis function network with N inputs, L hidden neurons and O output neurons respectively

3.1.2 Neural network Architecture

Since RBFNs approximate functions by a linear combination of RBFs, they are often represented as simple three-layer feed-forward Neural Networks [60]. Let us consider an RBFN with N inputs, L RBFs, and O outputs: in this case, each Radial basis function can be represented by a hidden neuron, using the corresponding RBF as activation function. It is possible to see a representation of this simple neural network in figure 3.1.

We remark that in this network, non-linearities are added directly by the radial basis function, and that each radial basis function differs from the others only because of its associated center c_l. For this reason, the network's output is computed as:

$$y_o = \sum_l w_{lo} h(||x - c_l||).$$

, where y_o is the o-th RBF output, w_{lo} are the network's learnable parameters, h is the RBF here used as activation function, and c_l is the l-th RBF center. Alternatively, we can write the output vector in matrix form, as:

$$Y = W^T H.$$

Given its simple architecture, the performance of a radial basis function network depends on three factors:

- **choice of the RBF shape**: there are multiple suitable radial basis functions that can be used in a RBFN. The one perhaps most commonly used is the Gaussian function:
$$h(||x - c||) = e^{\frac{||x-c||^2}{2\sigma^2}}.$$
- **choice of the centers**: since the difference between each hidden neuron is due only to its center, a suitable center selection strategy must be enforced: in certain cases, it is possible to learn the center's positions, while in other it is better to select them a-priori.
- **choice of the hidden layer's dimension**: It is possible to define more centers, meaning more RBFs, to increase the approximation capabilities of the RBFN. Vice versa, it is possible to use fewer RBFs to reduce overfitting.

3.1.3 Choice of Radial Basis Function

In theory, a Radial Basis Function is simply a function whose value depends only on the norm of the difference between input and output. For example, the function can: $y_c(x) = ||x - c||$ is an RBF. In practical applications, though, we are more interested in certain RBF classes that have proven to yield better results in approximating arbitrary functions. An important class is the of *localized RBFs*: we call a certain RBF $h(||x - c||) = h(r)$ localized if these two properties hold true:

$$\lim_{r \to \infty} h(r) = 0.$$

$$h(r_1) > h(r_2) \quad \forall\, r_1, r_2 \;:\; r_2 > r_1.$$

If a Radial Basis Function network is made of localized RBFs, then the network will produce similar outputs for sufficiently close input vectors; on the other way, having sufficiently distant input vectors yields nearly independent results [60]. This property is a form of local generalization, similar to the one of image-convolutional layers.

Besides the localization property, a radial basis function $h(||x - c||)$ has to satisfy the following properties [23] to guarantee the network's universal approximation capabilities:

$$\begin{aligned} &a) \quad h(||x - c||) > 0 \quad \forall\, x. \\ &b) \quad \nabla_x h(||x - c||) < 0 \quad \forall\, x. \\ &c) \quad \nabla_x^2 h(||x - c||) > 0 \quad \forall\, x. \end{aligned} \tag{3.1}$$

If a radial basis function is both localised and satisfies (3.1), then the actual shape of the RBF is irrelevant in terms of generalization power [23]: the network will be able to achieve universal approximation capabilities, and its precision will depend only on the number of hidden neurons. This does not mean, however, that all of such functions yield the same performance given a network with a fixed number of hidden neurons: on the contrary, it is possible that certain RBFs perform better, and certain others worse. This is analogous to what happens when dealing with Discrete Convolutional Neural Networks: the intrinsic capabilities of a CNN are not bound by the shape and dimension of its filters, but the performance obtained by a certain fixed CNN can be heavily impacted by its filter's shape and size. For this reason, many different radial basis functions were proposed to be used in RBF Networks. Some examples are:

$$\begin{aligned}
&a) \quad h(r) = \exp\left(\frac{r^2}{2\sigma^2}\right).\\
&b) \quad h(r) = \frac{1}{(\sigma^2 + r^2)^{\alpha}}, \quad \alpha > 0.\\
&c) \quad h(r) = \sqrt[\beta]{\sigma^2 + r^2}, \quad \beta > 0.\\
&d) \quad h(r) = r^2 ln(r).
\end{aligned} \tag{3.2}$$

Note that between the RBFs in 3.2, only a) and b) are localized functions. On the other hand, both RBFs c) and d) proved to be valid alternatives in specific contexts [60]. The most widely used radial basis function remains the Gaussian function a), because of its simplicity, the existence of only one tunable parameter σ and the fact that it's both localized and satisfying properties 3.1;

3.1.4 Choice of RBF centers

In the literature, numerous alternatives have been proposed to select RBF centers' positions. For example, J. Gomm and D.Yu in [16] propose to use Recursive Orthogonal Least Squares (ROLS). Another well known method is to perform k-means clustering over some input points and place the centers near each cluster's centroid [54]. In general, what method to employ heavily depends on the kind of data and on the task needing to be solved. In the case of point convolution, a possible solution is proposed in [51]: to select the centers' positions by solving an optimization problem. In this case, the authors wanted the RBF centers to be as far from each other as possible inside a given sphere. Each center was assigned a repulsive potential towards the other points, concurrently with an attractive potential towards the sphere's center. In this setting, the optimization problem consisted of minimizing the global energy. Another solution proposed in [51] is to simply learn the RBF centers by Stochastic Gradient Descent.

3.2 Properties of ML algorithms on point clouds

Point Clouds are simple, unordered sets containing points lying in an n-Dimensional space. Their importance derives from the fact that they can represent virtually any kind of n-D surface compactly. It is possible, although not necessary, to associate a certain number of qualitative features to each point in the set. For example, a possible feature is the vector normal to the surface from which the point was sampled. In machine learning, we are interested in building algorithms that are able to extract information from data: in other words, we expect our algorithm to generalize. To achieve this generalization capability on such unstructured and elementary data can be very challenging. There are specific rules our algorithm should comply with when defining how to interpret each datum, so some algorithms are more fit to the problem. The case of point clouds is precisely this one: Being Point Clouds elementary data structures, we need to enforce specific properties when designing a machine learning algorithm that learns from them. We report those presented by Charles Qi in Pointnet [37], one of the pioneering works about Deep Learning on point clouds:

1. **Order invariance:** *a machine learning algorithm should not enforce any ordering between points in a point cloud, nor should it be relying on such order*. This property derives the simple observation that point clouds are unordered sets: feeding the algorithm with a given sequence of points $(x_1, x_2, ..x_n)$ should yield the same result as when feeding the algorithm with the opposite sequence $(x_n, x_{n-1}, ...x_1)$. It is also easy to be convinced of the soundness of this property by thinking that a point cloud is, in most cases, a set of points sampled from an n-D surface that is meant to represent the surface itself; in this case, the ordering is a property about how the sample was performed, and not about the sampled surface.

2. **local structure awareness:** *machine learning algorithm should capture local*

structures contained in neighborhoods of nearby points. Since point clouds are sampled from metric spaces, it is possible to enforce a distance metric (like the euclidean distance) between points. In other words, points should not be considered as isolated entities: the information our model needs to extract resides in the interactions between them. In a certain sense, this property mimics the locality property typical of discrete CNNs.

3. **Invariance under transformations:** *a Machine Learning algorithm should yield the same result when fed with a given point cloud P_1, and when fed with a point cloud P_2 obtained by applying certain geometrical transformations on* P_1. This principle, though very intuitive, is left intentionally vague: the kind of geometrical transformations our algorithm should be invariant to depend on the application. For example, if our point clouds are sampled from rigid physical objects, it is safe to build an ML model that is invariant concerning rotation, translation and mirroring. At the same time, it is not safe to assume that the model should be invariant to scaling. An example of this latter statement could be distinguishing between dolls and people where the principal factor would be the dimension of the objects of interest. Thus, scale invariance can better be avoided in some particular environments while being helpful in others.

It is possible to trace a specific parallelism between image-based and point cloud-based algorithms: we expect that both image-based CNNs and Point Cloud-based Deep Learning models are transformation-invariant and capable of learning local and global structures. On the other hand, though, images are a data type containing an intrinsic ordering between its elements (the pixels). This is the main difference between a standard image-based deep learning algorithm and algorithms that deal with Point Clouds: it is impossible to rely on an inherent structure of the data.

3.3 Early approaches to deep learning with 3D data

Point clouds were first defined as intermediate step between real 3D objects and their digital representation, whether in the form of a mesh or as CAD model. This is why deep learning on point cloud is strictly tied with deep learning on 3D objects. The aim of this section is to present two specific families of deep learning algorithms devoted to 3D-based tasks.

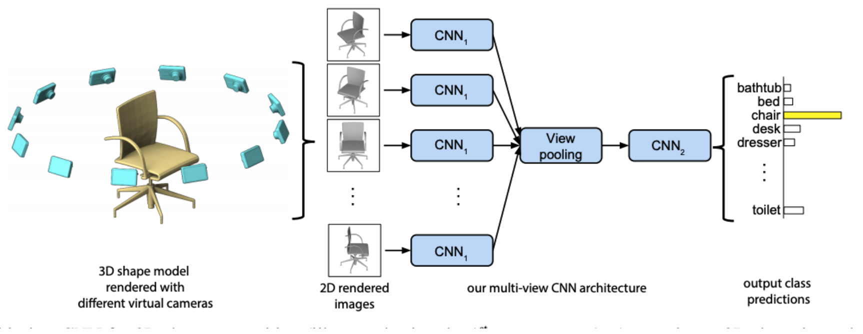

Figure 3.2: Image from [47] showing a multiview CNN architecture for 3D shape classification. Multiple images of a 3D model are taken, each one from a different (but fixed) point of view. Each of those images is sent to a different CNN. Results for all the CNNs are pooled together and another CNN is fed with them. The output of this last CNN represent the classification score of the given object for each class.

The first 3D Deep Learning models were *Multiview CNNs*, which take several 2D images from a 3D object (either a mesh, a cad model, a point cloud, or a physical 3D object) to represent it from different points of view. These images are then fed to a CNN-like deep neural network to extract information from them. These methods can achieve good performances, but they also require carefully crafted CNN architectures to work. In other words, this solution heavily depends on the network architecture, while it would be desirable to have architecture-agnostic approaches. An example of this kind of model is described in [47], and briefly presented in Figure 3.2. It is worth noticing that although the network described in [47] is meant for 3D objects, it is can also handle point clouds.

The second family of models is known as *Volumetric CNNs*, and it represents a direct 3D generalization of discrete Convolutional Neural Networks on images. This family of methods create a grid structure to discretize the 3D space. In this way, we obtain a 3D structure very similar to the matrix representation we use for images, thus we should be able to use tools that are analogous to those used in image recognition. The process of discretizing the space to fit a tensor-like structure is called Voxelization, and represent the core element of these methods. Unfortunately, there are several drawbacks in this approach:

- Voxelization implies loss of information: each element (voxel) of our matrix-like structure is meant to contain a value (or a set of values) that describes the whole

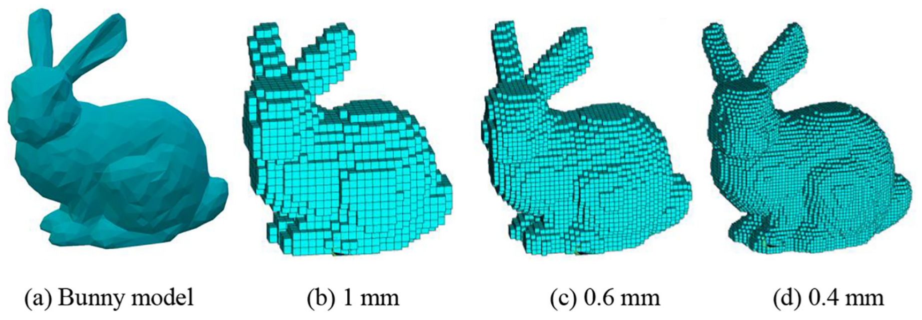

Figure 3.3: Image from [64] that shows a bunny mesh together with three voxelized representations. It is possible to see how such discrete representations heavily depend on the voxel resolution to capture certain details.

volume of the voxel itself. Using this technique, the "resolution" of our 3D grid heavily affects the fidelity of our 3D shape representation as shown in Figure 3.4. It is possible to solve this problem by performing a very fine-grained voxelization, but this causes significant problems in terms of performance.

- 4-Dimensional tensors are much more complex to handle than 3-D tensors, such as images. This is trivially understandable by comparing a $(32 \times 32 \times 3)$ RGB image, composed of 1024 pixel with three color channels each, and a $(32 \times 32 \times 32 \times 3)$ 4-D tensor, containing 32768 elements each one with three feature channels. Moreover, voxelization needs to be as fine-grained as possible to limit the loss of information due to discretizing the space, which only worsens the problem of handling N-dimensional tensors due to the additional tensor dimensions.

- A considerable share of the voxelized space typically does not contain any information. Not all voxels are built in over the surface of the represented object: those lying completely outside or inside the represented object are useless in describing its 3D shape.

In order to solve the second and the third caveat here cited, a possible solution is to make use of sparse tensors, which do not keep track of useless (i.e., empty) voxels. Some Sparse Convolutional Approaches rely precisely on this idea, an example being [18].

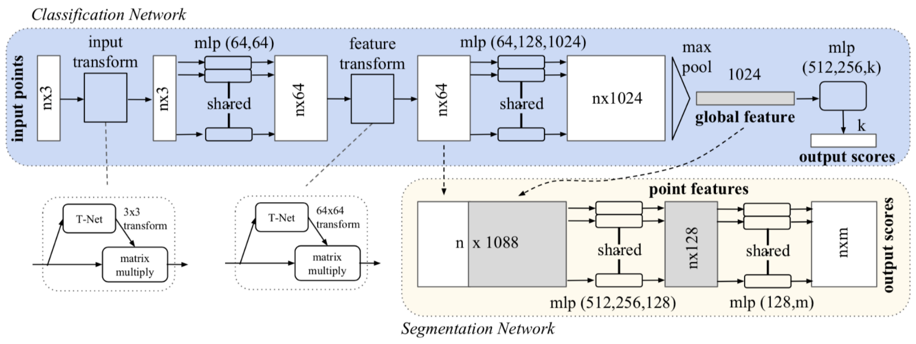

Figure 3.4: Image from [37] representing the architecture of *Pointnet*, both for classification (in blue) and segmentation (in yellow), indicating each input and layer dimension. As it is possible to notice, much of the architecture is shared both for segmentation and classification.

3.4 Supervised models for point clouds

As from 2015, a number of possible 3D deep learning models that make use point clouds as input data were presented. The aim of this section is to discuss four broad classes of 3d deep learning models related to point clouds: Point-wise MLPs, Point Convolutional networks, Graph based Networks, and other custom approaches shall be presented. It is not always easy to precisely classify a deep learning method into a certain category: for example, graph-based methods can be considered as an alternative to point cloud methods, but in [19] they are classified as point cloud based. It is also not rare to encounter methods that can be classified as part of two different classes: for example [14] can be both classified as a graph based method and as a point convolutional method.

3.4.1 Pointnet and Point-wise MLP methods

The most famous Deep Point Cloud model is certainly *Pointnet* [37], together with its successor *Pointnet++* [38] from the same authors. In these two models, presented for classification and segmentation, the given task is solved by the use of a very specific deep architecture. For example, a classification task is solved in *Pointnet* by performing the following steps:

1. the input point cloud is fed to a sub-network that produces a second point cloud thought to be invariant with respect to rotation and translation. Note that, if there are some features associated to each point of the input point cloud, those are

treated like additional spatial coordinates.

2. each point inside the transformation invariant point cloud is fed to a single, shared multilayer perceptron that extracts a number of features from it.

3. the feature point cloud extracted from the multi-layer perceptron is then again fed to a transformation invariant sub-network.

4. feature point clouds are then max pooled to achieve permutation invariance.

5. the output of the MaxPooling layer is fed to a dense layer that will output class probabilities associated to the point cloud.

Figure 3.4 shows a representation of the architecture over described. In this case, we chose not to describe in details how such operations are performed, because we are interested in pointing out the fact that the original contribution is the overall architecture of such model, more than its components. On the contrary, in the next sections we will see that many other works already presented in literature try to define new and original components thought to be used in already existing architectures.

Models such as Pointnet are known in literature as point-wise MLP methods [19], given the fact that semantic is extracted from point clouds with the help of a shared multilayer perceptron that analyzes each point separately. These methods were one of the first deep learning models designed to work directly on point clouds. Other examples of point wise MLP methods are *"Deep Sets"* [61] and "*Point Attention Transformers*" [57].

3.4.2 Convolution based methods

Convolution based methods represent another important family of 3D Deep Learning algorithms. These methods aim to define a convolution operator that can be used on point clouds, like the traditional discrete convolution operator is used on images. The main advantage of these approaches is that said convolutional layer can be used inside any traditional CNN architecture (such as the well known ResNet [**o**]r VGG16 [46]) without any substantial modification. Given both the advantages of being architecture agnostic and theoretically straightforward, this family of methods is the most successful one, and most of the already presented models can be classified as convolution based methods[19],.

We shall not discuss now how these models work, as we will present some examples in the next sections of this chapter. It though worth to cite some important models presented in literature: for example, *PointCNN* [29] transforms the input points into a latent ordered set through a MLP, and then applies typical convolutional operator on the transformed features. In *SpiderCNN* [56], the convolution is produced by approximating a continuous filter function with the help of a polynomial. Another possible approach is to approximate said filter function with the help of a MLP, like in the case of "*Deep parametric CNN for point clouds*" [53]. As it is possible to understand by the previous examples, there are many alternative and substantially different ways to define a point convolutional layer, which is perhaps one of the main reasons behind the success of this category of models.

3.4.3 Graph based methods

The third alternative for point cloud deep learning is represented by the so called graph based models. These models consider each point in a point cloud as a vertex of a graph, and generate directed edges for the graph based on the neighbors of each point. Feature learning is then performed in spatial or spectral domains, therefore it is possible to distinguish between spatial graph based models and spectral graph based models. In the case of spatial graph based models, many works available in literature implement some form of spatial convolution in a way similar to non graph based convolutional models: an example of this kind of models is *SplineCNN* [14]. On the other hand, spectral graph based methods usually make use of graph spectral theory, producing substantially different methods with respect to those presented in this thesis. A famous example of spectral graph based method is [10].

3.4.4 Other methods

Besides models that fall in the categories previously defined, there also exist some custom methods. For example, several methods like *OctNet* [40] and *Kd-net* [24] make use of hierarchical data structures to model and process point clouds. More inherently to this thesis, *RBFNet* [8] uses a radial basis function network as a sort of encoder for the input point cloud, exploiting the fact that radial basis function networks are able to catch local spatial structures. The encoded point cloud is then classified with the help of a MLP.

3.5 Anomaly Detection and Point Clouds

The field of deep learning with point clouds is relatively young compared to other research topics. One of the first works trying to define a deep neural network for classification and segmentation operating on point clouds is the already discussed PointNet in 2015; starting from PointNet, several different authors proposed different models designed to solve similar supervised learning problems. On the other hand, few works discussing unsupervised techniques with point clouds have already been presented. In this section we briefly discuss what unsupervised learning techniques with point clouds are available today, particularly focusing on the broad field of anomaly detection tasks.

3.5.1 Point Cloud Autoencoders

In the field of anomaly detection for images, many models rely on the concept of deep autoencoder. As in [17], we can define "deep Autoencoder" as a neural network that tries to learn the identity function or, in other words, to replicate its input. This is done by learning a compact input's representation capable to capture its main features. Several different kinds of deep autoencoder were proposed and implemented, depending on the task that needed to be solved and the kind of available data. In particular, autoencoders proved to be extremely useful when dealing with denoising and, most importantly in our case, when dealing with unsupervised one-class classification (OC). As pointed out in [19], there exist some proposed autoencoders for point clouds. Some relevant examples are:

- *FoldingNet* [58] presents a graph-based deep autoencoder that heavily relies on the fact that point clouds are usually sampled from surfaces. The idea is that any 3D object surface could be transformed to a 2D plane by the use of certain cutting, squeezing, and stretching operations. This operation is reversible, as it is possible to reconstruct the 3D shape starting said plane by repeatedly folding it and concatenating the vertices and the edges.
- in [62] Zamorski et al. Proposed a novel point cloud Adversarial Autoencoder, namely *3DAAE*.
- in *3d Point Capsule Networks* [63] the authors try to define a novel point cloud autoencoder by generalizing the idea of capsule networks already used in 2d images to 3d point clouds; Capsule networks [43] are an extension of the traditional

image CNN model that is designed to be the solution of problems such as the so called *Picasso Problem*: in a traditional image CNNs recognizing faces, the relative position of various elements - like mouth, eyes and nose - is not taken into account by the network, which relies only on the mere presence of such details when deciding whether an image contains a face or not.

Two measures are usually employed to evaluate how much a Point Cloud is well reproduced by an AutoEncoder. The first one is the so called *Chamfer distance*, employed by *Foldingnet* [58] and by *3DP Capsule Networks* [63]. The second one is called *Earth mover's distance* [41] and it is employed, for example, by *3DAAE*.

3.5.2 Denoising and OC classification for point clouds

Some of the of point cloud anomaly detection works already available in literature try to deal with the problem of denoising and outlier point detection. Because of the noisiness of the point cloud, especially those sampled from real-world environments, it is interesting to understand which points belonging to a given point cloud are perturbed by noise and which are not. Even better would be to adjust the position of each point in a way such that the noise is minimized. An interesting work in the field of point cloud denoising is *Total Denoising: Unsupervised Learning of 3D Point Cloud Cleaning* by Hermosilla et al. [21]. Another paper worth citing on this topic is *3D Point Cloud Enhancement using Unsupervised Anomaly Detection* by Regaya et al. [39].

Another important and well known kind of task in anomaly detection is the one of One-Class Classification: as from [6], we call One-Class classification the task in which the model is able to distinguish elements belonging to a given "*normal*" class from anomalous elements, i.e., elements that do not fit the training data distribution. In case of unsupervised OC, all the training set is composed by instances of said *normal* class. In this environment, the model is able to discriminate between normal and anomalous instances by learning a space region containing as many training elements as possible. As a result, if an instance falls outside the mentioned region, this instance is likely to be anomalous. Differently from denoising, it is not simple to find in existing literature works that describe how to solve unsupervised point cloud OC classification tasks. To the best of our knowledge, no deep learning model solving OC classification for point clouds has been proposed yet. This raised an issue during the development of this thesis, as it is not possible to compare the model we will present with other already existing works. Moreover, it is worth noticing that recent and very general surveys

[19] dealing with the topic of deep learning with point clouds do not mention one-class classification methods.

Chapter 4

Point Convolution: some examples

After discussing the State of the Art in a more general way, we propose a more thorough description of some convolutional methods for Deep Learning on Point Clouds, namely *Point Cloud CNN by Extension Operators* [2], *Kernel point Convolution* [51] and *ConvPoint* [4]. We choose to present such methods for a number of reason: first, they are closely related to what we present in our proposed solution. Second, they allow us to explain how some typical point convolutional layers are defined and implemented. Finally, presenting these already existing layers should allow us highlighting the novelty of our proposed solution better when compared to similar models. For these reasons, we also discuss the three works chronologically: the first one presented, PCNNEO, is the oldest among the three and at the time it presented an entirely new approach to the problem of deep learning with point clouds. The second one, KPConv, will try to enhance the idea behind PCNNEO by using a different definition of the convolutional operator. Finally, Convpoint presents a design similar to KPConv with an important novelty about how to define the convolutional filters. Discussing these works also allows us to introduce several elements needed to define our solution that shall also be presented in chapter 5.

