Escuela Politécnica Superior

3D Object Recognition with Convolutional Neural Networks

Master's Degree in Automation and Robotics

Master's Thesis

Author:
Alberto García García

Advisors:
José García Rodríguez
Jorge Pomares Baeza

Universitat d'Alacant
Universidad de Alicante

June 2016

UNIVERSITY OF ALICANTE

MASTER'S THESIS

3D Object Recognition with Convolutional Neural Networks

Author
Alberto GARCIA-GARCIA

Advisors
Jose GARCIA-RODRIGUEZ
Jorge POMARES-BAEZA

A thesis submitted in fulfilment of the requirements for the degree of Master of Science

in the

Master's Degree in Automation and Robotics
Department of Physics, Systems Engineering, and Signal Theory

May 27, 2016

This document was proudly made with LaTeXand Ti*k*Z.

"Will robots inherit the earth? Yes, but they will be our children."

Marvin Minsky

Abstract

In this work, we propose the implementation of a 3D object recognition system using Convolutional Neural Networks. For that purpose, we first analyzed the theoretical foundations of that kind of neural networks. Next, we discussed ways of representing 3D data in a compact and structured manner to feed the neural network. Those representations consist of a grid-like structure (fixed and adaptive) and a measure for the occupancy of each cell of the grid (binary, normalized point density, and surface intersection). At last, 2.5D and 3D Convolutional Neural Network architectures were implemented and tested using those volumetric representations. The experimentation included an in-depth study of their performance in synthetically simulated adverse conditions that characterize the real-world, i.e., noise and occlusions. The resulting system, the best one out of that experimentation, is able to efficiently recognize objects in three dimensions with a success rate of 85% in a common household CAD objects dataset.

Resumen

En este trabajo proponemos la implementación de un sistema de reconocimiento de objetos 3D utilizando Redes Neuronales Convolucionales. Para este propósito, analizamos en primer lugar los fundamentos teóricos de este tipo de redes neuronales. Seguidamente, discutimos formas de representar datos 3D de una manera compacta y estructurada para proporcionarlos como entrada a la red neuronal. Estas representaciones consistirán en una estructura de malla (fija o adaptativa) y de una medida de la ocupación de cada elemento de dicha malla (binaria, densidad de puntos normalizada o intersección de superficie). Por último, implementamos arquitecturas de Redes Neuronales Convolucionales 2.5D y 3D y las testeamos empleando las representaciones volumétricas antes descritas. La experimentación incluyó un estudio detallado sobre el rendimiento de dichas redes en condiciones adversas, simuladas de forma sintética,que caracterizan a las escenas del mundo real, es decir, ruido y oclusiones. Como resultado se ha obtenido un sistema capaz de reconocer de forma eficiente objetos en tres dimensiones con una tasa de acierto del 85% en un conjunto de datos de objetos CAD comunes del hogar.

Acknowledgements

Let us please observe a moment of silence for the many brave coffee beans that gave their lives, so that I could finish this Thesis in time...

Thanks NVIDIA for the hardware donations. Currently running Doom at 200 FPS. Experiments were fast too.