4.1 Point CNNs by Extension Operators

In the previous section we discussed Pointnet [37], a non-convolutional model that handles both the spatial coordinates and the features associated to them. In Convolutional Neural Networks, like those used in image recognition, pixel features (e.g., color channels, alpha, ...) are treated separately with respect to pixel positions. To be more precise, pixel positions are only used in determining the filter's $g(\tau)$ argument $\tau = x - y$, while pixel features are multiplied together with said filter. The core idea of the continuous convolution for Point Clouds is precisely to handle spatial coordinates and features separately. In this section we shall describe a model, called Point CNN by Extension Operators (PCNNEO) [2], that tries to generalize the conventional Convolutional operator to Point Clouds. Differently from PointNet, this approach is designed to be architecture-agnostic: the convolutional layer here described can be used in any architecture used in image-based convolutional neural networks. This will allow us to focus on describing the convolutional layer itself, while considering the choice of the architecture as an implementation detail. Another important detail in this model is the extensive use of Radial Basis Functions (RBFs): exploiting their ability to capture local structures, PCNNEO's authors use them to implement the core part of their convolution operator. As we shall see, this approach will be referenced by other authors in subsequent works.

4.1.1 Convolutional layer's input and output

PCNNEO's fundamental component is the convolutional layer, being it agnostic with respect to the Neural Network architecture used. We now present what we expect to be the input and the output of each convolutional layer within PCNNEO. Recall that since we are defining a convolutional layer, our input may either be the original Point Cloud fed to the model, or an intermediate Point Cloud produced by a layer higher in the network's architecture.

We define each layer's input I as follows:

$$I := (P_I \; ; \; \phi),$$

where $P_I \subset \mathbb{R}^d$ is a set of points lying in a d-dimensional space, and $\phi \; : \; P_I \rightarrow \mathbb{R}^J$ is a function associating to each point $x_i \in P$ a feature vector in a J-dimensional space. On the other hand, we expect each convolutional layer to have an output O similarly defined as:

$$O := (P_O \; ; \; \psi),$$

where $P_O \subset \mathbb{R}^d$ as for the input, and $\psi \; : \; P_O \rightarrow \mathbb{R}^M$ is the function mapping each output point to a set of output features, lying in a M-dimensional space. Note that, while both P_I and P_O lie in the same d-dimensional space, in general $|P_I| \neq |P_O|$. The behavior of the Convolutional Layer varies, depending on $|P_I|$ and $|P_O|$, in the following way:

- *if* $|P_I| = |P_O|$, our point-convolutionallayer is behaving like a non-strided convolutional layer. This is similar to what happens in a non strided Image Convolution: if we apply a non-strided discrete convolution to a (N, M) wide image, we obtain an output having the same (N, M) spatial dimension.
- *if* $|P_I| > |P_O|$, our point-convolutional layers behaves similarly to a strided discrete convolutional layer: as in images, we receive an output with reduced spatial dimension. In this thesis, we do not consider pooling, even though several works [51] [2] define Point-Pooling layers: instead, we use striding to reduce the cardinality of P_O in the various network layers.
- *if* $|P_I| < |P_O|$, our point-convolutional layer behaves like a deconvolution: this is useful, for example, in tasks like semantic segmentation.

Another important consideration is whether $P_O \subset P_I$ (or, if we are applying a deconvolution, $P_I \subset P_O$). In general, every algorithm can decide whether this is the case or not: in the case of PCNNEO, for example, $P_O \subseteq P_I$; The main advantage of choosing a subset $P_O \subseteq P_I$ as output Point Cloud is relative to the layer's implementation: having to sample a subset from a given Point Cloud can be much more efficient than defining a new output point set $P_O \nsubseteq P_I$. For instance, being all points known a-priori before training, it would be possible to know the distances from each point to the others before training, so that it wouldn't be necessary to compute the said distances on the run.

Finally, we briefly discuss the input feature space and the output feature space. As in the case of image CNNs, we expect that the number of each layer's output features M will be bigger (smaller) than the number of input features J in the case of convolution (deconvolution), depending on the architecture, the training data and the task that needs to be solved.

4.1.2 Convolutional layer's structure

In PCNNEO, Point Cloud convolution is defined as subsequent application of three different operators to the input:

1. **Extension** $\mathcal{E}_I$: starting from an input Point Cloud P_I, we define a continuous function generalizing it.
2. **Convolution** C: we apply a continuous convolution to the function produced by the extension operator, obtaining as a result another continuous function.
3. **Reduction** $\mathcal{R}_O$: we sample the Convolution operator's output function, obtaining a different Point Cloud that can be fed into the next layer of the network.

In formulae, the output of the convolution operator $C_{I,O}$ that maps the Point Cloud I to the Point Cloud O is obtained as:

$$C_{I,O} = \mathcal{R}_O \bullet C \bullet \mathcal{E}_I$$

In the next sections we describe the way these operators are defined. This model is quite different in its inner mechanism from the other models presented in this chapter; however, the overall scheme presented here - in particular regarding how to define input and output Point Clouds - is shared by the other convolutional models, namely, KPConv and ConvPoint.

4.1.3 Extension

Starting from the input $I := (P_I, \phi)$, the Extension operator should return a continuous function, defined over the entire $\mathbb{R}^d$:

$$\mathcal{E}_I : (P_I, \phi) \rightarrow C(\mathbb{R}^d, \mathbb{R}^J),$$

where with $C(\mathbb{R}^d, \mathbb{R}^J)$ we mean the collection of functions $f : \mathbb{R}^d \rightarrow \mathbb{R}^J$. The idea is that the Extension operator should return an interpolation of the feature function $\phi(\cdot)$, which is instead defined only over P_I. Let us recall what we discussed in Section 3.2 regarding the properties a Point cloud ML model needs to comply with:

- The algorithm should be invariant with respect to the order of the Point Cloud's elements.
- The algorithm should capture local and structures inside the Point Cloud.

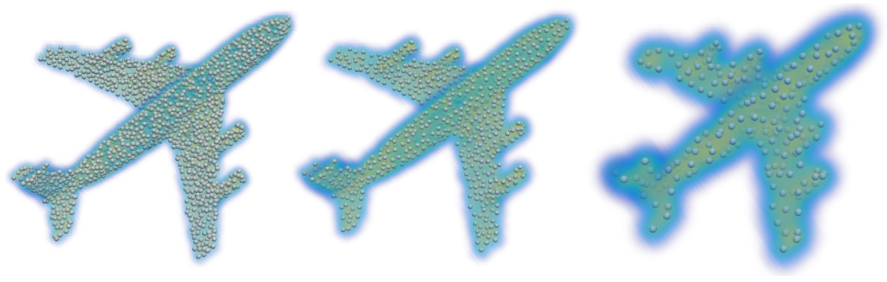

Figure 4.1: Image from [2] explaining how the extension operator works: the image is showing three Point Clouds (composed of 2048, 1024 and 256 points respectively) where each point is associated to a single, unitary feature. The colored area represent the continuous, d-Dimensional (here $d = 3$) function returned by the extension operator for each of the three Point Clouds. It is possible to note how the shape of said function is independent with respect to the cardinality of the Point Cloud that was used to generate it.

- The algorithm should be invariant to a certain reasonable set of geometrical transformations.

We can enforce the first two properties by the use of the Extension operator: to do that, the choice of the right model implementing $\mathcal{E}_I$ is crucial. PCNNEO uses a linear combination of RBF to implement the Extension, for three reasons: linear combinations of Radial Basis Functions were frequently used to interpolate a set of points [60] [23]; they are invariant with respect to the point order; they are also good, given a localized Radial Basis Function, to capture local structures. In formulae, the Extension operator becomes:

$$\mathcal{E}_I[\phi](x) = \sum_i \phi(x_i)\omega_i \, h(|x - x_i|),$$

where ϕ is the input feature function that needs to be interpolated, and $x_i \in P_I$ represents the spatial coordinates vector of each element contained in the input Point Cloud. Such linear combination is composed of a RBF $h(|x - x_i|)$ for each point $x_i \in P_I$: this means that if the input is composed of 256 points, our extension operator is computed as sum of 256 RBFs, each one centered on a different $x_i \in P_I$. Finally, ω is a parameter that depends on a given point $x_i \in P_I$, the weights of which contribute to the combination.

A possible choice for ω_i is:

$$\omega_i = \frac{1}{\sum_j h(|x_j - x_i|)} \quad x_j \in P_I.$$

This means that to weigh each point's x_i contribution relatively to the distance from the other points x_j: in this way, we are sure that the outliers have less weight than the inlier points.

Finally, to fully describe the Extension operator $\mathcal{E}_I$, we need to choose a suitable Radial Basis Function $h(|x - x_i|)$. PCNNEO uses a Gaussian function as RBF:

$$h(|x - x_i|) = e^{|x - x_i|^2/2\sigma^2}.$$

As we discussed in Section 3.1, a Gaussian Radial Basis Function is both localized and suitable to achieve universal approximation capabilities when used in a Radial Basis Function network. The main reason behind this choice, though, relies on how the convolutional operator is defined: as we shall see, it would not be possible to efficiently compute the convolutional operator without having defined $\mathcal{E}_I$ as sum of Gaussians.

4.1.4 Convolution

The definition of this convolution relies heavily on the specific way in which the extension operator $\mathcal{E}_I$ was computed. In this context, we denote the output of the Extension as:

$$f(x) = \mathcal{E}_I[\phi](x).$$

Our convolution operator $C[f](x)$ is defined as:

$$C[f](x) = \int_{\mathbb{R}^d} \sum_j f_j(y)\, \kappa_j(x - y)\, dy.$$

In this expression, each function f_j represents the j-th component of the function $f(x) = \mathcal{E}_I[\phi](x)$: in other words, each $f_j(y)$ contains the j-th feature of the point y. On the other hand, the filter is represented by the function:

$$\kappa_j \ : \ (\mathbb{R}^d) \rightarrow (\mathbb{R}^J, \mathbb{R}^M).$$

Recall that in this context, $\mathbb{R}^J$ is the input features' space, while $\mathbb{R}^M$ is the output feature's space: as expected, the kernel $\kappa(x - y)$ associates to each input spatial vector a set of weights representing how each input feature contributes to each output feature.

The main question that rises from this convolutional operator is how to compute the said integral efficiently. Having defined $f(y)$ as combination of Gaussian RBFs is useful to find a solution to this problem: [2] state that the convolution of two Gaussians $\Phi_\alpha(|x-\alpha|)$ and $\Phi_\beta(|x-\beta|)$ is proportional to a third Gaussian $\Phi_\delta(|x-\alpha-\beta|)$ where $\delta = \sqrt{\alpha^2+\beta^2}$. In formulae:

$$\Phi_\alpha(|x-\alpha|) * \Phi_\beta(|x-\beta|) \propto \Phi_\delta(|x-\alpha-\beta|). \tag{4.1}$$

Since the function $f(y)$ that needs to be convolved is already a Gaussian, it is possible to define a Gaussian kernel to make use of the rule above. For this reason, we define the kernel $\kappa(x-y)$ with the help of a proper Radial Basis Function Network:

$$\kappa(r) \;=\; \sum_l k_{j,l,m}\; h(|r-c_l|),$$

where j refers to the input feature, m refers to the output feature, r is the filter's argument, c_l are RBF centers and finally $k_{j,l,m}$ are the learnable parameters of our filter.

Having defined the convolution's kernel, we can rewrite the Convolution operator as:

$$C[f](y) = \sum_{i,j,l} \phi_{i,j}\omega_i k_{j,l,m} \int_{\mathbb{R}^d} \Phi(|x-x_i|)\Phi_\beta(|x-y-c_l|)\, dx. \tag{4.2}$$

Such integral is solvable by the use of (4.1); the only learnable parameters are $k_{j,l,m}$, which define the convolutional kernel. We expect the result of expression 4.2 to be a continuous function:

$$\psi_R(y) \;:\; \mathbb{R}^d \;\rightarrow\; \mathbb{R}^M,$$

assigning each point $y \in \mathbb{R}^d$ to a set of output features obtained by convolution. The next step is to sample a Point Cloud from this function, representing the output of our Convolutional Layer.

4.1.5 Reduction

The reduction operator is the simplest component of this Point Cloud convolutional layer. Its role is to return a Point Cloud from the function obtained from convolution operator. The only caveat is the following: the function $\psi_R(x)$ is defined over all $\mathbb{R}^d$, while we expect our output points P_O to be sampled from a surface. For this reason, a possible method is to resample the same points from the input Point Cloud, that is assumed to be correctly sampled. In this case, the reduction operator can be written as:

$$\psi(y) \;=\; R_O\,[\psi_R]\,(y) \;=\; \psi_R(y)\Big|_{P_O} \quad ; \quad P_O \equiv P_I.$$

It is also possible to sample from a subset of the original Point Cloud, obtaining a sort of strided convolution were $P_O \subset P_I$. Having both the elements belonging to the output O, namely the point set P_O and the output features function $\psi(x)$, we have completely defined PCNNEO's Convolutional Layer.

4.2 Kernel Point Convolution

The convolutional layer described in the previous section - known as PCNNEO - can achieve good performance when applied to different neural network architectures. On the other hand, though, PCNNEO is an indirect method: it operates on a continuous function defined starting from a Point Cloud. Most importantly, PCNNEO is not scalable: the Extension operator has quadratic complexity with respect to the number of points in P_I [51]. This derives from the fact that the definition of the Extension operator involves the entirety of the input Point Cloud.

For these two reasons, H. Thomas et al. Introduced a new convolutional layer called *Kernel point Convolution* (KpConv) [51], meant to define neighborhoods and convolutional kernel directly on Point Cloud's elements. This section aims to introduce the convolutional layer designed in KpConv, highlighting the differences between the previously presented models. In this context, we continue to refer to input I and output O in the same way as in Section 4.1.1; the considerations regarding the cardinality of input and output also hold. This convolutional Layer is also meant to be architecture agnostic, achieving good results using standard neural network architectures such as the ones used by image CNNs.

4.2.1 Key components of KP convolution

Let us consider a discrete image convolution operator with one input channel and one output channel:

$$\psi = (\phi * g)(y) = \sum_{x_i \in X} \phi(x_i)\, g(y - x_i). \tag{4.3}$$

In this operator, pixel positions decide the filter's value, and pixel features are weighted by the filter and summed together. We expect the same behavior in a point-convolutional layer: spatial coordinates will decide the filter's value for each point, while each point's input features $\phi(x) \in \mathbb{R}^J$ get convoluted into output features $\psi(y) \in \mathbb{R}^M$. This means that the key part of a point-convolutional layer is the filter's design. More precisely:

- the definition of the neighborhood X in which the convolution is applied. In image convolution, where the kernel is represented by a $(H \times W \times C)$ tensor, this is done by defining the kernel spatial dimensions H and W.
- the definition of a learnable model representing the function $g(\cdot)$. In image convolution we can easily enumerate all admissible filter values; in Point Convolution, where the possible filter arguments are infinitely many, the solution has to be different.

While in images each pixel's position is defined by relying on a matrix-like structure, such "grid" of possible configurations does not exist in the case of Point Clouds. Here, each point is free to assume any possible position vector in $\mathbb{R}^d$. This is the reason behind the complexity of a point-convolutional layer like KpConv. Another aspect that is trivial in Image Convolution but interesting in the Point Cloud case is how to choose where to compute convolution. In images, we know a-priori (given a particular stride) in which pixels apply our discrete convolution In Point Clouds though, being the input points P_I unordered, it is not possible to use the same strategy. In the following paragraphs, we shall answer these problems. First, we will discuss how to define a suitable neighborhood for our convolution. Second, we will define a learnable model representing our filter for each possible value $(y - x_i)$. Finally, we briefly discuss how to choose the output Point Cloud $P_O := \{y_o\}$

4.2.2 Defining the Neighbourhood

In the previous examples, the filter $g(y - x_i)$ defines by itself which possible arguments $(y - x_i)$ yield non-zero result. This is done explicitly in the case of images: given an output pixel y, we consider only pixels x_i for which $g(y - x_i)$ is defined. On the other hand, PCNNEO handles the issue in a similar way: being its convolutional filter $g(r)$ a sum of Gaussians, the filter's value $g(r)$ tends to zero for sufficiently large values of $r = y - x_i$. In the case of direct Point convolution, like in KPConv, it is useful to define directly which points participate in the computation and which not. This makes the network's implementation easier and also simplifies the definition of the filter function, so that it is not mandatory to have $\lim_{r \to \infty} g(r) = 0$. Two approaches are commonly used to define the neighborhood:

- To consider all points within a certain distance from the convolution's output. This approach's main drawback is that the neighborhoods may have different

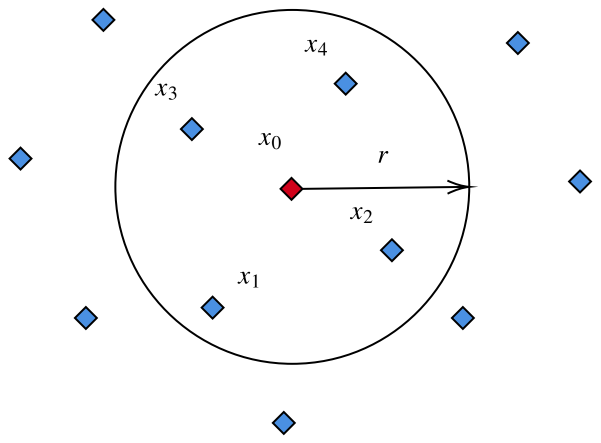

Figure 4.2: the image shows a neighborhood centered on point x_0 in a 2-dimensional Point Cloud. We can think of defining the neighborhood as the set of points where $||x_i - x_0|| < r$, but this solution produces neighborhoods with different cardinality. If we are sure that points in the Point Cloud are uniformly distributed along the sampled surface, we can rely on K-NN algorithms to define neighborhoods that have all the same cardinality.

cardinality depending on where the convolution is computed. While not a major issue, this fact makes the convolutional layer's implementation more difficult [4] [53].

- To consider the K points closer to the convolution's output point. This approach, usually performed using K-Nearest-Neighbors algorithms, is quite common because it produces neighbors with the same cardinality. However, it is not employable in non uniformly sampled Point Clouds [51] [4].

In the case of KPConv, the chosen approach is the first one. We will also see that a similar model (called ConvPoint [4]) makes use of the second approach.

4.2.3 Defining the Kernel

The second key issue in designing a direct point-convolutional filter is the definition of a learnable model representing $g(y - x_i)$. In this case, several approaches were proposed. For example, SpiderCNN [56] uses a polynome where the learnable parameters are the polynome coefficients; DPCNN [53] uses a Multi Layer Perceptron; KPConv [51], instead, uses Radial Basis Function Network (RBFN) conceptually very similar to the

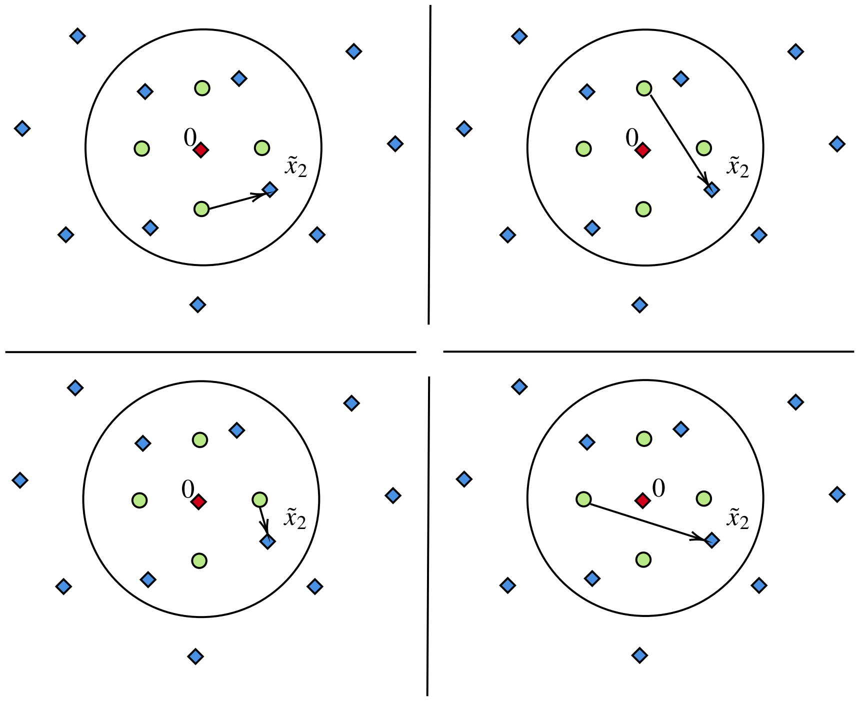

Figure 4.3: the image shows the same neighborhood presented in figure 4.2, with the addition of four RBF centers, represented in green. Suppose of having to compute $g(\tilde{x}_2)$. To do that, the distances betweeen all centers and have to be computed. After this, we compute each $h(||\tilde{x_2} - c_l||)$ and multiply it by the corresponding weight w_l.

one used by PCNNEO. In formulae:

$$g(\tilde{x_i}) = \sum_{l=1}^{|K|} w_l \, h(||\tilde{x_i} - c_l||) \quad where \quad \tilde{x_i} = y - x_i. \tag{4.4}$$

Here, the set of learnable parameters is $K := \{(w_l; c_l)\}$.Each element of K is a couple composed by:

- a weight vector $w_l \in \mathbb{R}^M$ (i.e., a vector with one component for each output feature), associated to each RBF.
- the center $c_l \in \mathbb{R}^d$ associated to each RBF, lying in the d-dimensional space in which the Point Cloud's elements are defined.

We recall that the key components of a Radial Basis Function network are:

- RBF shape.

- Center's positions.
- Number of Radial Basis functions used.

While the number of radial basis functions (i.e., the kernel's cardinality) is an hyper-parameter, we focus on the other two points. Differently from the Gaussian RBFs used in PCNNEO, KPConv uses the localized linear correlation as Radial Basis Function [51]:

$$h(||\tilde{x_i} - c_l||) = \max\left(0, 1 - \frac{||\tilde{x_i} - c_l||}{\sigma}\right) \quad where \quad \tilde{x_i} = y - x_i.$$

Here, σ can be considered a hyperparameter. It is easy to notice that such RBF satisfies all the properties (3.1), needed to achieve universal approximation.

Concerning the centers' position, it is relevant to notice that each center c_l has to be defined in relative coordinates with respect to the output y. In this way, we make the RBFN capable of describing the filter $g(\tilde{x_i})$ inside the given neighborhood. The problem then becomes how to choose these relative positions. What we present now is the solution implemented in *KPConv-Rigid*, the simplest “flavor" of KpConv, but other variations are possible. Here the set of centers $\{c_l\}$ is fixed *a priori* before training, by solving an optimization problem. The authors associate to each center a repulsive potential similar to the electric potential: in this way, each center applies a repulsive force to the others others. It is then assigned an attractive potential between the neighborhood's origin and the centers. The objective is to minimize the total energy of the system. An alternative to this approach may be to learn center positions by using gradient descent, like in [4].

4.2.4 Defining the output

The problem of choosing the point set P_O is theoretically not trivial, as many different solutions can be designed and applied. A possible solution is to rely on points that are already part of the input Point Cloud P_I. This solution allows the model to precompute distances between points without harming the layer's performance. In the case of KpConv, output points are sampled without replacement from the original Point Clouds. Other sampling strategies can be defined, especially when dealing with peculiar environments, like in the case of non-uniformly sampled PCs.

4.2.5 Rigid or deformable kernels

In 2017, *Deformable Convolutional Networks* by J. Dai et al [9] introduced an extension of the traditional image convolution operator. In a traditional image-convolutional NN, deformation invariance is achieved by data augmentation and by using large models with a high number of parameters [9]. On the other hand, Deformable CNNs are intended to work with simpler architectures and less data by making the convolution operator invariant to certain geometrical deformations. An important contribution of KpConv is to apply the idea of deformable convolution to Point Clouds; this is achieved by modifying the convolutional operator described before.

In order to achieve deformation invariance, *KPConv-deformable* defines a vector field:

$$\Delta : \mathbb{R}^d \rightarrow \mathbb{R}^d,$$

associating to each location $c \in \mathbb{R}^d$ d-dimensional vector called shift $\Delta(c)$. The shift vector $\Delta(c)$ represent the translation needing to be applied to the center c in order to "deform" the kernel, making it adaptable to deformed or scaled Point Clouds. In formulae, the kernel $g(\cdot)$ becomes:

$$g_\Delta(\cdot) = \sum_{l \leq |K|} w_l \ h(||\tilde{x_i} - \Delta(c_l) - c_l||). \tag{4.5}$$

To define said field vector Δ, KpConv makes use of another Radial basis function network, exploiting its universal approximation capabilities. In particular, each shift is computed by applying a *KpConv-Rigid* layer to the input points and producing a shift vector for each position c. This shift is later used to compute the value of *KpConv-Deformable* filter $g_\Delta(\cdot)$ on the same input points.

KPConv [51] also describes a regularization loss needed to improve the performances of its deformable convolutional operator, that will not be presented here: the aim of this section is to prove that we can straightforwardly extend many different image-convolutional operators, such as the one shown in [9], to work directly with Point Clouds.

4.3 Convpoint

Convpoint by A. Boulch [4] is a third example of point-convolutional layer, like KpConv operating directly on points. The overall approach is very similar to the one described in KpConv, with a significant difference: ConvPoint tries to completely separate the kernel spatial "structure" (the one handling points' positions) from the semantic one (the one handling features). To better explain this idea, let us think about an image convolution. Here, the kernel is represented as a matrix: a set of weights distributed in a grid-like structure. In this context, the spatial structure is the one given by the grid, while the weights give the semantic associated to said structure. Let us take into consideration the convolutional filter defined by KpConv, presented in (4.4), and here rewritten for multiple output features $\{\psi_o\}$:

$$g_{\psi_o}(\tilde{x_i}) = \sum_{l<=|K|} w_{lo}\, h(||\tilde{x_i} - c_l||) \quad where \quad \tilde{x_i} = y - x_i.$$

In this formula, for each output feature ψ_o we use a radial basis function network to define the filter. In this the only parameters shared between the different output features are the network centers. This approach is comparable to what happens with the images: here, the spatial structure is given by the centers and is shared between different features, while features themselves are combined using specific weights that rely on it. Is it possible to expand the said spatial structure by sharing more than the mere centers' position between different output features? Convpoint tries to answer the following question. By answering it, we follow the same structure used when describing KPConv, firstly by explaining how to define which point participates in convolution and then defining the kernel itself. We continue to refer to Section 4.1.1 when talking about the layer's input and output; moreover, the same considerations about being an architecture agnostic layer presented for KpConv and PCNNEO also holds when dealing with ConvPoint.

4.3.1 Neighbourhood definition

There are two main ways to choose which points participate in convolution. For example, KpConv chooses to select all the points closer to the output point than a certain fixed distance to join in convolution. Many approaches try to obtain something similar: a hypersphere of fixed radius containing all neighboring points.

Convpoint chooses to use the other approach presented in Section 4.2.2: under the assumption that points clouds are uniformly sampled from the d-Dimensional surfaces they represent, Convpoint select K-nearest neighbors as support for the convolution's filter. To obtain a hypersphere like in KpConv, ConvPoint also adopts the following steps:

- for each point, take its relative coordinates with respect to the output point.
- Normalize said relative coordinates so that the point farthest from the origin (i.e., the output point) is distant one unit.

By applying such normalization, we obtain a neighbor contained inside a hypersphere with a unitary radius. This is useful when needing to define the filter function, since all the neighborhoods are similar in scale; Most importantly, this allows us to efficiently achieve scale invariance without having to lean to deformable filters as in KpConv.

4.3.2 Kernel definition

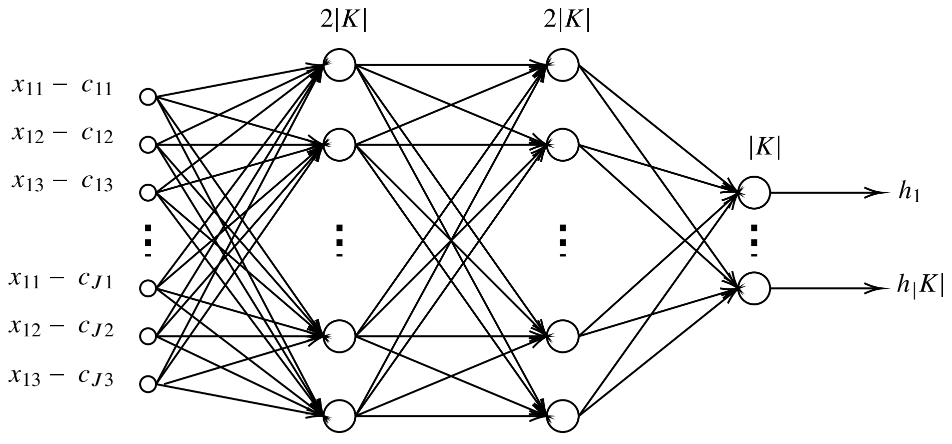

Figure 4.4: the image shows the 3-layer MLP used to compute the function $h_l(\cdot)$ in ConvPoint's convolutional layer. The first layer has dimension $(3|K|, 2|K|)$, the second layer has dimension $(2|K|, 2|K|)$, and the third layer has dimension $(2|K|, |K|)$. Note that in this case, every layer of this Multi Layer perceptron depends on the cardinality of K, that is the number of center points. Each $h_l(\cdot)$ is shared among all output features.

Convpoint's kernel is similar to the one presented by KpConv. As in KpConv, the kernel parameters are those in the set:

$$K = \{w_l;\, c_l\}_l.$$

In Convpoint the kernel is made of couples with one weight vector and one center point associated to it. We use these parameters to compute the filter $g(\cdot)$, in a way similar to the one presented for Kernel Point convolution:

$$g_(\tilde{x_i}) \;=\; \sum_{l<=|K|} w_l \, h_l(D) \quad where \quad \tilde{x_i} = y - x_i. \tag{4.6}$$

This equation differs from the one presented in (4.4): in KpConv the function $h_l(\cdot)$ was a radial basis function, while in ConvPoint we define $h_l(\cdot) \;:\; \mathbb{R}^{d\times|K|} \longrightarrow \mathbb{R}$ as the function:

- Receiving as input the set of the distances between the neighboring point x_i and all the center points $D := \{\tilde{x_i} - c_l\}_l$.
- Returning a value shared between all output features.

In this context, $h_l(\cdot)$ is no more a radial basis function nor the centers $\{c_l\}$ are employed to define an RBFN. This is not only because the set D is made by difference vectors instead of scalar distances, but also because all the centers are needed to compute the output of $h_l(\cdot)$. In KpConv, on the contrary, every radial basis function is influenced only by the value of its associated center.

The way to implement said function $h_l(\cdot)$ is crucial for Convpoint.In particular, the authors employ a 3-layer Multilayer Perceptron, as in figure 4.4. Note that we define a unique Multilayer Perceptron for all functions $h_l(\cdot)$.

4.3.3 Separating spatial and semantic structure

As we discussed while introducing Convpoint, this method's true novelty is to separate spatial and semantic part of the convolutional layer, having all features sharing some sort of underlying spatial structure. In the case of KpConv and PCNNEO, the portion of the network shared by different features was each RBF center's position. In this case, other than the centers, there is also a shared learned function (computed by a multilayer perceptron). In general, we expect the MLP to produce a spatial representation yielding much more information than the mere RBF centers used in KpConv. The main drawback is that said MultiLayer Perceptron is tied, in terms of input and output dimension, to the number of weight vectors w_l representing the semantic structure of the convolutional layer. The input of the MLP is in fact defined as a set $\{\tilde{x_i} - c_l\}_l$, that depends on c_l, and $|\{c_l\}| = |\{w_l\}| = |K|$ because of the definition of the set $K := \{(w_l, c_l)\}_l$. In other words, we cannot increase the MLP input dimension without increasing the number of weight vectors, which means that the spatial structure is tied in complexity to the semantic one. We anticipate that the approach proposed in this thesis is to define a model that removes the previously mentioned limitation and allows the spatial structure to be arbitrarily complex without increasing the overall complexity of the convolutional layer.

Chapter 5

Proposed Approach

This thesis proposes a Novel point-convolutional layer, to be applied in both multiclass and One-Class (OC) classification tasks. In this chapter, we shall present how such novel layer is defined, what its peculiarities are and how to employ it in both previously cited tasks. In discussing our approach, we will refer to two already discussed models, KpConv and Convpoint. This last model in particular is the foundation upon which our novel convolutional layer is built. The idea is to explore ConvPoint's key feature, which is to share a common spatial representation between different features.The drawback of convpoint is that such spatial representation is learned by a Multi Layer Perceptron whose dimensions are tied to the number of feature weights. This is the issue our model aims to solve: to be able to scale independently the sublayer learning the spatial representation from it combining features together, both in terms of dimensionality and architecture. Discussing our solution, we investigate different alternatives on how to implement said layer. In the second part of this chapter, we shall also discuss how to adapt our layer to be used for OC classification inside a so-called *Deep Support Vector Data Description* (Deep SVDD) architecture [42]. As we shall point out, networks used in Deep SVDD must comply with very specific constraints that must be considered when defining a convolutional layer. In the next chapter we will prove that our convolutional layer is functional and that it has performances competitive to the already presented point-convolutional methods.

5.1 Problem formulation

In the following subsections we will describe the problem of defining a convolutional layer. First of all, we shall define the kind of input and the kind of output that our layer needs to accept and return. Then, we shall describe the kind of convolutional operator our layer should apply to the input point cloud. In particular, we talk about the difference between point convolution and traditional discrete and continuous convolutions, by making special reference to the elements composing them. Moreover, we describe how to choose points belonging to the output point cloud of our convolutional layer: while this is not a problem for the convolutional operator, it is crucial in the layer's definition and performance. Finally, we discuss some non-functional requirements that we expect our layer to comply with.

5.1.1 Input and Output definition

We reprise the definition of input and output of a point-convolutional layer from section 4.1.1, where we described input and output in the case of Point Convolution by Extension Operators [2]. In particular, each convolutional layer will receive as input a Point cloud I, represented as a couple:

$$I := (P_I \; ; \; \phi),$$

where $P_I \subset \mathbb{R}^d$ is a set of points lying in a d-dimensional space, and $\phi \; : \; P_I \longrightarrow \mathbb{R}^J$ is a function associating to each point $x_i \in P$ a feature vector in a c-dimensional space. On the other hand, we expect each convolutional layer to have an output O similarly defined as:

$$O := (P_O \; ; \; \psi),$$

where $P_O \subset \mathbb{R}^d$ as for the input, and $\psi \; : \; P_O \longrightarrow \mathbb{R}^M$ is the function mapping each output point to a set of output features, lying in a M-dimensional space. Note that, while both P_I and P_O lie in the same d-dimensional space, in general $|P_I| \neq |P_O|$. The behavior of the Convolutional Layer varies, depending on $|P_I|$ and $|P_O|$, in the following way:

- *if* $|P_I| = |P_O|$, our point-convolutional layer is behaving like a non-strided convolutional layer. This is similar to what happens in a non strided Image Convolution. In fact, if we apply a non-strided discrete convolution to a (H, K) wide image, we obtain an output having the same (H, K) dimension.

- *if* $|P_I| > |P_O|$, our point-convolutional layers behaves similarly to a strided discrete convolutional layer. As in images, we receive an output with reduced spatial dimension. In this thesis, we do not consider pooling, even though several works [51] [2] define Point-Pooling layers. Instead, we use striding to reduce the cardinality of P_O in the various network layers.
- *if* $|P_I| < |P_O|$, our point-convolutional layer behaves like a deconvolution. This is useful, for example, in tasks like semantic segmentation.

Differently from what we discussed in section 4.1.1, where no constraint was defined regarding P_I and P_O, we impose that for each layer either $P_O \subseteq P_I$ or, in the case of deconvolution, $P_O \subseteq P_I$. This is fundamental for our convolutional layer, since it allows us to precompute the distances between each couple of points involved in the computations, thus saving an important amount of run time without losing any generality in terms of theoretical properties.

We expect that each output feature vector $\psi(y)$ with $y \in P_O$ is obtained through application of a point-convolutional operator over the input point cloud (P_I, ϕ). In section 5.1.2 we shall define a suitable convolution operator that can be applied to this kind of data.

5.1.2 Convolutional Operator definition

the primary goal of our convolutional layer is to define and implement a convolutional operator applicable on point clouds in the same way as a discrete convolutional operator is applicable on images. For the sake of simplicity, let us consider the case in which $\phi(\cdot)$ is a scalar function; since point clouds are defined in a continuous space $\mathbb{R}^d$, a possible starting point is the definition of convolution over continuous functions:

$$(\phi * g)(y) = \int_{\mathbb{R}^d} \phi(x)g(y - x)dx. \tag{5.1}$$

This kind of operator is not directly applicable on point clouds, because the function ϕ is defined only on a discrete set of points $P_I \subset \mathbb{R}^d$. To implement such continuous operator, it would be necessary to extend the domain of function ϕ on the entire space $\mathbb{R}^d$. The problem of defining a continuous feature function starting from a point cloud is not trivial. For example, in the case of PCNNEO [2] the extension of ϕ over the entire space $\mathbb{R}^d$ takes quadratic time complexty [51]. This suggests us that a better approach would be to define a *discrete* point-convolutional operator, like in the case of

KpConv and ConvPoint (see section 4.2 and 4.3). Such operator would be defined as:

$$(\phi * g)(y) = \sum_{x_i \in X \subset P_I} \phi(x_i) g(y - x_i), \tag{5.2}$$

where ϕ is discrete as defined in section 5.1.1. Note also that, where in (5.1) the sum was defined over all the space $\mathbb{R}^d$, in this case we are restricting it on a specific neighbourhood $X \subseteq P_I$. This is very similar to what happens in traditional discrete convolutions, where such neighbourhood is implicitly defined as the support of the filter $g(\cdot)$.

Differently from image convolution, where the filter is discrete, for point clouds we need to define a continuous filter $g(\cdot)$. This due fact that the filter's argument $y - x_i$ is by definition continuous, being both $y, x_i \in \mathbb{R}^d$; there are two solutions that are possible here:

- to define filter $g(\cdot)$ as a compact support function. In this way, we can implicitly define $X \subseteq P_I$ as a set of points $\{x_i \in P_I\}$ such that $y - x_i \in \text{supp}\ (g)$.
- to define set $X \subseteq P_I$ explicitly and leave no constraints on the shape of $g(\cdot)$. In this case, the filter's support will be implicitly limited by the fact that the filter will be only evaluated for a certain specific set of arguments.

In this thesis we consider the second option. We impose no constraints on the shape of the filter $g(\cdot)$, but we require to define a third element, a window or a neighbourhood X inside which the convolution output $\psi(y)$ is computed.

Summarizing, the main differences between Image Convolution and Point Convolution are the following:

- The domain of the function $\phi(\cdot)$ is discrete, while the filter $g(\cdot)$ is continuous in $\mathbb{R}^d$. In case of image convolution both functions are discrete.
- The point-convolutional operator is defined for a given neighbourhood X, while in case of image convolution such neighbourhood is implicitly defined by having $g(\cdot)$ with compact support.

Having defined such convolutional operator, we expect our convolutional layer to specifically learn the shape of the continuous filter $g(\cdot)$. The goal is then to represent such filter by a learnable model, able to represent also complex functions. This is the aspect in which we shall spend more words, also by making specific references to already existing state of the art methods.

5.1.3 Output point set construction

Having described how to compute output features, we now express some considerations regarding the construction of the output point set P_O. As already stated in section 5.1.1, we expect that $P_I \subseteq P_O$ (or $P_O \subseteq P_I$ in the case of deconvolution); on the other hand we never discussed what elements $y \in P_O$ we should include in the output point set. We require that the distribution from which the output points are sampled matches the distribution from which the input points are sampled. In other words, the two point sets should be sampled from the same surface. This allows the output set to carry the same spatial information encoded in the input set through the entire network.

In order to achieve this result, several strategies can be implemented. We enforce such property by sampling P_O from P_I using an uniform sampling strategy without replacement. We shall discuss this aspect in section 5.2, where we discuss our proposed convolutional layer.

5.1.4 Other requirements and constraints

Other than the functional requirements regarding what kind of output we want to obtain, we also state some important non-functional requirements our solution needs to comply with:

1. our convolutional layer needs to be architecture agnostic. It should be usable in the context of any well performing image CNN architecture.

2. our layer should be modular, so that it is possible to extend it by modifying behaviour of certain components without changing the overall layer architecture. This requirement is important with respect to the other State of the Art methods, since most of the layers previously presented are atomic and thus cannot be easily extended or modified.

3. we expect our layer to be implementable with a well-known deep learning framework such as Tensorflow [31] or Pytorch [36].

4. our layer to be comparable to the existing state of the art methods in terms of training convergence time.

5. we expect our layer to be comparable to the existing state of the art methods in terms of memory consumption.

Some of these requirements are related to the theoretical properties, while the others are related to the implementation of the convolutional layer. We shall reference these requirement both in this chapter and in chapter 6, when discussing experiments and benchmarking.

5.2 Proposed convolutional Layer

In defining our convolutionl layer,we follow an approach similar to Convpoint [4]: to define a filter $g(\cdot)$ that shares information between different features. To better understand this idea, let us rewrite the convolution operator in (5.2) in its more general form, with j input features, m output features and i input points:

$$\psi_m(y) = (\phi * g)(y) = \sum_{j=1}^{J} \sum_{i=1}^{|X|} \phi_j(x_i) g_{jm}(\tilde{x}_i) \quad where \quad \tilde{x}_i = y - x_i. \tag{5.3}$$

Here each filter's component $g_{jm}(\cdot)$, defining the relation between the j-th input feature and the m-th output feature, is uncorrelated with the other filter's components $g_{ab}(\cdot)$ with $a \neq j$ and $b \neq m$. our idea is to explore whether it is possible to share some information between them. For this reason, we have decided to implement each filter's component $g_{jm}(\tilde{x}_i)$ as subsequent application of two different functions over $\tilde{x}_i$:

$$g_{jm}(\tilde{x}_i) = f_{jm}(\ s(\tilde{x}_i)\ ). \tag{5.4}$$

Here the first function, that is:

$$s(\cdot) \quad : \quad \mathbb{R}^d \to \quad \mathbb{R}^K,$$

is shared among all input and output features. The idea is having $s(\cdot)$ mapping each difference vector $\tilde{x}_i = y - x_i$ to a vector $s(\tilde{x}_i) \in \mathbb{R}^K$, where the dimension K is an hyper-parameter.

The second function, namely:

$$f_{jm}(\cdot) \quad : \quad \mathbb{R}^K \to \quad \mathbb{R},$$

is different for each features' couple (ϕ_j, ψ_m); it's goal is to transform the vector $s(\cdot) \in \mathbb{R}^K$ by modifying it with information related to the corresponding input and output feature.

The next step is to describe how to implement such filter inside a CNN layer. We shall propose some alternatives on how to implement both the functions $s(\cdot)$ and $f_{jm}(\cdot)$, and describe other relevant aspects of our convolutional layer. Finally we shall also describe our layer's properties and advantages, by making explicit comparison to the previously described Convpoint [4] model.

5.2.1 A composite convolutional layer

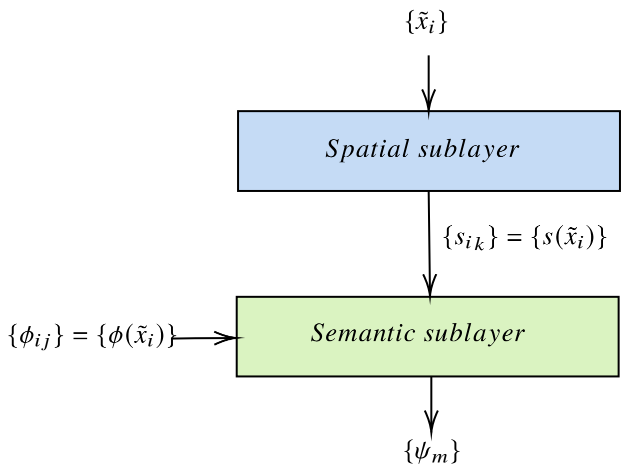
$\{\tilde{x}_i\}$
Spatial sublayer
$\{s_{ik}\} = \{s(\tilde{x}_i)\}$
$\{\phi_{ij}\} = \{\phi(\tilde{x}_i)\}$
Semantic sublayer
$\{\psi_m\}$

Figure 5.1: the simple structure of our proposed layer. Both the spatial and semantic sublayer may be defined in different ways: the only constraint is that the spatial layer should output a set of "spatial descriptors" $\{s_k\}$ for each point $\tilde{x}_i$; on the other way, The semantic layer should receive as input both the features $\{\phi_j\}$ and the spatial descriptors $\{s_k\}$ for each point $\tilde{x}_i$.

In order to define the functions $s(\cdot)$ and $f_{jm}(\cdot)$ described in this section's introduction, we propose a novel convolutional layer obtained by combining together two different sublayers:

- a *spatial sublayer* receiving the output's neighbourhood $\{\tilde{x}_i\}$ in relative coordinates and will return a set of spatial descriptors $\{s_{ik}\}$ for each point. Note that from this point we refer to $\{s_{ik}\}$ to mean both the output of function $s(\tilde{x}_i)$ and of the spatial sublayer implementing it.
- a *semantic sublayer*, receiving in input both the spatial descriptors and the features associated to each point $_i$ and combine them together; the result would be said point's contribution to the convolution. In other words, the semantic sublayer both defines the function $f_{jm}(\cdot)$ and combines it with the feature $\phi_j(x_i)$ in order to obtain the contribution of point x_i to the definition of $\psi_m(y)$

A proper convolutional composite layer should comply with (5.3) and (5.4), but the two sublayers we propose can also perform operations that are not compliant with the said definition. For example, if the descriptors $\{s_{ik}\}$ are a function of the entire neighbourhood $\{\tilde{x}_i\}$ instead of only a given point $\tilde{x}_i$, the overall layer would stop being convolutional; on the other hand, if the semantic sublayer does not combine $\phi_j(x_i)$ and $f_{jm}(\cdot)$ by multiplying them together, the resulting composite layer will also not be convolutional.