Contents

List of Figures

List of Tables

List of Acronyms

1D one-dimensional

2D two-dimensional

2.5D two-and-a-half-dimensional

3D three-dimensional

Adam Adaptive Moment Estimation

AMT Amazon Mechanical Turk

API Application Program Interface

BRIEF Binary Robust Independent Elementary Features

BRISK Binary Robust Invariant Scalable Keypoints

BVLC Berkeley Vision and Learning Center

CAD Computer Aided Design

CDBN Convolutional Deep Belief Network

CIFAR Canadian Institute for Advanced Research

CLI Command Line Interface

CN Computational Network

CNN Convolutional Neural Network

CNTK Computational Network Toolkit

CSO Computer Science and Operations

CUDA Compute Unified Device Architecture

cuDNN CUDA Deep Neural Network

FAIR Facebook Artificial Intelligence Research

FPGA Field Programmable Gate Array

FREAK Fast Retina Keypoint

GCC GNU Compiler Collection

GNU GPLv3 GNU General Public License v3.0

GPU Graphics Processing Unit

HDD Hard Disk Drive

IR Infrarred

JIT Just In Time

LIDAR Light Detection and Ranging

MEL Model Editing Language

MLP Multi-Layer Perceptron

MNIST Mixed National Institute of Standards and Technology

NAG Nesterov accelerated Gradient

NDL Network Definition Language

OFF Object File Format

OpenCL Open Computing Language

OpenMP Open Multi-Processing

ORB Oriented FAST and Rotated BRIEF

PCD Point Cloud Data

PCL Point Cloud Library

POV Point of View

RAID Redudant Array of Independent Disks

RBF Radial Basis Function

ReLU Rectified Linear Unit

PReLU Parametric Rectified Linear Unit

RGB Red Green and Blue

RGB-D RGB-Depth

RNN Recursive Neural Network

RMS Root Mean Squared

SGD Stochastic Gradient Descent

SIFT Scale Invariant Feature Transform

SSD Solid State Drive

SSH Secure Shell

SVM Support Vector Machine

SURF Speeded Up Robust Features

TSDF Truncated Signed Distance Function

Chapter 1

Introduction

This first chapter introduces the main topic of this work. It is organized as follows. Section 1.1 sets up the framework for the activities performed during this thesis. Section 1.2 introduces the motivation of this work. Section 1.3 elaborates a state of the art of object recognition systems and the evolution of the problem during the last years. Section 1.4 lays down the proposal developed in this work and presents the main and specific goals of this project. In the end, Section 1.5 details the structure of this document.

1.1 Overview

In this Master's Thesis, we have performed a theoretical and practical research focused on the problem of 3D object class recognition. The main goal is the proposal, design, implementation, and validation of an efficient and accurate 3D object recognition system. We leveraged deep learning techniques to solve this problem, specifically Convolutional Neural Networks. This work comprises the study of the theoretical foundations of CNNs, the analysis of volumetric representations for three-dimensional data, and the implementation of a 3D CNN.

1.2 Motivation

This document presents the work carried out to prove the knowledge acquired during the *Master's Degree in Automation and Robotics*, taken between the years 2015 and 2016 at the *University of Alicante*. This work follows up the line started with my Bachelor's Thesis [1][2]. The motivation for this work is twofold.

On the one hand, part of this work was carried out during the research collaboration period with the *Industrial Informatics and Computer Networks* research group of the *Department of Computer Technology* at the *University of Alicante*. This collaboration was supervised by Prof. José García-Rodríguez and funded by the grant "Ayudas para estudios de master oficial e iniciación a la investigación" from the "Desarrollo e innovación para el fomento de la I+D+i en la Universidad de Alicante" programme. The purpose of this collaboration is to introduce students into a certain line of research, in our case we were mainly focused on deep learning applied to computer vision.

On the other hand, this work has been performed under the frame of the *SIRMAVED: Development of a comprehensive robotic system for monitoring and interaction for people with acquired brain damage and dependent people* [3] national project (identifier code DPI2013-40534-R). Within this project, an object recognition system is needed to identify instances in the environment before handling or grasping them. This project is funded by the Ministerio de Economía y Competitividad (MEC) of Spain with professors José García-Rodríguez and Miguel Ángel Cazorla-Quevedo from the University of Alicante as main researchers.

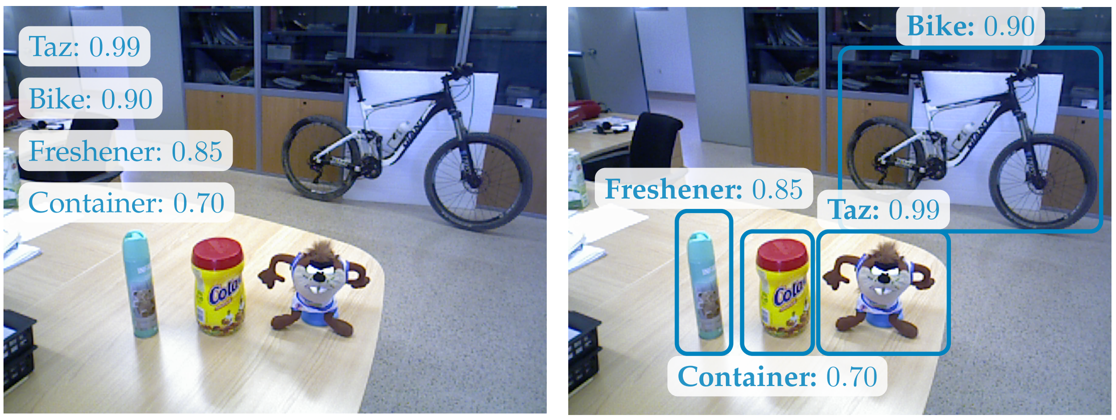
1.3 Related Works

In the context of computer vision, object classification and detection are two of the most challenging tasks required to achieve scene understanding. The former focuses on predicting the existence of objects within a scene, whereas the latter aims to localize those objects. Figure 1.1 shows a visual comparison of both tasks.

Since the very beginning of the computer vision research field, a considerable amount of effort has been directed towards achieving robust object recognition systems that are able to semantically classify a scene and detect objects within it, also providing their estimated poses [4]. This is due to the fact that semantic object recognition is a key capability required by cognitive robots to being able to operate autonomously in unstructured, real-world environments.

1.3.1 Feature-based Object Recognition

Over the past few years, intensive research has been done on feature-based object recognition methods. This approach relies on extracting features, i.e., pieces of information which describe simple but significant properties of the objects within the scene. Those features are encoded into *descriptors* such as Scale Invariant Feature Transform (SIFT)[5], Speeded Up Robust Features (SURF)[6], Binary Robust Independent Elementary Features (BRIEF)[7], Binary Robust Invariant Scalable Keypoints (BRISK)[8], Oriented FAST and Rotated BRIEF (ORB)[9], or Fast Retina Keypoint (FREAK)[10] to name a few. After extracting those descriptors, machine learning techniques are applied to train a system with them so that it becomes able to classify features extracted from unknown scenes. Based on the used types of features, these methods can be divided into two categories: global or local feature-based methods. Global ones are characterized by dealing with the object as a whole; they define a set of features which completely encompass the object and describe it effectively. On the other hand, local



(a) Object classification. **(b)** Object detection.

Figure 1.1: The object classification task focuses on providing a list of objects that are present in the provided image with an associated probability, as shown in Subfigure (a). On the other hand, the object detection task exhibits more complexity since it provides a bounding-box or even an estimate pose for each object in the image, Subfigure (b) shows an example of detection.

methods describe local patches of the object, those regions are located around highly distinctive spots of the object named *keypoints*.

Real-world scenes tend to be unstructured environments. This implies that object recognition systems must not be affected by clutter or partial occlusions. In addition, they should be invariant to illumination, transforms, and object variations. Those are the main reasons why local surface feature-based methods have been popular and successful during the last years – since they do not need the whole object to describe it properly, they are able to cope with cluttered environments and occlusions [11].

1.3.2 Usage of 3D Data

Traditionally, those object recognition systems made use of 2D images with intensity or color information, i.e., Red Green and Blue (RGB) images. However, technological advances made during the last years have caused a huge increase in the usage of 3D information. The field of computer vision in general, and object recognition research in particular, have been slowly but surely moving towards including this richer information into their algorithms.

Nowadays, the use of 3D information for this task is in a state of continuous evolution, still far behind, in terms of maturity, from the systems that make use of 2D images. Nevertheless, the use of 2D information exhibits a handful of problems which hinder the development of robust object recognition systems. Oppositely, the use of range images or point clouds, which provide 2.5D or 3D information respectively, presents many significant benefits over traditional 2D-based systems. Some of the main advantages are the following ones [12]: (1) they provide geometrical information thus removing surface ambiguities, (2) many of the features that can be extracted are not affected by illumination or even scale changes, (3) pose estimation is more accurate due to the increased amount of surface information. Therefore, the use of 3D data has become a solid choice to overcome the inherent hurdles of traditional 2D methods.

However, despite all the advantageous assets of 3D data, researchers had to overcome certain difficulties or drawbacks. On the one hand, sensors capable of providing 3D were expensive, limited, and performed poorly in many cases. The advent of low-cost 3D acquisition systems, e.g., Microsoft Kinect, enabled a widespread adoption of these kind of sensors thanks to their accessibility and affordability. On the other hand, 3D object recognition systems are computationally intensive due to the increased

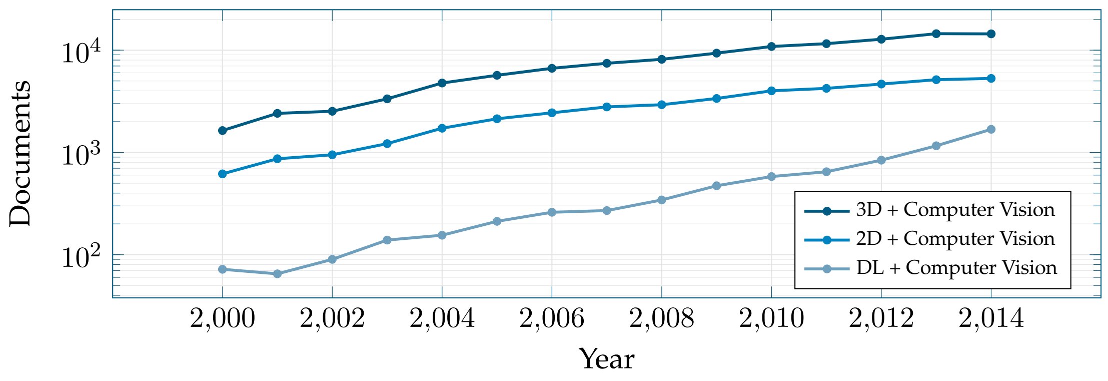


Figure 1.2: Evolution of the number of academic documents containing the terms 2D, 3D, and Deep Learning together with *Computer Vision*. Search terms statistics obtained from scopus.com.