Before presenting some alternatives on how to define these two sublayers, we explain how to define output points $y \in P_O$ and the corresponding neighbourhoods. The process of output and neighbourhood definition is composed by these steps:

- choice of the output points, by sampling them with replacement from the input point cloud P_I. For each sampled point, we lower its resampling probability by a factor of 10.
- definition of the neighbouring points for each output point by using KNN algorithms. The number for neighbouring points can be considered a hyper-parameter.
- neighbourhoods' normlization, such that: $\max(||\tilde{x}_i||) = 1$. In this way, we also achieve scale invariance without the need of deformable convolutions like in KpConv [51].

As done previously, we consider $\tilde{x}_i$ to be the neighbouring point x_i written in relative coordinates with respect to the output point y. In this way, the output point is the origin of the reference frame for the elements in the set $\{\tilde{x}_i\}$.

The final consideration is that we assume the input point clouds to be uniformly sampled from the given surfaces. This is important for both the fact that we are using K-NN the neighbouring points selection and the fact that we are normalizing neighbourhoods. If that assumption is not verified, the neighbourhoods would have different spatial scales depending on their output point and the points' density around it: we need to avoid this situation in order to allow our filter to learn consistently [19].

5.2.2 Proposed Spatial layers

In our model, the spatial sublayer outputs a set of descriptors $\{s_k\}$ for each input point $\{\tilde{x}_i\}$. hared between all features. We recall that, with a slight abuse of notation, we indicate as $\{s_{ki}\}$ the set $\{s(\tilde{x}_i)\}_i$ of all spatial descriptors for all points. Several solutions can be adopted to model $s(\cdot)$. In the following paragraphs we shall describe few alternatives, some of which we implemented and tested.

RBFN-based spatial layer

One possible and simple solution is to build the function $s(\cdot)$ defined in the previous paragraph with the help of a radial basis function network. In this case, we would have that:

$$s_k(x_i) \;=\; \sum_{l}^{L} v_{kl}\, h(||\tilde{x_i} - c_l||). \tag{5.5}$$

Note that we share the centers between each descriptor s_k: in this way, we are defining all the descriptors with the help of a single multi-output RBFN in a way similar to what Convpoint [4] does with its MLP. It is also possible to define each s_k with the help of a separate RBF with separate centers.

This approach is similar to what has been proposed in [51]. For example, let's complement our spatial sublayer with a linear semantic sublayer as defined in section 5.8. in this case convolution will become:

$$\psi_m(y) = \sum_{j,i,k,l} \phi_j(x_i)\; w_{mjk}\; v_{kl}\; h_l(\tilde{x}_i) \quad where \quad \tilde{x}_i = y - x_i. \tag{5.6}$$

Let $\kappa_{mjl} = \sum_k w_{mjk}\; v_{kl}$.The convolutional operator would become:

$$\psi_m(y) = \sum_{j,i,l} \phi_j(x_i)\; \kappa_{mjl}\; h_l(\tilde{x}_i) \quad where \quad \tilde{x}_i = y - x_i,$$

which is similar to how KpConv [51] defines its convolutional layer. The difference is subtle: while in KpConv all weights κ_{mjl} defining the kernel are independent from each other, in our case those weights are correlated along the input features dimension. In other words, we are sharing spatial information between the input features, which is exactly one of the goals of this novel convolutional layer we are building.

It is also possible to introduce another element that distinguishes our convolutional layer with respect to the previous works: we can insert a non-linearity at the end of our spatial layer, i.e. a module that accept the spatial vector s_i associated with each input point $\tilde{x}_i$ and applies a non-linear function to it. In our experiments we have decided to use the Rectified Linear Unit (ReLU), since nowadays it is the most frequently used non-linearity in deep neural networks, and it has been proved to enable faster training in many deep learning tasks [15]. As we shall discuss in 5.4, the choice of Rectified linear units as non-linearity between spatial and semantic layer it is also useful when adapting our model to OC classification.

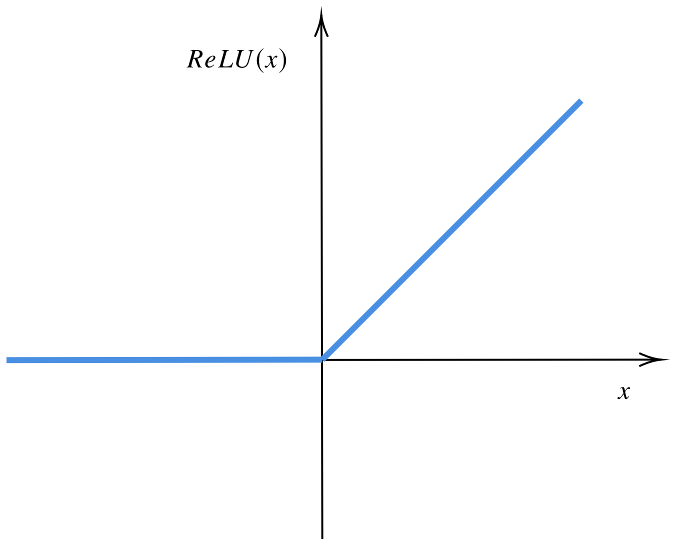

Figure 5.2: Plot of the Rectified Linear Unity function. As the image shows, $ReLU(x) = 0 \iff x \leq 0$; $ReLu(x) = x \iff x > 0$.

RBF-based spatial layer with learned RBF

We also propose another alternative spatial sublayer, that makes use of Radial Basis Function Networks like the previous spatial sublayer. As we discussed in section 3.1, Radial basis function networks can achieve universal approximation capabilities if an arbitrary number of centers is employed, independently from which localized RBF is chosen. Nonetheless, if only a fixed number of centers is available, it is possible that different Radial Basis functions yield different performances. This is the point that we aim to investigate: would it be possible and useful to learn the shape of the radial basis functions?

Inspired by the work presented in SpiderCNN [56] and SplineCNN [14], we tried to implement such RBFN with learned radial basis function by deciding to represent the learned RBF by using a Fourier expansion of the type:

$$h(||\tilde{x}_i - c||) \approx \frac{a_0}{2} + \sum_{n}^{T} a_n \cos(n||\tilde{x}_i - c||) + b_n \sin(n||\tilde{x}_i - c||) \tag{5.7}$$

In this case, the learnable parameters are represented by the bias term a_0, plus the couples $\{(a_n, b_n)\}$. The first consideration regarding such Radial Basis Function is that neither it is compliant with the constraints defined in (3.1), nor is it localized. This is generally not a problem, because the formulation of 5.7 is more general, i.e.

can represent both the function that is compliant with contraints 3.1 and the function that is not. On the other hand, the problem of having a localized radial basis function is solved implicitly. That's because we are constraining the support of each RBF, by preemptively selecting the neighbourhood via K-NN. given that no element outside the neighbourhood would participate in the convolution, each RBF will return non-zero values only for points "close" to its center, i.e. inside the neighbourhood. Another important consideration about (5.7) is that it is periodic. This property is generally not suited to a Radial Basis Function, and thus it can be a problem. We solve it by imposing a base Period that doubles the neighbourhood radius. In this way, we are sure that every point in the neighbourhood will fall inside the same period of the RBF. The way in which we enforce such constraint is by inserting an hyper-parameter representing frequency in (5.7). the new equation looks like:

$$h(||\tilde{x}_i - c||) \approx \frac{a_0}{2} + \sum_{n}^{T} a_n \cos\left(n\frac{f||\tilde{x}_i - c||}{2\pi}\right) + b_n \sin\left(n\frac{f||\tilde{x}_i - c||}{2\pi}\right).$$

Considering that we normalize the neighbourhood so that the farthest point from the output has distance 1 to the origin, we can consider as suitable parameter f to be 0.5. It may also make sense to modify said parameter, as long as it guarantees that the radial basis function is not periodic inside the given neighbourhood.

Allowing the network to optimize the radial basis function's shape should help obtaining good performances, specially when the number of RBF centers is low.

Other possible Spatial layers

While until now we only proposed spatial sublayers defined with the help of a Radial Basis Function network, other tools could also be used. For example, it is possible to use a multi-layer perceptron similarly to Convpoint [4]; an alternative could be to define a Multi-layer-perceptron similar to the one of PointNet [37], taking as input each point's spatial coordinates and directly returning the point's spatial descriptor. Other possible solutions are to model the filter directly, with the use of d-Dimensional expansions like the Fourier expansion or the Taylor expansion; again, it is possible to use splines like in [14].

In choosing which spatial sublayer to implement though, several aspects should be taken into consideration:

- *Robustness during training*. Certain spatial sublayers can yield less robust performances during training, possibly subject to relevant fluctuations in terms of loss function and measure of merit. we shall deal with this aspect in chapter 6.
- *proneness to overfitting*. Not all spatial sublayers are equally prone to overfitting. In general, the more the spatial sublayer is powerful, the more it is likely to overfit. we shall also discuss about this aspect in chapter 6.
- *Resources usage*. We tried alternative spatial layers, describing the neighbourhood directly as fourier or taylor d-dimensional expansion. unfortunately these spatial sublayers suffered of high memory usage, because of the complex and custom way in which they were computed. This made them unsuitable to be compared with other RBFN or MLP based Sobek Spatial sublayers.
- *Time usage*. Together with a layer's spatial complexity, it is important to evaluate the corresponding time complexity.Time complexity and spatial complexity are often in trade off. This means that some time it is possible to reduce memory usage by sacrificing a certain amount of time per training iteration.

The last two points in particular are strictly related to the sublayer's implementation: even though we are not discussing here about how to algorithmically implement each layer, it is important to consider that both performances and usefulness of a convolutional layer also depend on the quality of the implementation. Some possibly sound solutions, like the over cited d-dimensional expansion to used to define the spatial sublayer, are not easily implementable and thus cannot be compared to other, simpler solutions like those based on RBFNs and MLPs.

5.2.3 Proposed Semantic Sublayers

After presenting some possible spatial sublayers, we now investigate how to define the semantic sublayer.

This component has two main roles: the first one is to differently weight each couple of input and output features; the second one is to combine spatial descriptors and features together. In this context, the filter is defined by combining both the sublayers, while the the convolution itself is computed by the semantic one. In order to preserve the convolutional structure, each output feature should be computed as a sum of each input feature weighted by a certain filter value, for each input point.

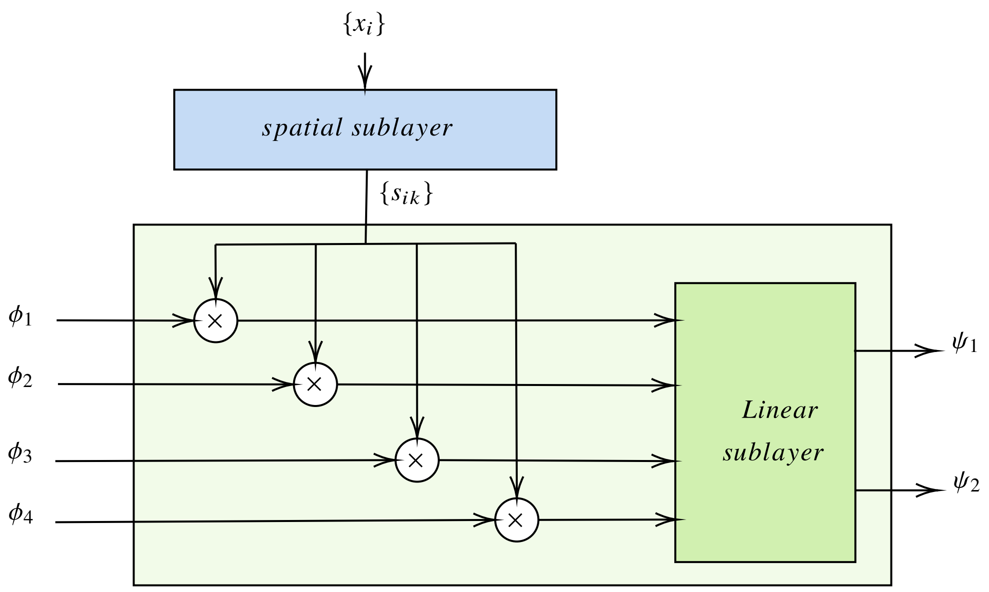

Figure 5.3: An example of composite convolutional layer with 4 input features and 2 output features. Here we highlight the structure of a possible semantic layer (in green). In order for the layer to be convolutional, we assume that each spatial descriptor given in the output by the spatial sublayer is a function of only one neighbouring point. We multiply element-wise each descriptor vector for each feature associated with the relative neighbouring point, and then feed the resulting matrix to a traditional linear layer.

Combining in other ways features and spatial descriptors obtained in input would yield a non-convolutional layer.

In this context, we present two alternatives for defining a suitable convolutional semantic sublayer. In Section 5.4 we shall present a non convolutional alternative, useful to solve some peculiar issues related to our One-Class Classification Architecture.

Linear Semantic Layer

The simplest convolutional semantic layer we can think of is a linear semantic layer, like the one shown in figure 5.3. in this kind of layer, for each point x_i we multiply together the spatial descriptors $\{s_{ik}\}$ (i.e. the values $s_k(\tilde{x}_i)$ computed for a given $\tilde{x}_i$), the features $\phi_j(x_i)$ and a set of weights associated to them w_{kjm}. After that, we sum the contribution of all the input points together. the result is the following formula:

$$\psi_m(y) = \sum_{i,j,k} \phi_j(x_i)\ w_{jmk}\ s_k(\tilde{x_i}). \tag{5.8}$$

In this case, the filter $g_{jm}(\cdot)$ is defined as:

$$g_{jm}(\cdot) = w_{jmk}\ s_k(\tilde{x_i}).$$

In our experiments, such very simple semantic layer proved to yield very good performances in terms of classification accuracy and AUC, while also being easy to implement and fast to compute.

MLP-Based semantic Layer

An alternative semantic layer can be obtained by approximating the function $f(\cdot)$ as defined in Section 5.2 with the help of a MLP. In this case, the convolution formula would be the following:

$$\psi_m(y) = \sum_{i,j,k} \phi_j(x_i)\ MLP(s_k(\tilde{x_i})) \tag{5.9}$$

Note that the Multi layer Perceptron receives in input only one spatial descriptor $s_k(\tilde{x_i})$ at a time. if the MLP was fed with the descriptors for all the neighborhood at the same time, the resulting operator would cease to be convolutional.

Using this kind of semantic layer yields two main drawbacks: the first one is that, MLP being more powerful than a singular linar layer, it is also more prone to overfitting. In order to avoid this problem, the Multi layer perceptron should be designed carefully; the second problem is about the fact that an MLP would increase the depth of the single composite convolutional layer. In particularly deep CNN architectures, it would be difficult to train such composite convolutional layers due to the so called "*vanishing gradient*" problem.

5.2.4 Composite Layer's properties and advantages

Our composite convolutional layer possesses some fundamental properties that are important to discuss. A first observation is about scalability: let us rewrite the convolution operator in (5.3) in the case of a RBF spatial layer and a linear semantic layer:

$$\psi_m(y) = \sum_{j,i,k,l} \phi_j(x_i)\ w_{mjk}\ \omega_{kl}\ h_l(\tilde{x}_i) \quad where \quad \tilde{x}_i = y - x_i, \tag{5.10}$$

where ϕ_j are input features, ψ_m are output features and x_i are input points. We can think of each semantic feature weight w_{jkm} as an element of a 3-dimensional tensor having shape $(C; K; M)$, where C is the number of input features, K is the number of dimensions of each spatial descriptor $s(\tilde{x}_i) = \sum_l \omega_{kl}\ h_l(\tilde{x}_i)$, and M is the number of output features; on the other hand, each spatial weight ω_{kl} is an element of a matrix with dimensions (M, L) where L is the number of RBF centers in the spatial sublayer. The number of semantic weights w_{jkm} is typically much higher than the number of spatial weights ω_{kl}. That's not only because the set of spatial weights is 2-dimensional and the set of semantic weights is 3-dimensional, but also because the number of incoming and outgoing features can be much higher than the number of RBF centers. An advantage of our model is that we can increment the complexity of our spatial sublayer, in this case by changing the number of RBF centers, without making the number of semantic weights explode. This is not possible, for example, in KpConv [51] where the weights are shared between the RBFN and the features: here incrementing the number of RBF centers by one unit means to add $(C \times M)$ new parameters to the convolutional layer, while in our case only K parameters are added. In other words, the complexity of the spatial layer has no impact in the semantic layer, given a constant K. This is not the case in Convpoint [4], where the dimensions of the spatial MLP was function of K. To Increase the MLP complexity, therefore, means also to increase the complexity of the spatial layer.

A second consideration is about flexibility. Depending on the problem, we can modify only one subcomponent of the composite convolutional layer, and leave the other unchanged. We can also think about implementing other, different spatial and semantic layers. As we discuss in section 5.4, also certain non convolutional combinations can yield useful results in certain environments.

5.3 Deep SVDD for OC classification

In this section we aim to describe a model called *Deep SVDD* [42] by Ruff et al. This model is meant to perform one-class classification by combining the use of Deep learning with the tecnique called *Support Vector Data Description*, presented in [49] by Tax and Duin. The idea is to train a function:

$$\mathcal{M}_W(\cdot)\ :\ I \rightarrow\ \mathbb{R}^\rho \tag{5.11}$$

that maps a certain input to a point lying in a ρ-dimensional space. more specifically, we want $\mathcal{M}_W(\cdot)$ to map all inputs in the training set inside a specific region of our space $\mathbb{R}^\rho$, minimising the volume of said region. The prediction is performed by checking if a certain test element is mapped inside the space region containing the training elements, or outside of it. In the first case, we predict the test element to be an instance of the normal class, in the second case we predict it to be anomalous. In the first subsection, we shall discuss more in depth how to structure the problem here hinted; In the second subsection, we discuss about wich training loss function can be used to train our network; In the third subsection, we discuss about the architectural constraints that are needed to be taken into account when defining a Deep SVDD network; in the fourth and last section we shall discuss about the advantages of Deep SVDD with respect to other existing OC classification methods.

5.3.1 Problem setting

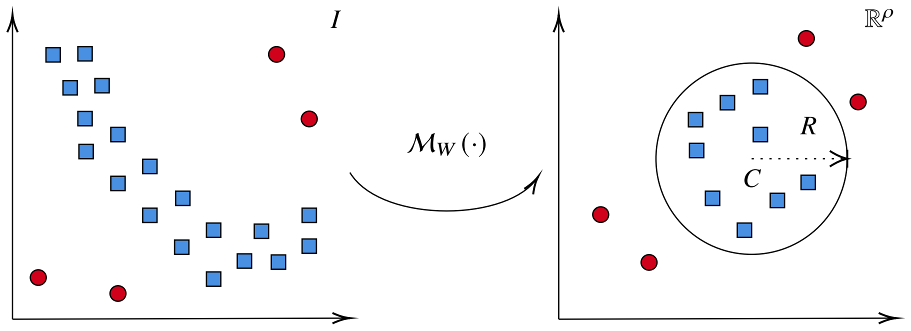

Figure 5.4: the image exemplifies the idea of deep SVDD. Represented in the left plot are a number of test elements: inliers are represented as blue squares, whether outliers are represented as red circles. the learned function $\mathcal{M}_W(\cdot)$ maps said test elements inside $\mathbb{R}^\rho$. all inliers should fall inside the topological disc with center C and radius R, whether anomalies should be mapped outside of said disc. It is possible to assign an "anomaly score" to each input element, depending on how much that element is far from the center: elements closer to the boundary are "more anomalous" than elements closer to the center C.

Deep SVDD is thought to be a method that is agnostic with respect on the kind of input data. Different inputs can easily be handled by changing the way in which the function $\mathcal{M}_W(\cdot)$ defined in expression 5.11 is learned. In order to be as much general as possible, said function is learned with the help of a Deep Neural Network. Every deep neural network that satisfies the constraints defined in section 5.3.3 can be used. The solution described in [42], for example, makes use of Convolutional neural networks to perform OC classification on images.

As described in this section's introduction, Deep SVDD works by mapping inputs to points lying in a certain region of an high dimensional space $\mathbb{R}^{\rho}$. More precisely, said region is defined as a topological disk, with a given center C and a given radius R. The aim is to map all training examples inside the hypersphere, minimising the hypersphere's radius as well. In formulae, the objective function describing this problem becomes:

$$\begin{aligned} \min_{R,C,\xi} \quad & \sum_{R,C,\xi} R^2 + \frac{1}{n\nu} \sum_i \xi_i \\ s.t.: \quad & || \mathcal{M}_W(x_i) - C || \leq R + \xi_i \quad \forall x_i \\ & 0 \leq \xi_i \quad \forall \xi_i \end{aligned} \tag{5.12}$$

where x_i is the training example, ξ_i is a slack variable associated to it penalizing examples falling outside the topological disk, and ν is a scaling factor that weights the importance of the slack variables in minimizing the objective function.

Note that this kind of objective function is very similar to the one of OC - Support Vector Machines as presented in [45]. This because Support Vector Data Description and Support Vector Machines are very similar methods, possibly equivalent depending on the problem setting [42].

5.3.2 Learning the map function

as already anticipated in the previous section, Ruff et al. proposed to use Deep neural networks to approximate the map $\mathcal{M}_W(\cdot)$. in this context, the training is performed via gradient descent on the radius R and the network's weights W.For this reason we define a fist loss function, called *soft-boundary Deep SVDD loss*, as:

$$\begin{aligned} \mathcal{L}(R,W) = \quad & R^2 + \\ & \frac{1}{n\nu} \sum_i \max\{ 0, ||\mathcal{M}_W(x_i) - C||^2 - R^2 \} + \\ & \frac{\lambda}{2} \sum_l^L ||W^l||_F^2 \end{aligned} \tag{5.13}$$

here the first term is used to minimize the radius; the second term penalizes every training example from being mapped outside the hypersphere; the third term is a regularization term for the network's parameters W, and λ is an hyperparameter. Note that with $||\cdot||_F$ we indicate the Frobenius norm. As we see, minimizing the radius is in direct tradeoff with the penalization assigned to each point falling outside the sphere. Reducing the radius means to increase the term $||\mathcal{M}_W(x_i) - C||^2 - R^2$.

Another simpler loss function can be used, assuming that training data are all inliers with respect to the real normal class data distribution. In this case, we rewrite the loss function as:

$$\mathcal{L}(W) = \frac{1}{n\nu}\sum_i ||\mathcal{M}_W(x_i) - C||^2 \quad + \quad \frac{\lambda}{2}\sum_l^L ||W^l||_F^2 \tag{5.14}$$

By following the notation in [42], we call such function *one class Deep SVDD loss*. In this case the Radius term is defined implicitly as the mean of the distances between the points and the center, and no point is penalized because it falls outside the hypersphere.

Independently from which loss we use, the metric used to discriminate anomalous samples from normal ones during testing is the distance $||\mathcal{M}_W(x_i) - C||$. The idea is, in fact, that normal samples are mapped closer to the center than anomalous ones. Note also that our decision boundary discriminating whether a certain sample is anomalous or not can be different from the optimal training radius R^* learned during the training phase, and needs to be tuned separately.

when trying to optimize the loss described in (5.13) via gradient descent, it is particularly important to consider the fact that the parameter R and the network's weights W are substantially different in terms of scale. Ruff et al [42]. suggest to alternate the optimization of the weights W and the radius R by gradient descent in separate phases.

5.3.3 Network requirements

In the previous two sections we explained that the map function $\mathcal{M}_W(\cdot)$ is learned by training a deep neural network. In order to avoid learning trivial and uninformative map function though, a certain number of conditions has to be met. here we intuitively discuss such conditions, that are more formally stated in the paper by Ruff et al. [42]:

1. *it needs to be assured that* $C \neq 0 \in \mathbb{R}^p$. If $C = 0$, then a perfect mapping would be $\mathcal{M}_W(x_i) = 0 \quad \forall x_i$, and this solution is easily obtainable by setting our network's weight $W = 0$. Note that the loss 5.13 for this mapping would be $\mathcal{L}(W) = 0$, which is optimum since all loss terms are positive; on the other way,

this mapping $\mathcal{M}_W(x_i)$ would map also anomalies to zero, thus being completely uninformative.

2. *no bias is present in the network approximating $\mathcal{M}_W(x_i)$.* If such bias term is present, it would be possible to learn any constant function $\mathcal{M}_W(x_i) = C \quad \forall x_i$ by setting to zero all non-bias parameters, such that the input is discarded, and combining the bias term(s) in a way such $M_W(x_i) = C$.As in the previous case where $C = 0$, the loss 5.13 would be $\mathcal{L}(W) = 0$, which is optimal. Again, this solution would map also anomalies to the center of our hypersphere, thus being completely uninformative.

3. *no activation function saturating to a value $\neq 0$ is present in the network approximating $\mathcal{M}_W(x_i)$.* A network unit with bounded activation function can be saturated for all inputs having at least one feature with common sign, thereby emulating a bias term in the subsequent layer.

As we understand from these properties, not all deep neural networks can be used in Deep SVDD. As we shall see in the next chapter, these conditions complicate how to implement a point cloud based OC classificator using deep SVDD, requiring substantial modification of the existing point cloud algorithms.

5.3.4 Advantages

Even being the constraints discussed in the previous section potentially difficult to met, the Deep SVDD has a key advantage with respect to other OC classification methods: as long as the said constraints are fulfilled, it is possible to freely define a Neural Network working on any kind of data employing Deep SVDD. The use of Deep neural networks is also useful to perform OC classification with complex or high dimensional data, in which traditional models such as OC-SVMs often fail. Moreover, Ruff et al. proved that their method is competitive in the field of OC classification on images, by comparing its results with other state of the art methods on well known datasets such as MNIST [27] and CIFAR10 [25]. Being so flexible, Deep SVDD was the natural choice when trying to perform OC classification on point clouds without having to define a completely new deep architecture.

5.4 Composite convolution in Deep SVDD

When presenting the Deep SVDD model in section 5.3, we pointed out that the main feature of this method is precisely its flexibility: it can be adapted to many different input data and different deep architectures. In this section we investigate how to use Deep SVDD for OC classification with point clouds. As we pointed out in section 3.5, there are no other examples of convolutional layers used in One Class classification architectures.

Using Deep SVDD with point clouds means to define a deep neural network capable of learning the map function defined in (5.11):

$$\mathcal{M}_W(\cdot) \; : \; I \rightarrow \; \mathbb{R}^p$$

Possible violations of the constraints described in 5.3.3 can be present both in the CNN architecture and in the convolutional layer itself. In this section, we aim to discuss only about the convolutional layer, as it is the core topic of this thesis. We shall now assume that a suitable CNN architecture is employed. An example of such architecture will be provided in chapter 6.

In the next subsection we discuss of each constraint: our goal is to explore whether a composite convolutional layer as defined in this chapter can fulfill all constraints, and what semantic and spatial sublayers can be used in this context.

5.4.1 Properties of Deep SVDD

As stated in the previous paragraph, our composite convolutional layer will have to comply with the following constraints:

1. *It needs to be assured that$C \neq 0 \in \mathbb{R}^p$.*
2. *No bias is present in the network approximating $\mathcal{M}_W(x_i)$.*
3. *No activation function saturating to a value ≠ 0 is present in the network approximating $\mathcal{M}_W(x_i)$.*

We can easily tell that constraint 1 is related to the hyperparameter C, which is unrelated to the convolutional Layer.

Constraint 3, on the other hand, deals with activation functions and non-linearities present in the network. Depending on the definition of our spatial and semantic sublayers, we can have different non-linearities inside our Composite convolutional layer. For example, we can use a MLP-based spatial sublayer, or use an RBFN-based spatial sublayer with a non-linearity between spatial and semantic layer. A way to ensure that our layer is compliant with constraint 3 is to use only functions that do not saturate to any value different from zero. An example of such activation function is rectified linear unit (ReLu), or its variations such as leaky-ReLu. Other activation functions, such as $tanh(\cdot)$ or sigmoid activation functions, are not suitable to be employed in Deep SVDD.

The most interesting constraint is the second one: here we state that any bias term should be removed, from both network and convolutional layers. While it is possible to remove all the explicit biases from the network and the layers, the problem is far to be solved: it is possible to have biases that are directly introduced by the input. If a certain element is constant for each point cloud and each neighbourhood, we obtain the same result as when having an explicit bias defined inside the layer.

We discussed in section 5.2.1 that we are selecting the neighbourhood by taking the k-nearest neighbours to the output point; We also stated in section 5.1.1 that we expect each output point to be part of the input point cloud. Those two conditions mean that the output point y is also the closest input point x_0 to y. Being $\tilde{x}_i = x_i - y$ the filter's argument, we expect to have the filter evaluated for $\tilde{x}_0 = x_0 - y = 0$ for each neighbourhood.

This would not be a problem, for example, when dealing with images. In image convolution, we will always evaluate the filter for a fixed set of arguments. The difference between the images and the point clouds is that in case of images information is also carried by the feature values of each pixel (e.g., representing RGB channels, brightness etc.) other than their spatial distribution; on the contrary, many point cloud datasets neglect to add features to the point sets: this is typical of datasets sampled from real world scenes or object, for example [11]. In this case the information is contained in the spatial distribution of the points, and features are only used by neural networks to encode the information derived from such spatial distribution.

If the dataset does not contain features, we consider each point to be associated with a unitary scalar feature when entering the network. Then, the first convolutional layer would produce several output features associated with the point cloud and pass it to the deeper layers, thus solving the problem. In such first layer, having an unitary feature associated with each point reduces the convolution operation to the following:

$$\psi_m(y) = \sum_{i=0}^{|X|} g_m(\tilde{x}_i) \ = \ g_m(0) \ + \ \sum_{i>0}^{|X|} g_m(\tilde{x}_i) \quad where \quad \tilde{x}_i = x_i - y.$$

In such equation, the term $g(0)$ is precisely the bias we want to avoid in order to implement a Deep SVDD architecture.

We have several ways to tackle this issue:

- use only datasets with features associated to each point, or preemptively assign non-constant features to each point cloud.
- use a composite convolutional layer that does not take into consideration the central point of the neighbourhood. An example is a composite convolutional layer with a MLP-based, unbiased spatial sublayer. In such case the contribution of the central point of the neighbourhood would be zero. The main drawback of this solution is that we lose the information associated with the central point, which is considerably more important as we proceed deeper in the network.
- select output points so that they are not part of the input point cloud. Doing this, we avoid having any $\tilde{x}_i = 0$. This solution, however, would be the less efficient: having the output points inside the input point cloud allows us to precompute all distances between output and input points, thus saving a considerable amount of run time.
- perturbate the central point of the neighbourhood so that its associated difference vector $\tilde{x}_0$ is different from 0. Such solution introduces noise in the neighbourhood and thus can decrease performance.
- define a composite, non-convolutional layer that aggregates spatial information from the neighbourhood before combining them with the features. In this way, to exploit the bias term related to $g(0)$ becomes much less trivial.

Of these four solutions, we are more interested in exploring the last one: redefining a different, non convolutional layer will prove the flexibility of our composite approach.

5.4.2 An alternative semantic layer

In order to cope with the bias problem highlighted in section 5.4.1, we decided to define a different composite layer, that does not employ convolution but instead aggregates neighbourhood information in a different way. For us, this is also a way to prove that convolutionality is not a mandatory property of our composite layer.

The main component of our non-convolutional composite layer is a new semantic sublayer, called *aggregate* semantic sublayer. Thought to be used in combination with any other RBFN-based spatial sublayer, the idea behind the aggregate semantic sublayer is to make the bias more difficult to exploit, rather than eliminating it completely.

Receiving from the spatial layer a set of descriptors $\{s_k(\tilde{x}_i)\}$ for each input $\tilde{x}_i$, the idea is to aggregate them before combining them with the features. We propose to aggregate such set of spatial descriptors by computing their first N central moments with respect to the points in the neighbourhood. Here we indicate such moments with $\mu_1(\boldsymbol{s}), \mu_2(\boldsymbol{s}), ..., \mu_N(\boldsymbol{s})$, where s is the vector $\boldsymbol{s} = (s_1(\cdot), s_2(\cdot), ..., s_k(\cdot))$. Since the aggregation is performed over points, we expect each $\mu_n(\boldsymbol{s})$ to have k components. After computing the first N central moments, we concatenate them in a single vector:

$$\boldsymbol{\sigma} = (\mu_1(\boldsymbol{s}), \mu_2(\boldsymbol{s}), ..., \mu_N(\boldsymbol{s})),$$

having dimension kn. The same idea is applied on features, so that we will have the same N central moments computed point-wise and concatenated in a single vector:

$$\boldsymbol{\theta} = (\mu_1(\phi), \mu_2(\phi), ..., \mu_N(\phi)),$$

having dimension nC. The convolution is then performed by multiplying the spatial information contained in vector σ by a matrix of weights W, and then by the feature moments vector θ. In case of having a single output feature, the convolution operator would look like:

$$\psi_m(y) = \sum_{j=0}^{nC} \sum_{i=0}^{nk} \theta_j w_{ji} \sigma_i. \tag{5.15}$$

This solution is not intended to solve the bias problem completely: it will be still theoretically possible, for the network, to reach certain configurations that produce biases. For example, the spatial layer could learn a function $s(\tilde{x}_i)$ similar to:

$$\begin{cases} s(0) = 1 \\ s(\tilde{x}_i) = 0 \quad \forall \tilde{x}_i \neq 0 \end{cases} . \tag{5.16}$$

On the other hand, such function is really difficult to approximate by using a RBFN-based spatial layer like the ones we proposed, especially when the number of centers is limited. That's because the RBFN output is a continuous function, being a linear combination of continuous RBFs [23] [60], while (5.16) is not. Knowing that the RBFN alone is not fully able to produce an output like (5.16), we avoid that the semantic layer can participate in defining a bias-exploiting convolutional filter.

This aggregate layer is clearly different from a convolution, since both features and filter are condensed together by the use of central moments. Our experiments show that our semantic layer can be used to address OC classification in point clouds, while also being able to well perform in multiclass classification tasks.

Chapter 6

Benchmarking and Experiments

In chapter 5 we introduced a novel layer, called Composite Layer to be used in semi-supervised Classification and unsupervised OC classification tasks. This layer was designed to be architecture agnostic, as many other State of the Art alternatives like KpConv [51], PCNNEO [2] and ConvPoint [4]. In this chapter, we demonstrate our solution's capabilities by performing experiments both in the field of supervised Classification and unsupervised One-Class Classification. In Section 6.1 we present some datasets that are considered standard benchmarks in 3-d deep learning. In particular, we focus on *ModelNet40* and *ShapeNetCore*, datasets that are composed of 3d meshes, from which it is possible to extract point clouds or other types of geometric data. In the second subsection, we propose a possible CNN architecture that employs our composite convolutional layer; after describing such architecture, we test our model and compare its performance with other State of the Art methods. We follow a similar approach in Section 6.3, where we discuss OC Classification. Since no other OC classification methods from the literature operate on point clouds, we are more interested in proving that it is possible to perform OC classification on Point clouds rather than comparing our model with existing alternatives.

6.1 Datasets for 3D Deep Learning

Unlike in other fields, the datasets used in 3D deep learning can be quite diverse. As we discussed in chapter 3, The reason behind this is that 3D deep learning is a vast field including many alternative methods. For instance, graph-based methods need datasets containing graphs; many methods employing multiview CNNs are better off when trained on images acquired from meshes; other methods, like ours, work on point clouds; several models from this last category, like PointNet, do not require features associated to the points, while other methods are better off with additional features. For this reason, it is crucial to choose a meaningful dataset when testing a 3D deep learning model. Focusing on Point Clouds, we present *ModelNet40* [55] and *ShapeNet* [7], two widely-used datasets in 3D shape classification. We also cite other relevant datasets, particularly regarding point clouds, that can be considered when developing more application-specific solutions.

Class	**#Instances**	**Class**	**#Instances**	**Class**	**#Instances**	**Class**	**#Instances**
airplane	726	cup	99	laptop	169	sofa	780
bathtub	156	curtain	158	mantel	384	stairs	144
bed	615	desk	286	monitor	565	stool	110
bench	193	door	129	night_stand	286	table	492
bookshelf	672	dresser	286	person	108	tent	183
bottle	435	flower_pot	169	piano	331	toilet	444
bowl	84	glass_box	271	plant	340	tv_stand	367
car	297	guitar	255	radio	124	vase	575
chair	989	keyboard	165	range_hood	215	wardrobe	107
cone	187	lamp	144	sink	148	xbox	128

Table 6.1: Modelnet40: classes and instance distribution

6.1.1 ModelNet40

ModelNet40 [55] is probably the most widely used benchmarking dataset in the field of 3D deep learning. Two factors contributed to its success: first, it is composed of meshes from many everyday use categories. Since it is possible to extract different types of data from a mesh, such as point clouds and graphs, Modelnet40 is readily employable by most 3D deep learning algorithms. Secondly, a significant advantage is that ModelNet40 does not contain real-world objects: it is synthetic, which means that it contains only fictitious meshes. Thanks to this, the dataset is free from the intra-shape noise.

As the name suggests, this dataset is composed of 40 classes containing meshes of commonly used objects, for a total of 12311 shapes. We can see the list of these

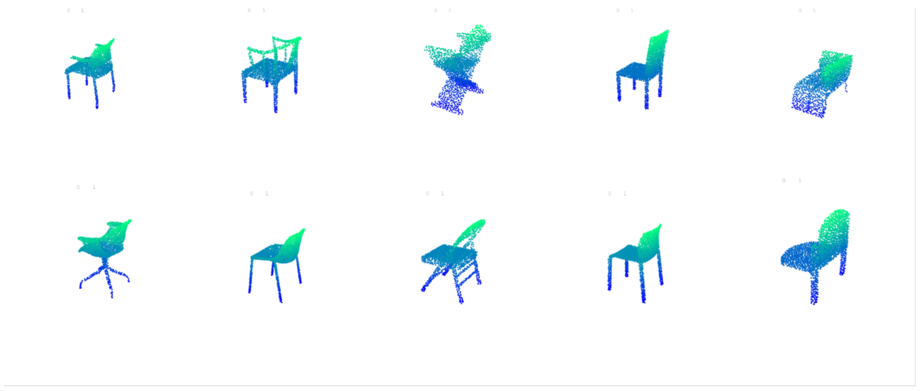

Figure 6.1: Some instances extracted from the class “chair” of Modelnet40. In particular, we are showing point clouds sampled from the mesh contained in said dataset, as explained in section 6.2.3.

classes, alongside their cardinality, in table 6.1. The first thing we can notice is that these classes are not perfectly balanced in terms of instances. For example, the class *cup* is roughly 90% smaller than the class *chair*. It is possible to measure the class imbalanced by the standard deviation of the class cardinality distribution:

$$\sigma = \sqrt{\frac{1}{N}\sum_i \left(c_i - \frac{1}{N}\sum_j (c_j)\right)^2} = 215.5446,$$

Where N indicates the total number of instances in the dataset and c_i the number of instances of class i.It is interesting to notice that Modelnet40, even being the most widely used benchmarking dataset in the field, is fairly unbalanced in terms of class instances. On the contrary, the benchmark of image classification datasets such as CIFAR-10 [25] are perfectly balanced.

Modelnet40 also comes with an official train-test split, which is suggested to correctly compare performances with other methods. This official split takes 9843 shapes (the 80% of the entire dataset) for training and leaves the remaining 20% for testing. Since some classes (like *cup*, *bowl* or *stool*) are relatively small compared to the others, it is difficult to produce a split that includes a validation set as well: it would mean either to produce a validation set with too few instances on the small classes, or to significantly reduce the impact of those classes in the training set.

Class	#Instances	Class	#Instances	Class	#Instances	Class	#Instances
airplane	2832	cap	4612	lamp	113	rifle	420
bag	58	car	455	laptop	380	rocket	115
basket	77	cellphone	65	mailbox	296	skateboard	149
bathtub	599	chair	762	microphone	1620	sofa	1656
bed	167	clock	5863	microwave	319	speaker	59
bench	1260	dishwasher	323	monitor	1116	stove	106
birdhouse	340	earphone	5863	motorcycle	65	table	2198
bookshelf	125	faucet	84	mug	46	telephone	152
bottle	628	file	272	piano	107	tin_can	1356
bowl	1076	guitar	45	pillow	235	tower	116
bus	227	helmet	51	pistol	149	train	404
cabinet	79	jar	519	pot	167	vessel	51
can	39	keyboard	208	printer	67	washer	310
camera	2458	knife	557	remote_control	185	–	–

Table 6.2: Shapenet Core: classes and instance distribution

6.1.2 ShapeNet

Another well-known synthetic dataset for 3D deep learning is *Shapenet* [7]. This dataset is closely related to Wordnet [32], a lexical database based on the English language that proposes to organize, describe and define concepts expressed by words. In Wordnet, an important conceptual unit is the one of *synset*: a set of words that can be used to describe a single concept, object, or idea. Shapenet's goal is precisely to associate to a vast number of commonly used synsets a set of 3D shapes representing the concepts behind them. Concretely, there are a number of different "flavors" in which Shapenet is distributed, depending on both the kind of 3D shapes and the quality of annotation. While certain flavors like *ShapenetSem* [44] are widely used in the field of 3D shape segmentation, In the field of 3D shape classification, we are more interested in discussing a subset of ShapeNet called *ShapenetCore*. This ShapeNet subset comes with single clean 3D models with manually verified category and alignment annotations, and covers 55 common object categories with about 51,300 unique 3D shapes.

Differently from Modelnet40, this dataset is distributed with an official 3-way split, defining a training set, a validation set and a separate test set. We can see the classes distribution inside the training set in table 6.2. Here it is possible to notice how much this dataset is unbalanced towards some classes. For example, the class "guitar" only contains 45 instances, while the class "clock" contains 5863 instances; this makes the guitar's class cardinality less than 1% of the clock's class cardinality.

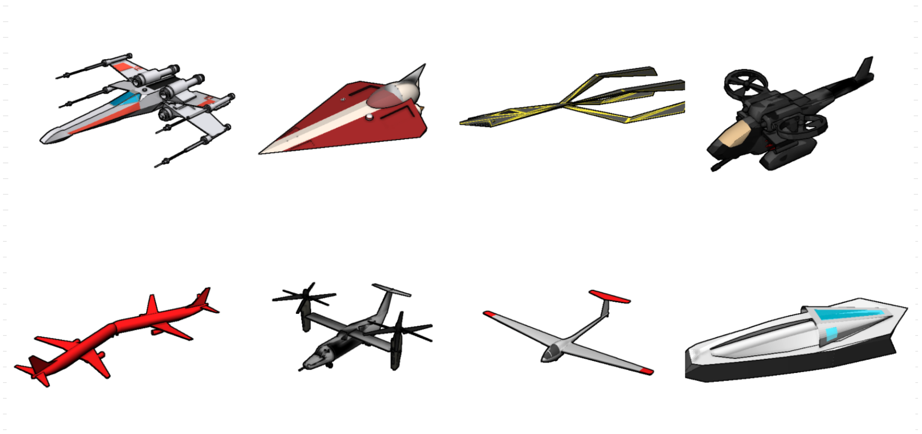

Figure 6.2: Some shapes extracted from the class “airplanes” of ShapenetCore. As can be observed, this definiton of “airplane” also includes spaceships from videogames or other media products, helicopters, drones and even a shape representing the collision between two airplanes in the 1990 Wayne County Airport accident [1].

We can quantify the skewness of ShapenetCore by calculating the standard deviation over the number of instances c_i for each class:

$$\sigma = \sqrt{\frac{1}{N}\sum_i \left(c_i - \frac{1}{N}\sum_j (c_j)\right)^2} = 1096.7402,$$

Where N indicates the total number of instances in the dataset and c_i the number of instances of class i. As we can see, the standard deviation for ShapenetCore is much higher than for Modelnet40. Another important feature of this dataset is the great intraclass variability, in particular when dealing with certain classes with high number of istances. We can see an example of this in figure 6.2.

These two factors contribute in making ShapenetCore a difficult dataset to train a deep learning model. Some works try to overcome this difficulty by selecting a specific subset of shapenet, like the case of [59]. Many other methods rely primarily on Modelnet40 when testing their performances in 3d shape classification, whithout referring to ShapenetCore.

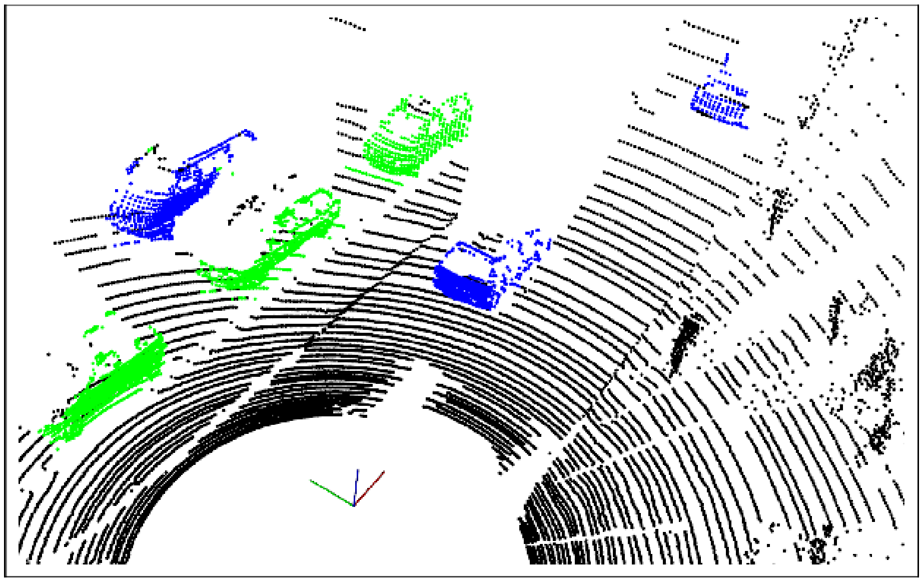

Figure 6.3: Image from [12] representing the result of a semantic segmentation experiment over a scene represented by a 3D point cloud extracted via LiDAR. It is possible to see how the elements closer to the observer, here at the origin of the reference frame, occlude objects behind them.