dimensionality. In this regard, advances in computing devices like GPUs provided enough computational horsepower to run those algorithms in an efficient manner. In addition, the availability of low-power GPU computing devices like the NVIDIA Jetson TK1 has supposed a significant step towards deploying robust and powerful object recognition systems in mobile robotic platforms.

The combination of those three factors (the advantages of 3D data, low-cost sensors, and parallel computing devices) transformed the field of computer vision in general, and object recognition in particular. As we can see in Figure 1.2, there has been a significant dominance of 3D over 2D research in computer vision since the year 2000.

Therefore, creating a robust 3D object recognition system, which is also able to work in real time, became one of the main goals towards for computer vision researchers [13]. There exist many reviews about 3D object recognition in the literature, including the seminal works of Besl and Rain [14], Brady et al. [15], Arman et al. [16], Campbell and Flynn [17], and Mamic and Bennamoun [18]. All of them perform a general review of the 3D object recognition problem with varying levels of detail and different points of view. The work of Guo et al. [12] is characterized by its comprehensive analysis of different local surface feature-based 3D object recognition methods which were published between the years 1992 and 2013. In that review, they explain the main advantages and drawbacks of each one of them. They also provide an in-depth survey of various techniques used in each phase of a 3D object recognition pipeline, from the keypoint extraction stage to the surface matching one, including the extraction of local surface descriptors. The review is specially remarkable due to its freshness and level of detail. It is important to remark that all the described methods make use of carefully designed feature descriptors by experts in the field.

1.3.3 Learning Features Automatically by Means of Deep Learning

From the earliest days of computer vision, the aim of researchers has been to replace hand-crafted feature descriptors, which require domain expertise and engineering skills, with multilayer networks able to learn them automatically by using a general-purpose training algorithm [19]. The solution for this problem was discovered during the 1970s and 1980s by different research groups independently [20][21][22]. This gave birth to a whole new branch of machine learning named deep learning.

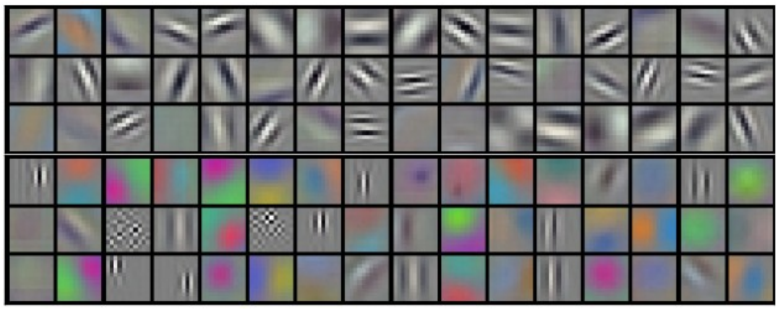


Figure 1.3: Filters learned by the network proposed by Krizhevsky et al. [23]. Each of the 96 filters shown is of size $11 \times 11 \times 3$. The filters have clearly learned to detect edges of various orientations and they resemble Gabor filters. Image analysis using that kind of filters is thought to be similar to perception in the human visual system [24].

Deep learning architectures usually consist of a multilayer stack of hierarchical learning modules which compute non-linear input-output mappings. Those modules are just functions of the input with a set of internal weights. The input of each layer in the stack is transformed, using the functions defined by the modules, to increase the selectivity and invariance of the representation. The backpropagation procedure is used to train those multilayer architectures by propagating gradients through all the modules. In the end, deep learning applications use feedforward neural network architectures which learn to map a fixed-size input, e.g., an image, to a fixed-size output, typically a vector containing a probability for each one of the possible categories [19].

Figure 1.3 shows some sample filter modules automatically learned by training one of the most successful deep learning architectures: the deep convolutional neural network proposed by Krizhevsky et al. [23] to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 [25] contest into 1000 different classes.

1.3.4 Convolutional Neural Networks for Image Analysis

In spite of the fact that these kind of architectures showed a huge potential for solving many computer vision problems, they were ignored by the computer vision community during the 1990s. The main reason for that was twofold. It was widely accepted that learning feature extractors with little prior knowledge was impossible. In addition, most of the researchers thought that using simple gradient descent techniques to train those networks would get them inevitably trapped in poor local minima.

In the latter years, certain breakthrough works revived the interest in deep learning architectures [19]. Recent studies proved that local minima are not an issue with large neural networks. Following a set of seminal works for the field on training deep learning networks [26][27], a group of researchers from the Canadian Institute for Advanced Research (CIFAR) introduced unsupervised learning procedures to create layers of feature detectors without labelled data, they also pre-trained several layers and added a final layer of output units; the system was tuned using backpropagation and achieved a remarkable performance when applied to the handwritten digit recognition or pedestrian detection problems [28]. In addition, the advent of GPUs, which were easily programmable and extremely efficient for parallel problems, made possible the training of huge networks in acceptable time spans [29].

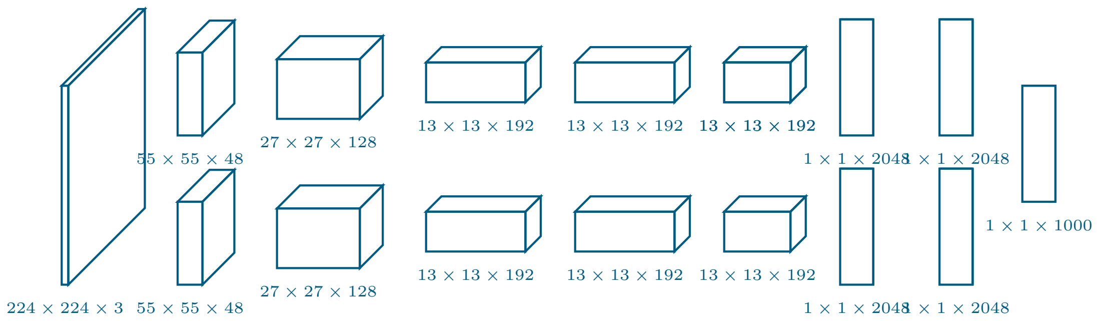


Figure 1.4: Illustration of the architecture of the aforementioned CNN proposed by Krizhevsky et al.[23] for the ImageNet challenge. Besides the normal components, e.g., convolutional, pooling, and fully connected layers, this network features two different paths. One GPU runs the layers at the top while the other runs the layers at the bottom.

All those contributions to the field led to the birth of probably the most important milestone regarding deep learning: the Convolutional Neural Network (CNN). This special kind of deep network was designed to process data in form of multiple arrays and gained popularity because of its many practical successes. This was due to the fact that they were easier to train and generalized far better than previous models. The architecture of a typical CNN is composed by many stages of convolutional layers followed by pooling ones and non-linearity Rectified Linear Unit (ReLU) filters; in the end, convolutional and fully connected layers are stacked. The key idea behind using this stack of layers is to exploit the property that many natural signals are compositional hierarchies, in which higher-level features are obtained by composing lower-level ones. Figure 1.4 shows a typical architecture of a CNN.

1.4 Proposal and Goals

In this work, we propose the implementation of a 3D object class recognition system using CNNs. That system will be accelerated using GPUs for both training and classification. The input of the system consists of segmented regions of point clouds which contain objects. Those point clouds will be synthetically simulated as if they were generated by low-cost RGB-Depth (RGB-D) sensors. To accomplish that proposal, we established a set of goals to be achieved:

- *Analyze the theoretical background of CNNs*. In order to implement our system, or extend existing frameworks to suit our needs, it is mandatory to understand the underlying principles of CNNs. For this purpose, we will review the core concepts and ideas that might be used during this work.
- *Generate volumetric representations for 3D data*. This includes analyzing the state of the art for successful representations, proposing a suitable and efficient representation for our kind of data, implementing the means to generate them, and also choosing a proper dataset.
- *Design, implement, and test a 3D CNN*. As an intermediate step, 2.5D CNNs will be studied as well. To achieve this goal, the state of the art of both 2.5D and 3D CNNs will be reviewed. We will also analyze existing machine learning frameworks to determine the one that best suits our needs. In addition, a set of experiments will be carried out to analyze the accuracy and performance of the proposed architectures.
- *Assemble and configure a server for training the system*. Given the significant computational needs of deep learning system during training, it becomes a matter of utmost importance to set up an appropriate server for experimenting with different configurations.