6.1.3 Other relevant datasets

The common feature of both ModelNet and Shapenet is that they are synthetic mesh datasets. On the other hand, there are situations in which a model should be evaluated on a dataset obtained from real-world objects. This is particularly true in the case of point clouds, where several techniques exist to sample a point cloud from a real-world object. For example, *Terrestrial Laser Scanning* (TLS) techniques are used in topography [35] to produce representations of the landscape or its features. The main difference between real-world and synthetic datasets is that Synthetic datasets have complete and noiseless elements, while in real-world datasets the elements can be noisy or partially occluded [19]. In Figure 6.3 we report an example of an occluded 3D shape. In the field of 3D shape classification, two interesting real-world datasets are the *Sidney Urban Object Dataset* [11] (SUOD) and *Scan Object NN* (SONN) [52].SUOD is one of the first 3D point cloud datasets built from real-world samplings, but having less than 600 instances it is also much smaller than the alternatives. On the other hand, SONN is a recent real-world dataset (published in 2019) with size comparable to the already discussed ModelNet40. Unfortunately, the authors have not released the dataset to the public yet.

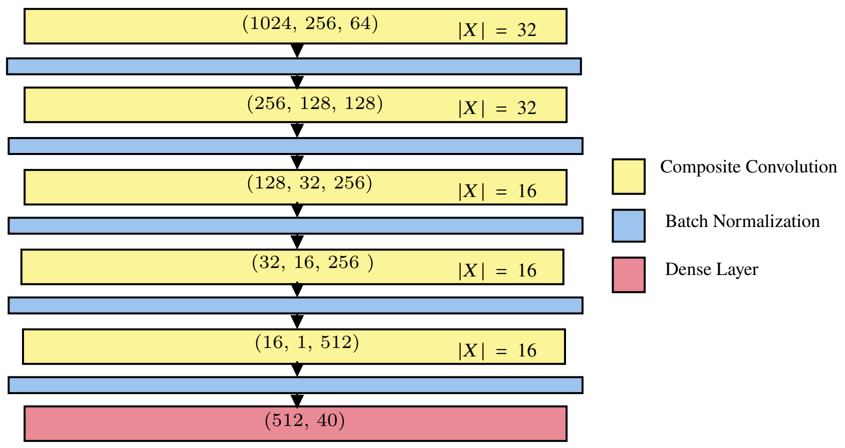

Figure 6.4: The chosen neural network architecture, very similar to the one employed in [4] by Convpoint. Each convolutional layer is labeled with a 3-uple $(|P_I|, |P_O|, \gamma)$ where $|P_I|$ is the number of input points, $|P_O|$ is the number of output points, γ is the number of output features produced by the layer. On the other hand, we label $|X|$ the number of neighbouring points considered in each layer.

6.2 3D Shape Classification

To evaluate our novel point-convolutional layer's performances, we conducted some 3D shape classification experiments. We chose to employ two synthetic datasets: Modelnet40 and ShapenetCore, both described in Section 6.1. Firsts, we describe the Neural Network architecture in which we employ our convolutional layer and present some comparable State of the Art models that we use as baselines in our experiments. As already stated in section 6.1, both ModelNet40 and ShapenetCore are composed of 3D meshes. Since our model operates on point clouds, we dedicate a paragraph to describe how we construct such point clouds starting from the 3D shapes contained in the over stated Datasets; moreover, we briefly describe some simple data augmentation techniques we employed during training and testing. After presenting the appropriate performance metrics, we discuss the experiments' nature and present their results.

6.2.1 Neural Network Architecture

Our layer needs to be deployed in a suitable Neural Network Architecture to be appropriately tested. Inspired by Convpoint [4], we decided to define a quite simple NN Architecture, composed of 5 strided convolutional layers followed by a final dense layer. We can see the details of such architecture in figure 6.4. The rationale behind this architecture is simple: since we need to test the capabilities of the sole composite layer, any functional architecture can be employed; moreover, the architecture should be simple enough to be used in both Modelnet40 and ShapenetCore without substantial modifications. In this sense, the NN proposed by Convpoint is fitting. Indeed, it is both simple, functional and has already shown to perform well in 3D shape classification.

6.2.2 Baselines and comparable Methods

To evaluate our solution's performance, we train three different State of the Art methods and compare their results with the ones obtained using our Composite Layer. These methods are:

- **Pointnet** [37]. A non-convolutional method employing a custom neural network architecture, representing one of the earliest and most known approaches to the field of deep learning with point clouds. Since the power of Pointnet resides in its peculiar architecture, this model will be the only one not using the convolutional architecture described in section 6.2.1.
- **ConvPoint** [4]. A convolutional method presented in 2020 and sharing some similarities with our proposed solution. being it an architecture-agnostic convolutional layer, we shall test inside the same architecture previously defined in section 6.2.1.
- **KpConv** [51]. Until very recently, this convolutional method was the one obtaining the best results in the field of 3d shape classification in terms of Overall Accuracy (OA). In this case, we refer to *KpConv-rigid*, a variant of KpConv that does not employ deformable convolutions. Like in the case of ConvPoint [4], we shall test KPConv inside the NN architecture we defined in section 6.2.1.

Note that the results of our experiments with these methods are different from the ones presented in the respective papers. This primarily because of the different NN Architecture. None of these methods was reimplemented, as the experiments were run on the implementations suggested by the respective authors.

6.2.3 Data preprocessing

All the algorithms we test operate on the point clouds, but the shapes included inside the Modelnet40 and ShapeNetCore datasets are shipped in the form of 3D meshes. In this sense, a fundamental operation in our workflow is to convert these 3D meshes to point clouds. A common way to address this task is to sample the point cloud from the mesh by using the so-called *farthest point sampling* [34] strategy. Broadly speaking, the process of sampling T points from a given mesh is the following:

1. Construct a point set V comprising all polygon vertices inside the mesh.
2. Sample a point set P_{rand} from the mesh surface, in such a way that each polygon contains a number of sampled points proportional to its area.
3. Initialize the point cloud P as the set containing one random point $p_0 \in V \cup P_{rand}$.
4. Add to the point cloud P the farthest point $p_f \in V \cup P_{rand}$ from all the elements already in P.
5. Repeat step 4 until $|P| = T$.

It is possible to perform this process before training, obtaining point-cloud-based variants of ModelNet40 and ShapeNetCore. Several already sampled versions of these two datasets are available online, an example being [48].

Instead, an operation that we perform online (i.e. during training) is data augmentation. In our case, we decided to augment our data by performing the following operations over each point cloud:

- Random translation along all three spatial axes x, y, z.
- Random rotation from 0 to 360 degrees around the vertical axis z.
- Random rotation from -30 to $+30$ degrees around the x and y axes.
- Mirroring with respect to the vertical axis z.

To be compliant with other works dealing with 3D shape classification, we employ the standard training/test splits from ModelNet40 and ShapeNetCore. In this way, our results are comparable with the ones presented in literature.

6.2.4 Evaluation metrics

The last aspect we discuss before presenting the experiments is the set of performance metrics we are interested in measuring. In our 3D shape classification task, we decided to employ three different figures of merit. The first two are commonly employed by other State of the Art methods. In contrast, the third one is less common though beneficial in imbalanced datasets such as the ones we are considering. We now present some definitions that are useful in describing such performance metrics. Given a certain test set and a multiclass classifier, we can define four sets of instances for each class:

- **set of true positives for class *i***: the set comprising the correctly classified instances of class i . We indicate its cardinality with TP_i.
- **set of true negatives for class *i*** : the set of instances not belonging to class i that are not classified as examples of class i. We indicate its cardinality with TN_i.
- **set of false positives for class *i*** : the set of instances incorrectly classified as examples of class i. We indicate its cardinality with FP_i.
- **set of false negatives for class *i*** : the set of instances belonging to class i correctly recognized. We indicate its cardinality with FN_i.

Since the entire dataset is composed by the union of all classes, the total number of instances in the test dataset is $N = \sum_i^c (TP_i + FN_i)$, where c is the number of classes in the dataset. It is possible to use the sets introduced before to define several metrics, like the one we are using.

Overall Accuracy (OA)

OA is the most common figure of merit. It is simply defined as the number of correctly classified shapes over the total number of classified shapes:

$$OA = \frac{\sum_i^c TP_i}{\sum_i^c (TP_i + FN_i)}.$$

Intuitively, it represents the probability that the evaluated algorithm will correctly classify a given test instance. A significant drawback arises of imbalanced datasets. In this case, OA cannot capture whether the algorithm performs worse in smaller classes.

Average Accuracy (AA)

AA is a useful metric in imbalanced datasets, obtained by averaging the per-class accuracy. While it is possible to weight each class differently, we use the same weight for each class. In formulae:

$$AA = \frac{1}{c}\sum_{i}^{c} \frac{TP_i + TN_i}{TP_i + FP_i + TN_i + FN_i}.$$

This metric captures whether the classifier performs worse in specific classes, as all of them are equally weighted independently from their cardinality. It is interesting to confront OA and AA: the first one is able to capture overall performances, while the second one tells us if the model is able to well perform on every class.

Average 1vs1 AUC (AAUC)

In binary classification, performances also depend on the threshold we use to discriminate between the two classes in terms of algorithm score. In this context, a useful tool to evaluate the classification performance independently from said threshold is the so-called Receiver Operating Characteristic Curve. Such curve is created by plotting the true-positive rate (TPR) against the false-positive rate (FPR) at various threshold settings. This plot illustrates a binary classifier's diagnostic ability as its discrimination threshold is varied. In particular, the Area Under the ROC Curve (AUC) is a metric that expresses how well the two classes are separated, with an $AUC = 1$ meaning a perfect classifier and an $AUC = 0.5$ meaning a random classifier. In 2001, Hand and Till[20] proposed a generalization of the AUC for multiclass settings.

Let us suppose that the classes are labeled as $1, 2, ..., c-1, c$. Given a pair of classes (i, j), A good classifier should assign a high probability to the correct class, while assigning low probabilities to the other classes. This can be formalized in the following way: Let $(i|j)$ be the probability that a random instance of class j is recognized as belonging to class i with an higher probability of a random instance of i. Let also $(j|i)$ be defined accordingly. In this context, we can consider the metric $(i, j) = \frac{1}{2}((j|i) + (i|j))$ as a measure for the separability for classes i and j. In this sense, the Average 1vs1 Area under the ROC curve (AAUC) is defined as average of all possible (i, j). in formulae:

$$AAUC = \frac{2}{c(c-1)}\sum_{i<j}(i, j).$$

6.2.5 Experiments and results

Since our composite layer can be implemented as a combination of different semantic and spatial layers, there are multiple ways of defining it. For this reason, we perform different experiments to investigate the impact of certain sublayers or certain parameters specific to our novel composite layer in terms of performance. After this ablation study, we compare the overall performance of our solution with the State of the Art methods introduced in section 6.2.2. For these experiments we use the already cited ModelNet40 and ShapeNetCore datasets, subsampled so that each point cloud is composed by 1024 points. Note also that each point is not associated to any features when it enters the network, the only information that it carries being its spatial position.

Experiment 1: robustness to spatial descriptor dimension variation

First, we are interested in investigating the way performances vary with different dimensions for each spatial descriptor $K = |\{s_k(x_i)\}|$. In this experiment, we consider a composition of a RBFN spatial layer and a Linear semantic Layer. We test such composition with different values of K, as shown in table 6.3

Name	Spatial L.	ReLu	Semantic L.	K	notes
L1.1	RBFN	yes	Linear	8	48 RBFN centers
L1.2	RBFN	yes	Linear	16	48 RBFN centers
L1.3	RBFN	yes	Linear	24	48 RBFN centers

Table 6.3: Layers tested in Experiment 1, about the robustness to variation of the spatial descriptors dimensionality.

In this context, each Radial Basis Function is Gaussian and fixed a-priori:

$$h(r) \;=\; \exp\left(\frac{r^2}{\sigma}\right) \quad where \quad \sigma = 0.08. \tag{6.1}$$

Such choice of the hyperparameter σ was performed by Hyperparameter Tuning. Similarly, we chose the starting learning rate $\lambda = 5 \cdot 10^{-4}$.

To measure the metrics described in section 6.2.4, we perform three different trainings on each model configuration described in table 6.3. Our measure is obtained as average of the result obtained by testing these trained models. For the sake of brevity, we perform these measures only on ModelNet40.

Model	OA	AA	AAUC	N. of Param.s	notes
L1.1	**0.8993**	**0.8668**	**0.9898**	1.92M	–
L1.2	0.8951	0.8515	0.9886	3.83M	–
L1.3	0.8936	0.8553	0.9884	5.73M	–

Table 6.4: Results of Experiment 1 over ModelNet40, about the robustness to variation of the spatial descriptors dimensionality.

As can be seen from table 6.4, there is little difference between the three proposed layers. The experiment strongly indicates that having more semantic parameters (i.e. parameters used in the semantic layers, which compose the vast majority of the network's parameters) does not correspond to better performances. said number of semantic parameters depends both on the number of output features (which is fixed in this experiment) and the number of spatial descriptors K for each point. In this case, the best model is the simplest one, with smallest K and thus with fewer seantic parameters. this experiment also shows that to change K may be potentially useful: this is surely an advantage with respect to models, such as ConvPoint, in which K is fixed.

Experiment 2: use of ReLu between spatial and semantic layers

In this experiment, we evaluate how ReLu activation between spatial and semantic layers impact classification performance. Again, we choose to perform the test over a combination of RBFN spatial layer and Linear Semantic layer.Table 6.5 summarizes the characteristics of the tested layers.

As in the previous experiment, we consider the use of fixed gaussian RBFs, with $\sigma = 0.08$ and starting learning rate $\lambda = 5 \cdot 10^{-4}$. $\lim_{r\to\infty} g(r) = 0$

As can be seen in table 6.6, layer L2.1 clearly outperforms layer L2.2. This experiment shows that, while it is possible to add non-linearities between spatial and semantic layer, this is not always useful and might worsen the performance.

Name	Spatial L.	ReLu	Semantic L.	K	notes
L2.1	RBFN	no	Linear	16	48 RBFN centers
L2.2	RBFN	yes	Linear	16	48 RBFN centers

Table 6.5: Layers tested in Experiment 2.

Model	OA	AA	AAUC	N. of Param.s	notes
L2.1	**0.9032**	**0.87724**	**0.9914**	3.83M	–
L2.2	0.8951	0.8515	0.9886	3.83M	–

Table 6.6: Results of Experiment 2 over ModelNet40, about the non-linearity between spatial and semantic layers.

Experiment 3: Training the Radial Basis Function

In section 5.2.2 we described a possible way to learn the shape of the Radial Basis Functions composing a RBFN. We are now interested in verifying the impact of such choice in terms of performances. To do this, we compare the results obtained by the layers summarized in table 6.7.

Name	Spatial L.	ReLu	Semantic L.	K	notes
L2.1	RBFN	no	Linear	16	48 RBFN centers
L3.1	RBFN w/ Fourier RBF	no	Linear	16	16 RBFN centers
L3.2	RBFN w/ Fourier RBF	yes	Linear	16	32 RBFN centers

Table 6.7: Layers tested in Experiment 3.

Note that layer L2.1 is the same used in Experiment 2, since we consider it a baseline to evaluate the performances of layers L3.1 and L3.2 .

As described in section 5.2.2, learning the shape of the RBF with Fourier expansions requires the choice of some additional parameters. In particular, we are referring to the base frequency f and the expansion length T. In both layers L3.1 and L3.2, we employ

$f = 0.5$ and $T = 10$. Again, for the sake of clarity we expose only the results obtained over ModelNet40.

Model	OA	AA	AAUC	N. of Param.s	notes
L2.1	**0.9032**	**0.87724**	**0.9914**	3.83M	–
L3.1	0.9014	0.8702	0.9902	3.83M	–
L3.2	0.9031	0.8702	0.9904	3.83M	–

Table 6.8: Results of Experiment 3 over ModelNet40, about the impact of learning the radial basis function in a RBFN.

The results of this experiment, reported in Table 6.8 tell us that layers L3.1 and L3.2 perform similarly to Layer L2.1, by using a RBFN with fewer centers as Spatial sublayer. This is probably due to the flexibility obtained by learning the RBF shape. Moreover, we saw in Experiment 2 that the ReLu between spatial and semantic sublayers damaged the performances of layer L2.2; In this case though, Layer L3.2 is able to obtain the same performances as the other two regardless the presence of the non-linearity.

Experiment 4: testing the aggregate semantic layer

Finally, we are interested in testing the aggregate semantic layer we proposed in section 5.4.2. We recall that this semantic layer aggregates spatial descriptors and features before combining them. For this reason, when using this semantic sublayer our composite layer is not convolutional. We reprise the layer's operator definition reported in (5.15):

$$\psi_m(y) = \sum_j^{nC} \sum_i^{nk} \theta_j w_{ji} \sigma_i .$$

In this context, θ_j and σ_i are obtained by concatenating the n central moments, computed respectively over the spatial descriptors and input features. In this experiment, we compute only the first 2 central moments, namely mean and variance. In this way, we do not dramatically increase number of parameters w_{ji}, while employing the same hyper-parameters used in previous experiments. We compare the performances of the aggregate semantic layer, with those of Layer L2.1 from experiment 2 and Layer L3.1 from experiment 3.

Name	Spatial L.	ReLu	Semantic L.	K	notes
L2.1	RBFN	no	Linear	16	48 RBFN centers
L3.1	RBFN w/ fourier RBF	no	Linear	16	16 RBFN centers
L4.1	RBFN w/ fourier RBF	no	Aggregate	16	16 RBFN centers

Table 6.9: Layers tested in Experiment 4, about the performances of the aggregate semantic layer.

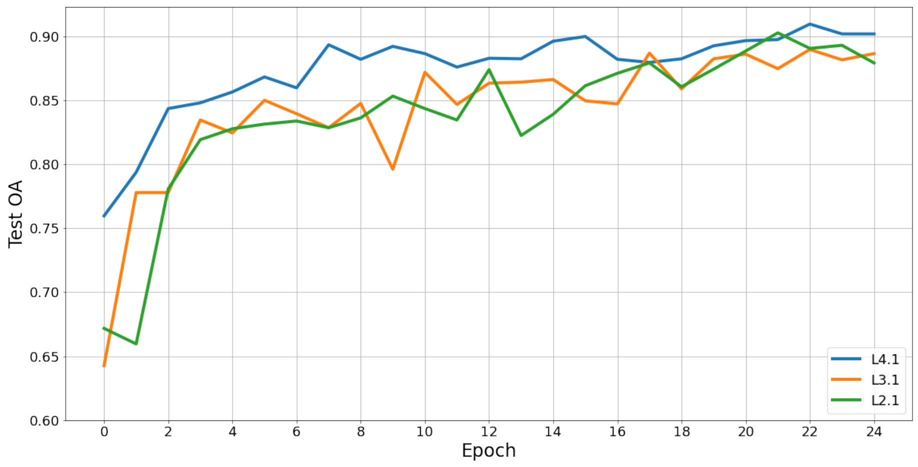

Figure 6.5: Overall Accuracy over the Test Set of ModelNet40, plotted per epoch of training.

In Table 6.10 we can see the results obtained by our layers over ModelNet40. An important aspect to note is that layer L4.1 has a larger number of parameters than the other methods. This is due to the fact that both the semantic features and the spatial descriptors are duplicated inside each layers: considering the first two central moments implies having a vector θ that has two times more components than $s(\cdot)$, and the same goes for σ and $\phi(\cdot)$. Interestingly, layer L4.1 outperforms both L3.1 and L2.1 in terms of Overall Accuracy and Average 1vs1 AUC; on the other hand, layer L2.1 carries a noticeable advantage compared to L4.1 in terms of Average Accurracy. Another important result of this experiment is that layer L4.1, even having four times more parameters, is able to converge to these results faster than L3.1 and L2.1. This later aspect can be seen in Figure 6.5.

Model	**OA**	**AA**	**AAUC**	**N. of Param.s**	**notes**
L2.1	0.9032	**0.8772**	0.9914	3.83M	–
L3.1	0.9014	0.8702	0.9902	3.83M	–
L4.1	**0.9063**	0.8706	**0.9920**	15.2M	–

Table 6.10: Results of Experiment 4 over ModelNet40, about the performances of the aggregate semantic layer.

Overall Performances over ModelNet40

After having discussed the alternative spatial and semantic layers we presented in chapter 5, we briefly discuss the overall performances we obtained over ModelNet40 and compare them with the other State of the Art methods, as stated in section 6.2.2.

Model	**OA**	**AA**	**AAUC**	**N. of Param.s**	**notes**
PointNet	0.8892	0.8496	0.9899	3.64M	–
KpConv	0.8991	0.8734	0.9892	3.8M	–
ConvPoint	**0.9124**	0.87623	0.9906	3.81M	–
L2.1	0.9032	**0.8772**	0.9914	3.83M	–
L3.1	0.9014	0.8702	0.9902	3.83M	–
L4.1	0.9063	0.8706	**0.9920**	15.2M	–

Table 6.11: Performances over ModelNet40 of some of the previously defined Composite Layers, compared with the other state of the art models.

We report the results obtained by some of our layers in table 6.11, toghether with the results obtained by other state of the art models. In particular, we are comparing:

- the best performing composite layer with linear semantic (L2.1).
- the best performing layer with respect to OA (L4.1, with aggregate semantic layer).

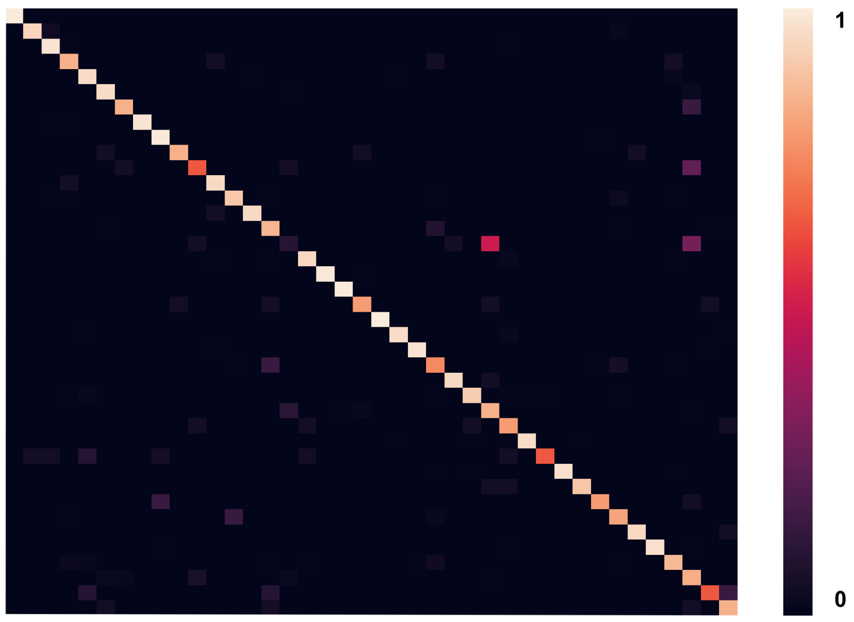

Figure 6.6: Confusion matrix for layer L4.1 in modelNet40, represented as heat-map. Given the high number of classes, class labels are omitted. Each row is normalized such that it sums up to 1.

- a layer with the same configuration as L4.1, but with linear semantic layer (L3.1).

It is possible to see that our layers achieve comparable performances with the existing methods. In particular, we are able to outperform PointNet [37] in all metrics; similarly, we are able to outperform KpConv, though its results in our architectureare worse than those obrained in [51]. On the other hand, Convpoint [4] remains superior in terms of Overall Accuracy, while falling behind in terms of Average Accuracy and Average 1vs1 AUC.

An aspect we are want to investigate is the discrepancy between Overall Accuracy and Average Accuracy, which we registered in all experiments with both our models and state of the art models. To explain this phenomenon, it is useful to observe the confusion matrix of a model. In this case, since all models yield very similar results, we report just the confusion matrix obtained by layer L4.1 in Figure 6.6.

As it is possible to see from the image, one specific class (the class “flower_pot”)

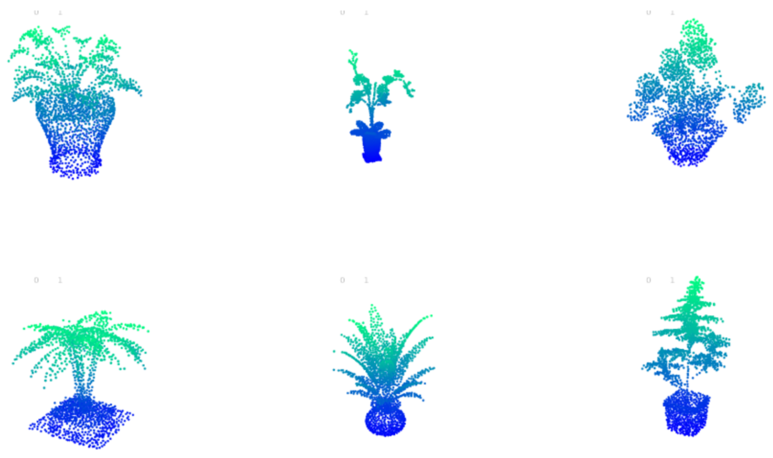

Figure 6.7: Comparison between some instances of the class “flower_pot” (top) and of the class “plant” (bottom) of ModelNet40.

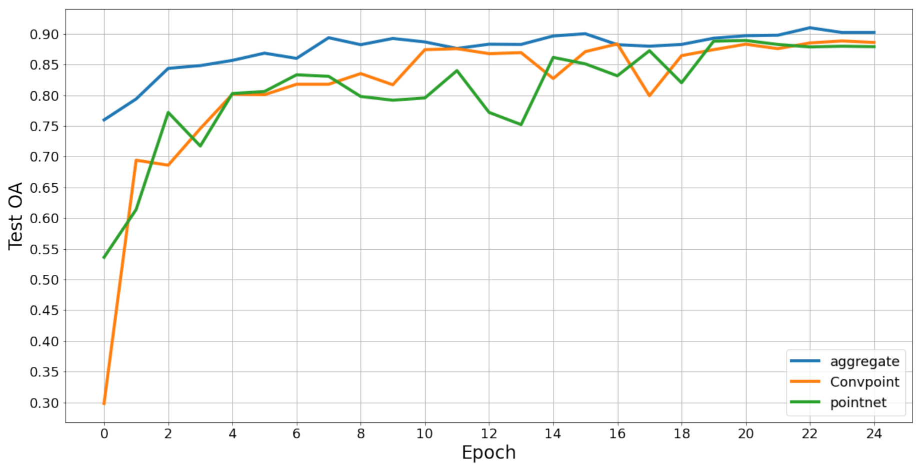

Figure 6.8: Overall Accuracy over the Test Set of ModelNet40, plotted per epoch of training. In this plot only the first 25 epochs are reported. KpConv has a convergence time way higher than 30 epochs and for this reason is omitted.

is almost completely misclassified. In this case, just 2 out of 21 test set instances are correctly classified, yielding a per class accuracy of 9.5%. We can explain this behaviour by noting that 10 out of 21 “flower_pot” instances are classified as part of the “plant” class. This is understandable: the “plant” class contains almost the double

Model	OA	AA	AAUC	N. of Param.s	notes
PointNet	0.8297	0.6856	0.9761	3.64M	–
KpConv	0.8288	0.6712	0.9758	1.9M	–
ConvPoint	0.8366	0.6690	0.9752	1.9M	–
L3.1	0.8286	0.6533	0.9713	1.9M	–
L4.1	**0.8392**	**0.6985**	**0.9762**	7.6M	–

Table 6.12: Performances over ShapeNetCore of some of the previously defined Composite Layers, compared with the other state of the art models.

of the instances than the class “flower_pot”, and the two are very similar when scaled to the same size; it is possible that our models are not able to well distinguish between flower pots and plants, thus assigning “dubious” test instances to the biggest class. To be almost completely inaccurate over a single class can cost more than 2% in terms of Average Accuracy, thus explaining a significant part of the gap between OA and AA. The Overall Accuracy is less influenced by the problem as the samples belonging to “flower_pot” are only a small fraction of the entire dataset.

A second aspect we discuss is the convergence time, in terms of Epochs, of the presented models. This aspect is significant: having a more rapid convergence usually means that the model is easier to optimize. We already briefly discussed of the fact that Layer L4.1 converges faster than layer L3.1 and L2.1 in experiment 4; we repeat this observation by comparing L4.1, ConvPoint and PointNet. As in the previous experiment, L4.1 outperforms the other two layers by converging much faster to its peak performances. It is possible to observe this in figure 6.8. In particular, ConvPoint's behaviour is worth commenting: while being able to achieve better performances in OA than the other two models, in the first 25 epochs ConvPoint performs worse than L4.1 and very similarly to PointNet.

Overall Performances over ShapeNetCore

Until now, we evaluated our model only on ModelNet40 which is quite a simple dataset compared to others. For this reason, we also want to investigate how the different models perform on a larger, more complex and more unbalanced dataset. In our case,

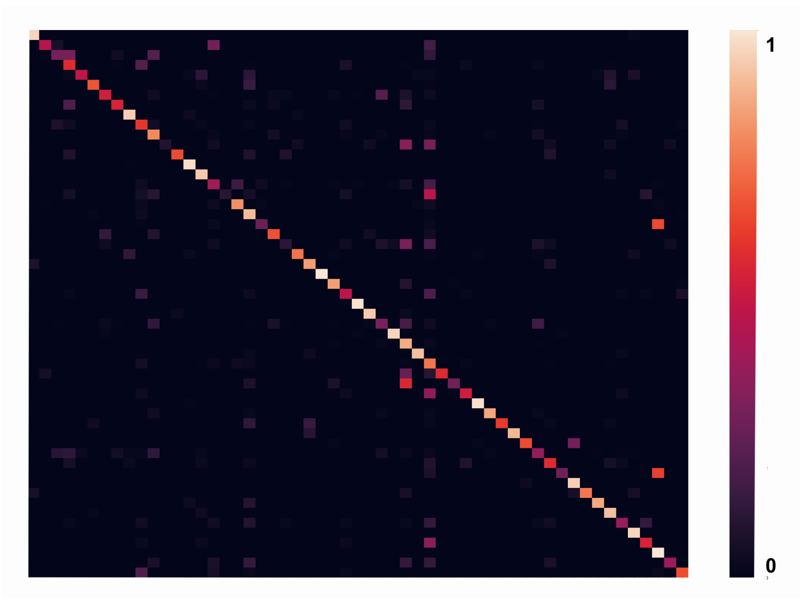

Figure 6.9: Convpoint's Confusion matrix over ShapeNetCore, represented as heat-map. Given the high number of classes, class labels are omitted. Each row is normalized such that it sums up to 1.

the best option in this sense is ShapeNetCore. We already hinted in section 6.1 that ShapeNet is a highly unbalanced dataset, which is an important issue in multiclass classification. For this reason, performances are significantly lower not only for models based on our Composite Layer, but also for other State of the Art models.

To avoid overfitting, we halved the number of incoming and outgoing features from each layer of our networks. For this reason, the first layer of our NN architecture will produce 32 outgoing features instead of 64, the second layer will produce 64 features instead of 128 and so on. This simple measure significantly improved performances in this specific experiment.

As can be seen from table 6.12, all the tested models present a very significant gap between the performances in Average Accuracy and the ones in Overall Accuracy. This is due to the fact that several classes are really small compared to the others, so that the network struggles to correctly classify their instances. This phenomenon is also clear by observing the confusion matrices of the tested models. For example, in figure 6.9 it is possible to spot several vertical bands, corresponding to some of the bigger classes in ShapeNet. The same bands are also present in the confusion matrix produced by our layer L3.1, as in figure 6.10 .On the other hand, they are less visible in the confusion matrix corresponding to L4.1, reported in figure 6.11. This suggests that the use of our aggregate semantic layer makes the model more robust to dataset inbalance, and this hypothesis is also supported by the fact that our aggregate layer achieves an higher performance in terms of per-class Accuracy (AA).

Finally, we want to highlight the fact that all the tested models are rather consistent in terms of Overall Accuracy (OA), since they all reach very similar values. Moreover, the performances in Average 1vs1 AUC tell us that every model is capable of clearly separate the different classes, almost like in the experiments involving ModelNet40.

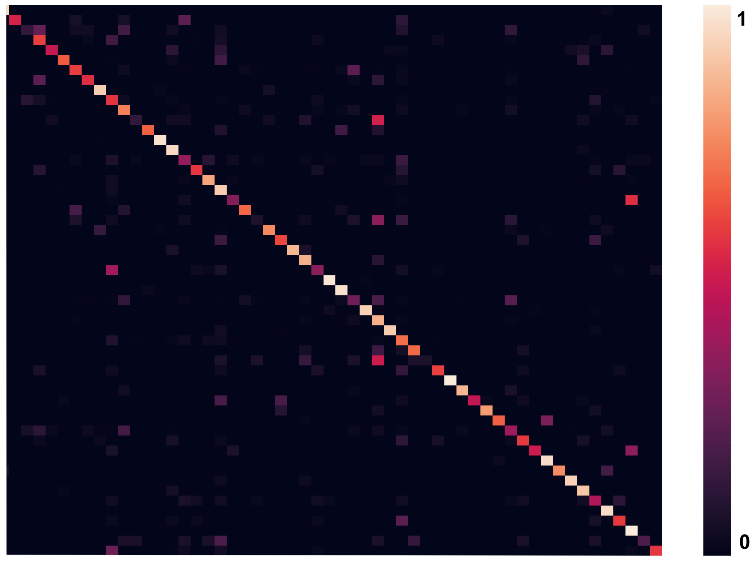

Figure 6.10: Layer L3.1 Confusion matrix over ShapeNetCore, represented as heat-map. Given the high number of classes, class labels are omitted. Each row is normalized such that it sums up to 1.

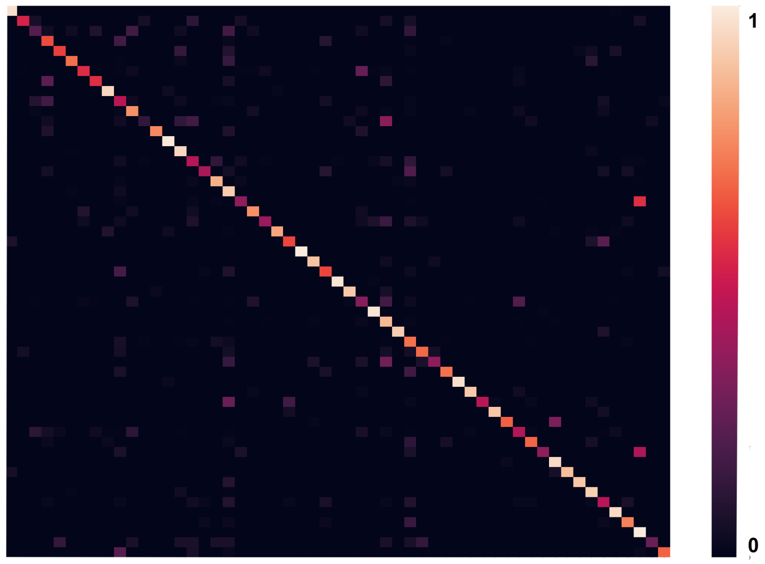

Figure 6.11: Layer L4.1 Confusion matrix over ShapeNetCore, represented as heat-map. Given the high number of classes, class labels are omitted. Each row is normalized such that it sums up to 1.

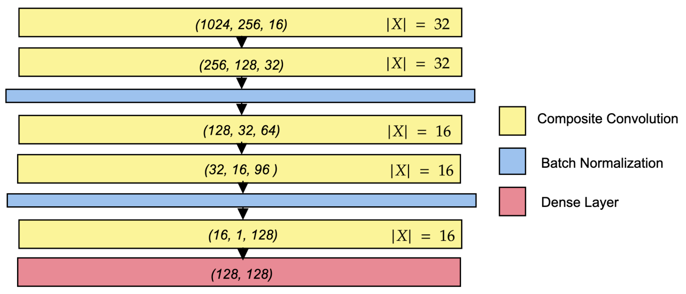

Figure 6.12: Our neural network architecture for OC classification, very similar to the one employed in supervised multiclass classification. Each convolutional layer is labeled with a 3-uple $(|P_I|, |P_O|, \gamma)$ where $|P_I|$ is the number of input points, $|P_O|$ is the number of output points, γ is the number of output features produced by the layer. On the other hand, we label $|X|$ the number of neighbouring points considered in each layer.

6.3 One-Class Classification

The second problem we aim to solve is the one of Anomaly Detection. To the best of our knowledge, there are no models trying to solve this task in the case of point cloud data. For this reason, there is no available baseline to confront our results with. Our proposed approach makes use of the Deep SVDD tecnique described by L. Ruff et al. in [42]. First, we present how to adapt the neural network presented in section 6.2.1 to this different technique: as we shall see, this is not an easy task and requires particular attention. We then present our reference performance metric, and briefly discuss about how to evaluate this kind of unsupervised models. Finally, we present the experiment and its results.

6.3.1 Architecture and hyper-parameters

To define our Neural Network architecture for Deep SVDD, we reprise the same network we used for supervised multiclass classification and modified it to suit our needs better. In particular, the first consideration is that in OC classification we will train our network with only one class. This means that the overall number of samples the network will see would be considerably smaller than in supervised classification. For this reason, we need to reduce the number of learnable parameters to avoid overfitting. A first way to accomplish this is to reduce the number of input and output features between each layer; a second solution may be to reduce each layer's complexity. We decided to employ both solutions: as can be seen from image 6.12, we at least halved the number of ingoing features for each composite layer. Moreover, we noticed that batch normalization (BN) layers could damage the network's performances. For this reason, we removed three BN layers.

Choice of K

The second solution to cope with the smaller training dataset is to reduce each layer's complexity. In our case, this means to decrease the number of spatial descriptors $K = |\{s_k(\tilde{x}_i)\}|$. For this reason, we decided to use $K = 6$, which is substantially less than the usual $K = 16$ we employed in classification. This parameter turned out to be fundamental, since performances are severely harmed when K is too high or too low.

Caveats of Deep SVDD

When describing Deep SVDD in section 5.3, we pointed out some constraints that we need to comply with when defining our Neural Network. We briefly recall them here, and point out some other caveats that we addressed when designing our solution:

- **the Deep SVDD center C needs to be non-zero.** while this requirement seems trivial to satisfy, the choice of the Deep SVDD center C is crucial in terms of performances. Having a center that is too distant from the initial forward pass of the network prevents the model from learning, especially in high-dimensional output spaces. on the other hand, we need the center to be far enough from the origin of the output space, in order to avoid zero-weight solutions. In our case, we chose to perform an initial forward pass $\mathcal{M}^I(\cdot)$ on some training data sample $d := (P_d, \phi_d)$. By indicating C_i the i-th component of C and $\mathcal{M}^I_i(d)$ the i-th component of $\mathcal{M}^I(d)$, we define the Deep SVDD center C as:

$$C_i = \text{sign}(\mathcal{M}^I_i(d))\ \ \max(\,|\mathcal{M}^I_i(d)|,\ 0.1\,)$$

- **there should be no biases inside the network.** For this reason, we need to remove bias terms from each layer of the network, including the batch normalization ones.

- **the output dimensionality ρ greatly influences performances.** Having the network output lying in a 64-dimensional space can lead to very different results compared to having the output lying in a 512-dimensional space. When the number of parameters is large the model has difficulty in learning the task, while when the number of the parameters is small the network tends to learn the most trivial solutions. This parameter needs to be optimized depending on the architecture and the data employed. In our case, we choose $\rho = 128$.

- **the choice of the activation functions matter.** As already pointed out in section 5.3, the use of saturating activation functions can lead the network to learn uninformative solutions. At the same tame, Rectified Linear Units can be subject to the so called "Dying ReLU" problem [30]. For this reason, the best solution is to employ Leaky ReLUs or ELUs as activation functions inside our network.

6.3.2 Dataset and Data preprocessing

In our One-Class Classification experiments, we only employ data from the ShapeNet-Core dataset. This because since we shall train the network to recognize only one class at a time, it is important that the employed classes have a sufficient number of instances: differently from ModelNet, ShapeNetCore possesses several classes with more than 1,000 instances, thus allowing proper training.

To prove our model's performances in OC classification, we shall train the network multiple times, each time to recognize a specific class. In other words, we use multiple training sets, each one comprising instances from a single "normal" class. ShapeNet-Core contains 55 different classes, but since most of them are too small in terms of instances, we select (some of) the most numerous ones to train our model over them. In particular, these classes are reported in table 6.13.

Finally, we spend a few words on data preprocessing. We use the same method described in section 6.2.3 to extract point clouds from ShapeNetCore's meshes. We also employ the same data augmentation techniques we used in supervised Multiclass Classification. Likewise, each point cloud is composed by 1024 points and each point carries no features other than its spatial position.

6.3.3 Evaluation metrics

Deep SVDD classifies a given point cloud as normal or anomaly depending if it is mapped inside the SVDD hypersphere or not. For this reason, the radius of the

Class	**#Instances**
Airplane	2832
Bench	1260
Bowl	1076
Camera	2458
Cap	4612
Clock	5863
Microphone	1620
Monitor	1116
Sofa	1656
Table	2198
Tin_Can	1356

Table 6.13: Classes used as training sets for One-Class classification experiments.

hypersphere is a very important threshold: it distinguishes anomalies from normal instances.

Let us consider the following notation:

- we call **False Negatives (FN)** the number of Normal instances that are classified as Anomalous.
- we call **True Negatives (TN)** the number of Anomalous instances that are classified correctly.
- we call **False Positives (FP)** the number of Anomalous instance that are classified as Normal.
- we call **True Positives (TP)** the number of Normal instance that are classified correctly.

Starting from these definition, we can define two metrics. The first one is called **False Positive Rate** (FPR), and it is defined as:

$$FPR = \frac{FP}{FP + TN}. \tag{6.2}$$

The second metric is called **False Negative Rate** (FNR), and it is defined as:

$$FNR = \frac{FN}{FN + TP}. \tag{6.3}$$

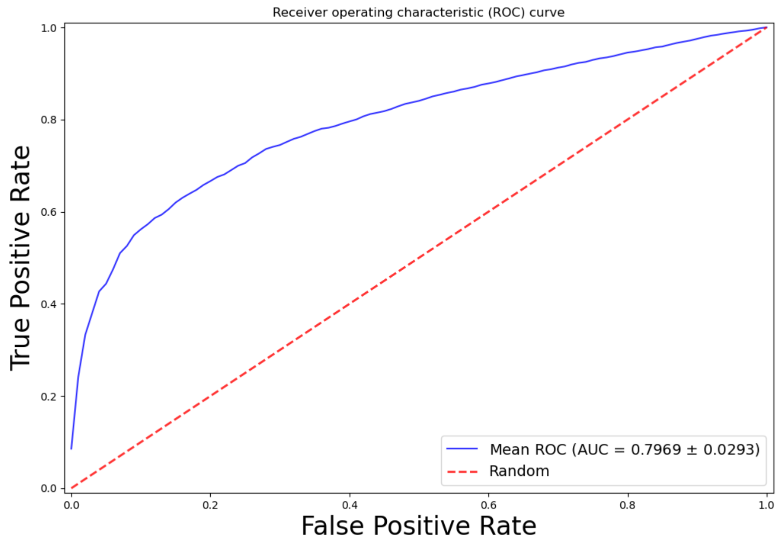

Figure 6.13: An example of ROC curve, produced by our model LOC.2 over the class "Sofa". In the image is also indicated the Area under the ROC curve.

the False Positive and False Negative Rates depend heavily on the hypershphere radius: a "small" hypersphere would probably have an higher False Negative Rate (FPR), while a "big" hypersphere would probably have an higher False Positive Rate (FPR). In fact, every metric that relies on those two definitions would be function of a certain threshold. This is not useful for us in understanding if the model is learning correctly.

For these reasons, to measure our model's performances we follow the same approach used by Ruff in [42]: we use the Area Under the ROC Curve, a threshold-independent metric.

As we already briefly discussed in section 6.2.4, the ROC curve is the curve obtained by plotting the True Positive Rate ($TPR = 1 - FNR$) against the False Positive Rate (FPR) at various threshold settings. The AOC is simply the area behind such ROC curve. Differently from the case of supervised Multiclass Classification, OC classification is a binary classification problem: we are distinguishing one class against all others. For this reason, we can produce a ROC curve for each training on each class, and then calculate its AUC directly. Having an AUC of 1.0 would mean to have optimal performance, while having an AUC of 0.5 would mean having the same performances as a random guesser.

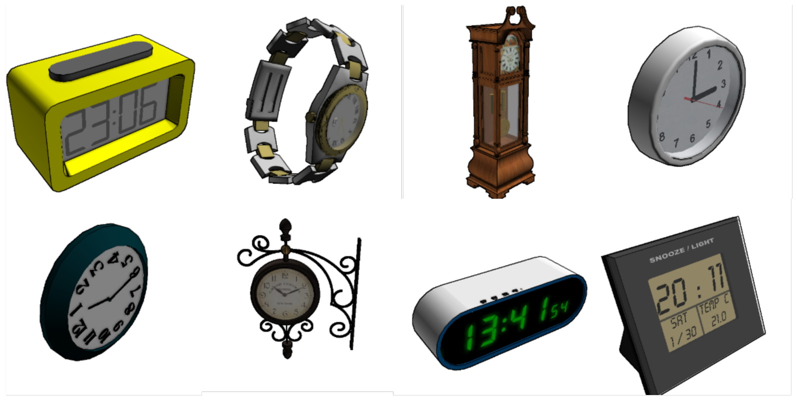

Figure 6.14: Some shapes extracted from the class “clocks” of ShapenetCore.

Name	Spatial L.	ReLu	Semantic L.	K	notes
LOC.1	RBFN	no	Linear	6	16 RBFN centers
LOC.2	RBFN	no	Aggregate	6	16 RBFN centers

Table 6.14: Layers tested in One-Class classification experiments.