1.5 Outline

This document is structured as follows. Chapter 2 analyzes the theoretical background of CNNs. Chapter 3 describes the set of tools, data, and resources used in this work. Chapter 4 is devoted to volumetric representations for 3D data. Chapter 5 discusses and experiments with 2.5D and 3D CNNs for object recognition. At last, Chapter 6 draws conclusions, shows the publications derived from this Thesis, and proposes future works.

Chapter 2

Convolutional Neural Networks

This chapter provides a theoretical primer to Convolutional Neural Networks, describing the concepts and techniques that will be used in subsequent chapters of this document for implementing a 3DCNN. We will first introduce CNNs in Section 2.1. Next, in Section 2.2 we will describe what a convolution operator is. We will also show the typical architecture of a CNN in Section 2.3. We then discuss how to train a CNN in Section 2.4. At last, we draw some final remarks and conclusions in Section 2.5.

2.1 Introduction

Convolutional Neural Networks are models inspired by biological neural nets that replace dense matrix multiplications, employed by fully connected layers such as the ones of a Multi-Layer Perceptron (MLP), by convolutions in at least one layer [30].

More specifically, they are a kind of specialized neural network that takes advantage of data with clear topologies, e.g., images as 2D grids of pixels, as a side effect of that specialization they also turn out to scale reasonably well to large sized models. CNNs are arguably one of the most important examples of succesfully transferring knowledge and insights achieved by studying the brain to machine learning. Furthermore, they were pioneer models in deep learning due to their exceptional performance and also for being easily trained with back-propagation.

2.2 The Convolution Operator

From a mathematical point of view, a convolution is just an operation on two functions of a real-valued argument which can be expressed as follows

$$y(t) = (x * w)(t) = \int x(\tau)w(t-\tau)d\tau \,,$$

where $t \in \mathcal{R}$, $\tau \in \mathcal{R}$, $x : \mathcal{R} \rightarrow \mathcal{R}$, and $w : \mathcal{R} \rightarrow \mathcal{R}$. The resulting function $y : \mathcal{R} \rightarrow \mathcal{R}$ after applying the convolution operator, typically denoted with an asterisk $*$, to the functions x and w is defined as the integral of the product of both functions after one is reversed and shifted (τ). The first function x is usually referred to as the *input*, whilst w is a weighthing function known as *kernel*. The output y is named *feature map*.

When implementing a convolution operation in a computer, the inputs are discrete and so has to be the operation. The index t can only take integer values. Assuming that both the input and the kernel are defined only on t, a discrete convolution can be defined as

$$y(t) = (x * w)(t) = \sum_{\tau=-\infty}^{\tau=\infty} x(\tau)w(t-\tau)\,.$$

In practice, within the machine learning field, the input and the kernel are not real-valued functions but multidimensional arrays of data with discrete sizes for each dimension. Those arrays are called *tensors*. Taking all of this into account, the discrete convolution can be redefined as a finite summation over tensor elements. For instance, a 1D convolution is defined as

$$Y(i) = (X * W)(i) = \sum^{m} X(m)W(i-m)\,,$$

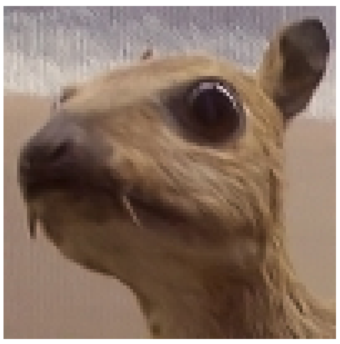
where Y, X, and W are one-dimensional tensors. Similarly, a 2D convolution over two-dimensional tensors can be written as

$$Y(i,j) = (X * W)(i,j) = \sum^{m}\sum^{n} X(m,n)W(i-m,j-n)\,.$$

Equivalent expressions can be easily derived for n-dimensional convolutions. It is important to remark that convolution is commutative, so the previous expression for 2D can be defined in a more straightforward way for a computer implementation named *cross-correlation*:

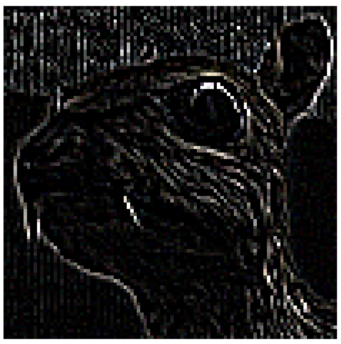
$$Y(i,j) = (X * W)(i,j) = \sum^{m}\sum^{n} X(i+m,j+n)W(m,n)\,.$$

Following up on the 2D case, whose input is presumably an image with a grid-like topolgy as a 2D array of pixels, a convolution can be seen as a feature detection stage in which the kernel acts as a filter. By convolving the kernel over the image, different feature activation values are obtained at each location in the image, thus generating a feature map as output. Figure 2.1 shows an example of a border detection filter.



(a) Original

$$\begin{bmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & 8 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{bmatrix}$$



(b) Filtered

Figure 2.1: Effect of a 5×5 edge detection kernel over an input image (a). The output feature map (b) shows activations in white and supressed locations in black.

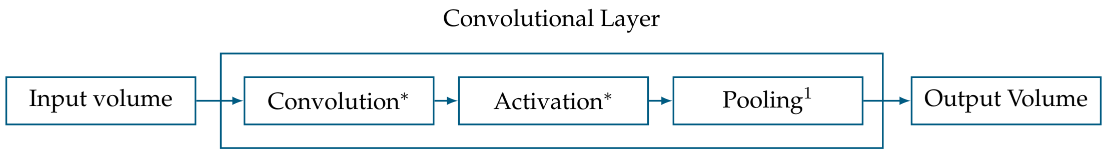


Figure 2.2: Basic components of a typical Convolutional Neural Network. A CNN is viewed as a stack of convolutional layers which are composed by a combination of convolution and activation (also named detector) layers followed by a single pooling layer. Following this scheme, an input volume is transformed to an output one which can be provided to the next layer of the network.

2.3 Architecture of a CNN

A CNN typically consists of a set of stacked layers that transform an input volume into an output one through differentiable functions. The three most common types of layers are the following ones: convolution, activation, and pooling. In fact, the combination of those three layers is the building block of a CNN and it is typically referred as a *convolutional layer* (see Figure 2.2). Apart from those basic layers, a CNN is followed by a set of fully connected layers for high-level reasoning. In the end, a loss layer specifies how the network penalizes the deviation between the predicted and true labels. Each layer type is discussed further below.

2.3.1 Convolution

The convolution layer embodies the core concept of a CNN: take into account the spatial structure of the input volumes. Typical fully connected neural network architectures treat every element of the input volume in the same way (see Figure 2.3), so the spatial structure must be inferred by learning. CNN use convolution layers instead of fully connected ones to process the input volumes and take advantage of their spatial structures. For this purpose, convolution layers feature three basic concepts: *local receptive fields*, *shared weights*, and *pooling* (which is described in detail in Section 2.3.3).

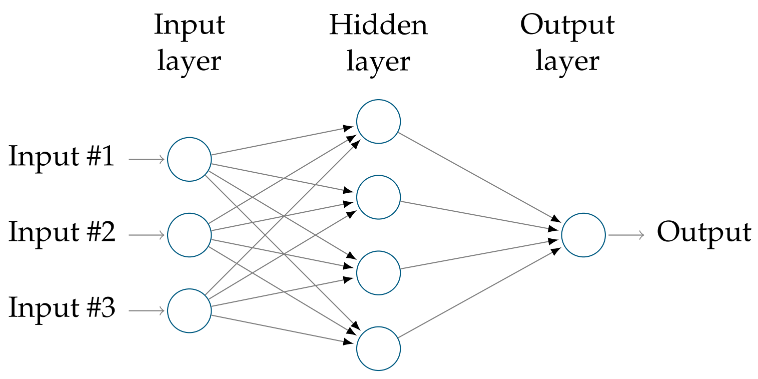


Figure 2.3: Fully connected architecture with three inputs and one hidden layer. As we can observe, each hidden neuron is connected with every neuron of the previous layer, thus no inherent spatial structure is taken into account.

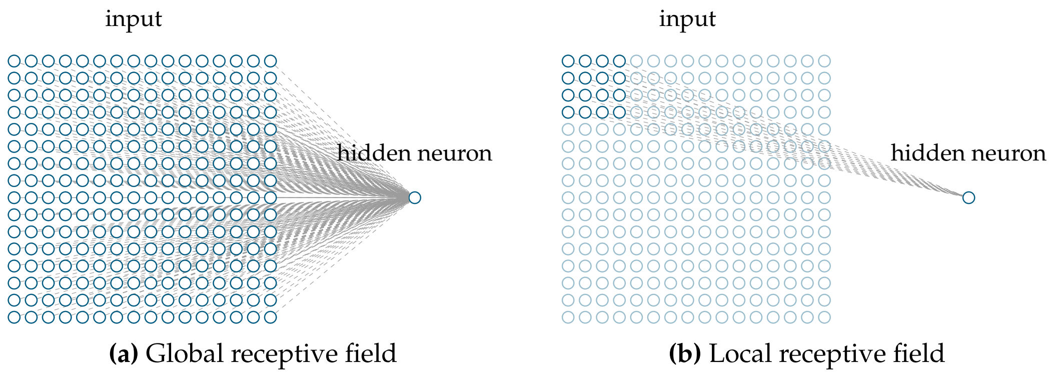


Figure 2.4: Global (a) and local (b) receptive fields for a hidden neuron over a 16×16 input. Each connection (in gray) learns a weight.