6.3.4 Experiments And results

Our experimental framework is quite simple. For each class reported in table 6.13 we perform 10 different trainings. Each trained model should be able to distinguish between the class used in training and the other classes of the Dataset. In other words, we perform a “One vs rest” experiment for each class present in table 6.13. In this case, the training class will contain the “normal” instances, while the rest of the dataset will contain the “anomalous” instances. to evaluate the performances of our model, we employ ROC curve and AUC. Since we perform 10 experiments for each class, our measure will be the average AUC and its standard deviation.

In the case of ShapeNetCore, an official three-way (Train/ Test/ Validation) split is defined [7]. We rely on the official validation set to perform hyperparameter-tuning, while using the official test set to evaluate the models.We perform this experiment for two different Models, with the same architecture but different Composite Layers. Such Layers are summarized in table 6.14.

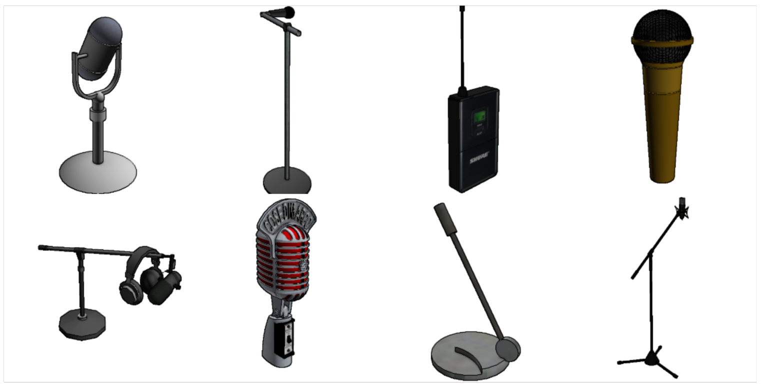

Figure 6.15: Some shapes extracted from the class “microphones” of ShapenetCore.

The last consideration is about the Loss Function. In section 5.3, we presented two alternatives: the so-called *soft-boundary Deep SVDD loss* presented in (5.13) and the *One Class Deep SVDD loss* presented in (5.14). Since ShapeNetCore is a dataset with high intraclass variability, we employ in these experiment the *soft-boundary Deep SVDD loss* 5.13. This because such loss is specifically designed for problems where it is possible to encounter outliers inside the training data.

The results obtained by our two models are presented in table **??**. As we can see, the performances in terms of AUC are -for many classes- quite far from the optimal AUC of 1. This can be due to many different factors:

- **many classes have high intra-class variability**. An extreme example is the class “microphone”: as shown in figure 6.15, such class contains both table microphones, hand microphones and microphones mounted on floor stands. Considering that our network is designed to be scale-invariant, this fact poses quite a challenge for our OC classification network.

- **certain objects are not recognizable by their geometrical shape**. An example of this facy is shown in figure 6.14. In this case, the feature that distinguishes clocks from other instances in the dataset is not the geometrical shape, but the fact that each clock has a dial, either analog or digital. This detail is not represented in the point clouds we are using, as each point is not associated with any feature other than its geometrical position. On the other hand this kind of 3D shapes is easily recognizable by using Multiview CNNs, which analyse shape images.

- **Some classes are really similar to others in terms of shape**. For example, "bowls" and "caps" are really similar in terms of shape but different in terms of function. Another example may be to confront cylindrical or rectangular "Tin Cans" with cylindrical or rectangular "clocks". Without context and features associated to the point cloud's points, it can be quite difficult to distinguish between the two.

Class	**LOC.1**		**LOC.2**	
	AAUC	σ	AAUC	σ
Airplane	60.02%	±4.17%	**70.10**%	±3.05%
Bench	58.84%	±4.61%	**61.70**%	±1.35%
Bowl	47.48%	±1.89%	**62.78**%	±2.38%
Camera	55.56%	±9.59%	**66.85**%	±7.49%
Cap	46.33%	±5.54%	**56.70**%	±1.60%
Clock	51.25%	±1.79%	**55.36**%	±2.42%
Microphone	**54.43**%	±4.33%	52.35%	±1.79%
Monitor	51.03%	±6.10%	**58.61**%	±1.59%
Sofa	76.59%	±7.81%	**79.69**%	±2.93%
Table	48.03%	±4.35%	**60.68**%	±1.72%
Tin_Can	**64.33**%	±4.97%	61.47%	±7.33%

Table 6.15: Comparison in terms of Average AUC and standard deviation (over 10 experiments) of layers LOC.1 (having a linear semantic layer) and LOC.2 (having an aggregate semantic layer). We show AUC in percentages to enhance the readability of the standard deviations.

Note that these problems are common also in different datasets, other than ShapeNet-Core. For example, Ruff et al. in [42] tested their model over the CIFAR10 [25] dataset and obtained quite similar results to ours in terms of Average AUC. In this sense, our results should be interpreted as a proof of concept: it is possible to perform OC classification over point cloud with the help of Deep SVDD. In particular, we see that LOC.2 perform way better in almost all tested classes than LOC.1, both in terms of mean AUC and in terms of standard deviation. The only difference between the two composite layers is the use of the aggregate semantic layer instead of the standard linear later. This fact suggest us that the aggregate semantic layer can help, at least in this

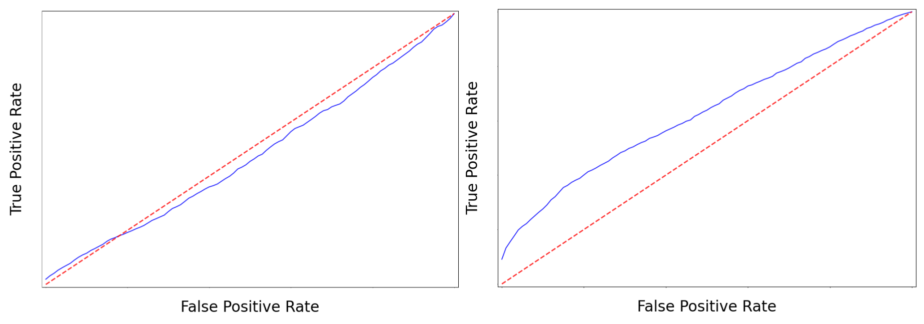

Figure 6.16: ROC curve of LOC.1 (left) and LOC.2 (right) over class “Bowl”. The red line indicates random guesser performances. Ticks on the axes are omitted to improve readability.

kind of framework, in obtaining better OC classification performances.

In figure 6.16 it is possible to see more clearly the difference in terms of performances between the two layers. In this case, we are comparing two ROC curves obtained over the class “bowl”. As we can see, performances of LOC.1 are worse than random guessing, while LOC.2 achieves considerably better performances. It is also interesting to note that the ROC curve for LOC.2 starts with a non-zero true positive rate since the start of the ROC curve corresponds to the minimum threshold, this means that for hypersphere radius $\rho \approx 0$ the deep SVDD still recognizes something as not-anomalous. In other words, certain test inputs are correctly mapped very close to the hypershpere’s center. Another example is shown in figure **??**, this time regarding the class “airplane”. In this case, both the layers are achieving better performances than random guessing, but LOC.2 is able to learn better than LOC.1. The fact that our network is able to better learn how to recognize this latter class than the previous one is possibly related to how the class itself is defined there are less elements similar to airplanes in ShapeNet, and airplanes have in most cases a very specific shape. This is probably also the case for class “sofa”, which is the best recognized one. Also in this case all instances share very similar shape, while being sufficiently different from the rest of the dataset.

For these reasons, it is clear that the dataset choice is crucial when evaluating an OC classification model such as ours. Unfortunately, we did not have the possibility of choosing a dataset specifically designed for anomaly detection or unsupervised classification with point clouds, since no similar projects are nowadays available. A possible future development in the field can surely be the definition of a suitable benchmarking dataset, in which most of the information carried by each datum is

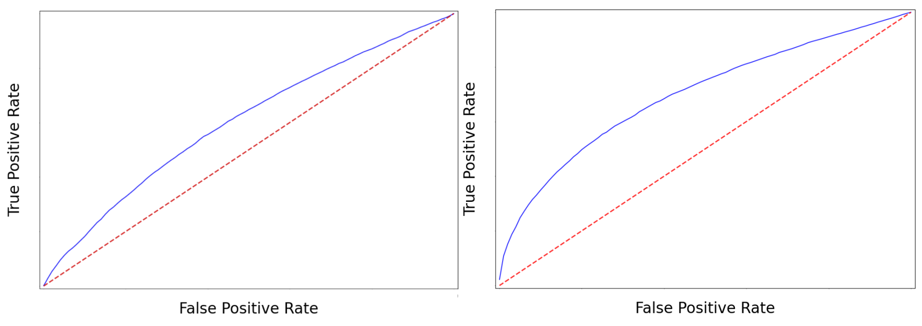

Figure 6.17: ROC curve of LOC.1 (left) and LOC.2 (right) over class "Airplane". The red line indicates random guesser performances. Ticks on the axes are omitted to improve readability.

encoded in its shape.

Chapter 7

Conclusions

In this thesis, we defined a novel convolutional layer, called *Composite Layer*, thought to be employed over Point Clouds. Chapter 6 shows that our solution can achieve comparable performances with respect to other State of the Art Methods. In particular when dealing with multiclass classification, the *aggregate layer* has shown to be quite promising: it leads to better convergence time and obtained the best performances among the various possible Composite Layers. A possible future development may be to investigate other non-convolutional ways to aggregate neighbourhood and features. On the other hand, the most interesting results are obtained in One-Class Classification: we showed that it is possible to apply Deep-SVDD over Point Clouds, a development not yet shown in literature. OC Classification over Point Clouds may have several application in engineering and industry, and for this reason it is surely a topic that worth further investigation. To allow such developments, the development of a dataset dedicated to Anomaly Detection is crucial: the resources now available (ModelNet40 and ShapeNetCore) have several problems that make difficult to apply OC Classification directly. Having such novel dataset, it would be possible to easily compare different models and to understand which one yield better performances. Moreover, it would be possible to compare Deep SVDD with completely methods, like ones based on AutoEncoders. A final remark is about the "bias problem" we presented in section 5.4.1: a surely helpful development would be to show its effect during Leaning, and explore new ways to avoid it.

As technology develops in all fields regarding Computer Vision, so the possible application for Point Cloud Deep Learning grows: for this reason, it is sure that works dealing with this peculiar data structure will become increasingly relevant in the scientific community.

List of Figures

List of Tables

Bibliography

[1] *3D model from here:* https://3dwarehouse.sketchup.com/model/541ad6a69f87b134f4c1adce71073351/Northwest-DC9B727-collision. Accessed: 2020-03-05.

[2] Matan Atzmon, Haggai Maron, and Yaron Lipman. “Point Convolutional Neural Networks by Extension Operators”. In: *ACM Trans. Graph.* 37.4 (July 2018). ISSN: 0730-0301. URL: https://doi.org/10.1145/3197517.3201301.

[3] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[4] Alexandre Boulch. “ConvPoint: Continuous convolutions for point cloud processing”. In: *Computers and Graphics* 88 (2020), pp. 24–34. ISSN: 0097-8493. DOI: https://doi.org/10.1016/j.cag.2020.02.005. URL: http://www.sciencedirect.com/science/article/pii/S0097849320300224.

[5] D.S. Broomhead and D. Lowe. “Multivariable Functional Interpolation and Adaptive Networks”. In: *Complex Systems* 2 (1988), pp. 321–355.

[6] Varun Chandola, Arindam Banerjee, and Vipin Kumar. “Anomaly Detection: A Survey”. In: *ACM Comput. Surv.* 41.3 (July 2009). ISSN: 0360-0300. DOI: 10.1145/1541880.1541882. URL: https://doi.org/10.1145/1541880.1541882.

[7] Angel X. Chang et al. *ShapeNet: An Information-Rich 3D Model Repository*. cite arxiv:1512.03012. 2015. URL: http://arxiv.org/abs/1512.03012.

[8] Weikai Chen et al. *Deep RBFNet: Point Cloud Feature Learning using Radial Basis Functions*. 2019. arXiv: 1812.04302 [cs.CV].

[9] J. Dai et al. “Deformable Convolutional Networks”. In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 764–773. DOI: 10.1109/ICCV.2017.89.

[10] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. “Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering”. In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. NIPS’16. Barcelona, Spain: Curran Associates Inc., 2016, pp. 3844–3852. ISBN: 9781510838819.

[11] M. Deuge et al. “Unsupervised feature learning for classification of outdoor 3D Scans”. In: *Australasian Conference on Robotics and Automation, ACRA* (Jan. 2013).

[12] A. Dewan, G. L. Oliveira, and W. Burgard. “Deep semantic classification for 3D LiDAR data”. In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017, pp. 3544–3549. DOI: 10.1109/IROS.2017.8206198.

[13] Thomas Fabry, Dirk Smeets, and Dirk Vandermeulen. “Surface representations for 3D face recognition”. In: Apr. 2010. ISBN: 978-953-307-060-5. DOI: 10.5772/8951.

[14] Matthias Fey et al. *SplineCNN: Fast Geometric Deep Learning with Continuous B-Spline Kernels*. 2018. arXiv: 1711.08920 [cs.CV].

[15] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. “Deep Sparse Rectifier Neural Networks.” In: *AISTATS*. Ed. by Geoffrey J. Gordon, David B. Dunson, and Miroslav Dudík. Vol. 15. JMLR Proceedings. JMLR.org, 2011, pp. 315–323. URL: http://dblp.uni-trier.de/db/journals/jmlr/jmlrp15.html#GlorotBB11.

[16] J. B. Gomm and D. L. Yu. “Selecting radial basis function network centers with recursive orthogonal least squares training”. In: *IEEE Transactions on Neural Networks* 11.2 (2000), pp. 306–314. DOI: 10.1109/72.839002.

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[18] Ben Graham. “Sparse 3D convolutional neural networks”. In: (2015). arXiv: 1505.02890 [cs.CV].

[19] Yulan Guo et al. “Deep Learning for 3D Point Clouds: A Survey”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2019).

[20] David J. Hand and Robert J. Till. “A Simple Generalisation of the Area Under the ROC Curve for Multiple Class Classification Problems”. In: *Mach. Learn.* 45.2 (Oct. 2001), pp. 171–186. ISSN: 0885-6125. DOI: `10.1023/A:1010920819831`. URL: `https://doi.org/10.1023/A:1010920819831`.

[21] Pedro Hermosilla, Tobias Ritschel, and Timo Ropinski. “Total Denoising: Unsupervised Learning of 3D Point Cloud Cleaning”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. Oct. 2019.

[22] A.Kerstens J.-P.Kruth. “Reverse engineering modelling of free-form surfaces from point clouds subject to boundary conditions”. In: *Journal of Materials Processing Technology* 76.1-3 (1998), pp. 120–127. DOI: `https://doi.org/10.1016/S0924-0136(97)00341-5`.

[23] N. B. Karayiannis. “Reformulated radial basis neural networks trained by gradient descent”. In: *IEEE Transactions on Neural Networks* 10.3 (1999), pp. 657–671. DOI: `10.1109/72.761725`.

[24] R. Klokov and V. Lempitsky. “Escape from Cells: Deep Kd-Networks for the Recognition of 3D Point Cloud Models”. In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 863–872. DOI: `10.1109/ICCV.2017.99`.

[25] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. “CIFAR-10 (Canadian Institute for Advanced Research)”. In: (). URL: `http://www.cs.toronto.edu/~kriz/cifar.html`.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: `https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf`.

[27] Yann LeCun and Corinna Cortes. “MNIST handwritten digit database”. In: (2010). URL: `http://yann.lecun.com/exdb/mnist/`.

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

[29] Yangyan Li et al. “PointCNN: Convolution On X-Transformed Points”. In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio et al. Vol. 31. Curran Associates, Inc., 2018, pp. 820–830.

[30] Lu Lu. “Dying ReLU and Initialization: Theory and Numerical Examples”. In: *Communications in Computational Physics* 28.5 (June 2020), pp. 1671–1706. ISSN: 1991-7120. DOI: `10.4208/cicp.oa-2020-0165`. URL: `http://dx.doi.org/10.4208/cicp.OA-2020-0165`.

[31] Martın Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[32] George A. Miller. “WordNet: A Lexical Database for English”. In: *Commun. ACM* 38.11 (Nov. 1995), pp. 39–41. ISSN: 0001-0782. DOI: `10.1145/219717.219748`. URL: `https://doi.org/10.1145/219717.219748`.

[33] Tom Mitchell. *Machine Learning*. `http://www.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ftp/mlbook.html`. McGraw Hill, 1997.

[34] Carsten Moenning and Neil A. Dodgson. “Fast Marching farthest point sampling”. In: *Eurographics 2003 - Posters*. Eurographics Association, 2003. DOI: `10.2312/egp.20031024`.

[35] Tien Nguyen, Gu Xi, and Liu X.G. “Analysis Of Error Sources In Terrestrial Laser Scanning”. In: *Joint ISPRS Workshop on 3D City Modelling and Applications and the 6th 3D GeoInfo, 3DCMA 2011* (Jan. 2011).

[36] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[37] Charles R Qi et al. “PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation”. In: *arXiv preprint arXiv:1612.00593* (2016).

[38] Charles R Qi et al. “PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space”. In: *arXiv preprint arXiv:1706.02413* (2017).

[39] Y. Regaya, F. Fadli, and A. Amira. “3D Point Cloud Enhancement using Unsupervised Anomaly Detection”. In: *2019 International Symposium on Systems Engineering (ISSE)*. 2019, pp. 1–6. DOI: `10.1109/ISSE46696.2019.8984428`.

[40] G. Riegler, A. O. Ulusoy, and A. Geiger. "OctNet: Learning Deep 3D Representations at High Resolutions". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 6620–6629. DOI: 10.1109/CVPR.2017.701.

[41] Yossi Rubner, Carlo Tomasi, and Leonidas J. Guibas. "The Earth Mover's Distance as a Metric for Image Retrieval". In: *International Journal of Computer Vision* 40.2 (Nov. 2000), pp. 99–121. ISSN: 1573-1405. DOI: 10.1023/A:1026543900054. URL: https://doi.org/10.1023/A:1026543900054.

[42] Lukas Ruff et al. "Deep One-Class Classification". In: *Proceedings of the 35th International Conference on Machine Learning* 80 (2018), pp. 4393–4402.

[43] Sara Sabour, Nicholas Frosst, and Geoffrey E. Hinton. "Dynamic Routing between Capsules". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, pp. 3859–3869. ISBN: 9781510860964.

[44] Manolis Savva, Angel X. Chang, and Pat Hanrahan. "Semantically-Enriched 3D Models for Common-sense Knowledge". In: *CVPR 2015 Workshop on Functionality, Physics, Intentionality and Causality* (2015).

[45] B. Schölkopf et al. *Estimating the support of a high-dimensional distribution*. Tech. rep. MSR-TR-99-87. Microsoft Research, 1999.

[46] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556 (2014). URL: http://arxiv.org/abs/1409.1556.

[47] H. Su et al. "Multi-view Convolutional Neural Networks for 3D Shape Recognition". In: *2015 IEEE International Conference on Computer Vision (ICCV)*. 2015, pp. 945–953. DOI: 10.1109/ICCV.2015.114.

[48] An Tao. *Point Cloud Datasets*. URL: https://github.com/AnTao97/PointCloudDatasets.

[49] David M.J. Tax and Robert P.W. Duin. "Support Vector Data Description". In: *Machine Learning* 54.1 (Jan. 2004), pp. 45–66. ISSN: 1573-0565. DOI: 10.1023/B:MACH.0000008084.60811.49. URL: https://doi.org/10.1023/B:MACH.0000008084.60811.49.

[50] Simone Teruggi et al. "A Hierarchical Machine Learning Approach for Multi-Level and Multi-Resolution 3D Point Cloud Classification". In: *Remote Sensing* 12 (Aug. 2020), p. 2598. DOI: 10.3390/rs12162598.

[51] Hugues Thomas et al. "KPConv: Flexible and Deformable Convolution for Point Clouds". In: *Proceedings of the IEEE International Conference on Computer Vision* (2019).

[52] Mikaela Angelina Uy et al. "Revisiting Point Cloud Classification: A New Benchmark Dataset and Classification Model on Real-World Data". In: *International Conference on Computer Vision (ICCV)*. 2019.

[53] S. Wang et al. "Deep Parametric Continuous Convolutional Neural Networks". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, pp. 2589–2597. DOI: 10.1109/CVPR.2018.00274.

[54] K. Warwick, J. D. Mason, and E. L. Sutanto. "Neural network basis function center selection using cluster analysis". In: *Proceedings of 1995 American Control Conference - ACC'95*. Vol. 5. 1995, 3780–3781 vol.5. DOI: 10.1109/ACC.1995.533845.

[55] Zhirong Wu et al. "3d shapenets: A deep representation for volumetric shapes". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1912–1920.

[56] Yifan Xu et al. "SpiderCNN: Deep Learning on Point Sets with Parameterized Convolutional Filters". In: *Computer Vision – ECCV 2018*. Ed. by Vittorio Ferrari et al. Cham: Springer International Publishing, 2018, pp. 90–105. ISBN: 978-3-030-01237-3.

[57] J. Yang et al. "Modeling Point Clouds With Self-Attention and Gumbel Subset Sampling". In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, June 2019, pp. 3318–3327. DOI: 10.1109/CVPR.2019.00344. URL: https://doi.ieeecomputersociety.org/10.1109/CVPR.2019.00344.

[58] Y. Yang et al. "FoldingNet: Point Cloud Auto-Encoder via Deep Grid Deformation". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, pp. 206–215. DOI: 10.1109/CVPR.2018.00029.

[59] Li Yi et al. "A Scalable Active Framework for Region Annotation in 3D Shape Collections". In: *SIGGRAPH Asia* (2016).

[60] Biaobiao Zhang Yue Wu Hui Wang and K.-L. Du. “Using Radial Basis Function Networks for Function Approximation and Classification”. In: *International Scholarly Research Notices* 2012 (2012), p. 34. DOI: `https://doi.org/10.5402/2012/324194`.

[61] Manzil Zaheer et al. “Deep Sets”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017, pp. 3391–3401. URL: `https://proceedings.neurips.cc/paper/2017/file/f22e4747da1aa27e363d86d40ff442fe-Paper.pdf`.

[62] Maciej Zamorski et al. “Adversarial autoencoders for compact representations of 3D point clouds”. In: *Computer Vision and Image Understanding* 193 (2020), p. 102921. ISSN: 1077-3142. DOI: `https://doi.org/10.1016/j.cviu.2020.102921`. URL: `https://www.sciencedirect.com/science/article/pii/S107731422030014X`.

[63] Y. Zhao et al. “3D Point Capsule Networks”. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, pp. 1009–1018. DOI: `10.1109/CVPR.2019.00110`.

[64] Yong Zhou et al. “Voxelization modelling based finite element simulation and process parameter optimization for Fused Filament Fabrication”. In: *Materials and Design* 187 (2020), p. 108409. ISSN: 0264-1275. DOI: `https://doi.org/10.1016/j.matdes.2019.108409`. URL: `https://www.sciencedirect.com/science/article/pii/S0264127519308470`.