Local Receptive Fields

The concept of receptive field of a neuron has a strong biological inspiration. It is defined as the particular region of the sensory space in which a stimulus will trigger the firing of that neuron. From the computer vision point of view, the receptive field of a neuron is the input region which impacts the state of that neuron, i.e., those elements of the input which are connected to the neuron.

As we have previously stated, a hidden neuron of a fully connected architecture will be connected to every element coming from the previous layer thus creating a *global receptive field*. This fact implies a huge computational cost and immense memory requirements because of the vast amount of weights featured by the network (one per each connection). Besides, no spatial structure is enforced by this pattern. On the other hand, convolution layers do not connect the hidden neuron with every input element. Instead, they make connections in reduced and localized areas of the input named *local receptive fields*. Figure 2.4 shows a comparison of both global and local approaches.

Convolution layers use a local receptive field with a size that corresponds to the size of the convolution filter. The convolution process consists of sliding the receptive field across the entire input using a certain *stride*. For each local receptive field, there is a hidden neuron connected to it in the next layer. By doing this, the spatial structure of the input is taken into account. Figure 2.5 shows the sliding process.

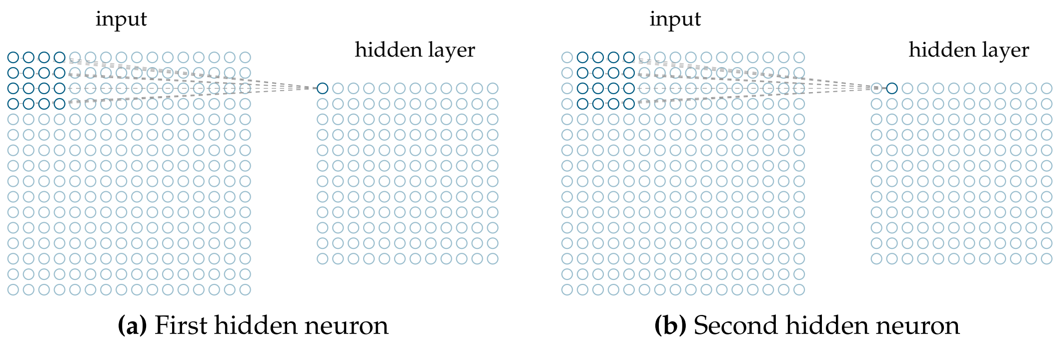


Figure 2.5: Sliding a local receptive field of 4×4 over an input of 16×16 elements. A hidden layer of 12×12 elements is produced by using a stride of 1×1 elements.

Shared Weights

Intuitively, and following the conventions of a fully connected scheme, each neuron of the hidden layer would have a bias, and each connection of the receptive field would learn a weight. CNNs change this connection scheme so that the same weights and bias are used for each neuron of the hidden layer. By doing this, the number of parameters is dramatically reduced. For the example shown in Figure 2.5, only a 4×4 filter and a single bias is required for the convolution instead of $4 \times 4 \times 12 \times 12$ weights and 12×12 biases if the parameters were not shared.

In this way, the output for the i, j neuron of the hidden layer H is

$$H_{i,j} = b + \sum_{k=0}^{f_w-1} \sum_{l=0}^{f_h-1} X_{i+k,j+l} W_{k,l} \, ,$$

where b is the bias, X is the input, and W is the $f_w \times f_h$ weight matrix or filter. This means that all neurons of the hidden layer will detect the same feature, since they are using the same filter, but each one will detect that very same feature in its particular receptive field. By sliding the receptive fields across the whole image, CNNs do not only take spatial structure into account but also adapt to *translation invariance*. In other words, a feature map is computed for the whole input so the feature to be detected does not need to be in a certain position of the input.

It is important to notice that the aforementioned examples show how to learn a single filter to detect a single kind of feature in the input. However, convolution layers usually learn many detectors to generate feature maps for various kinds of features as shown in Figure 2.6. This concept is called filter bank.

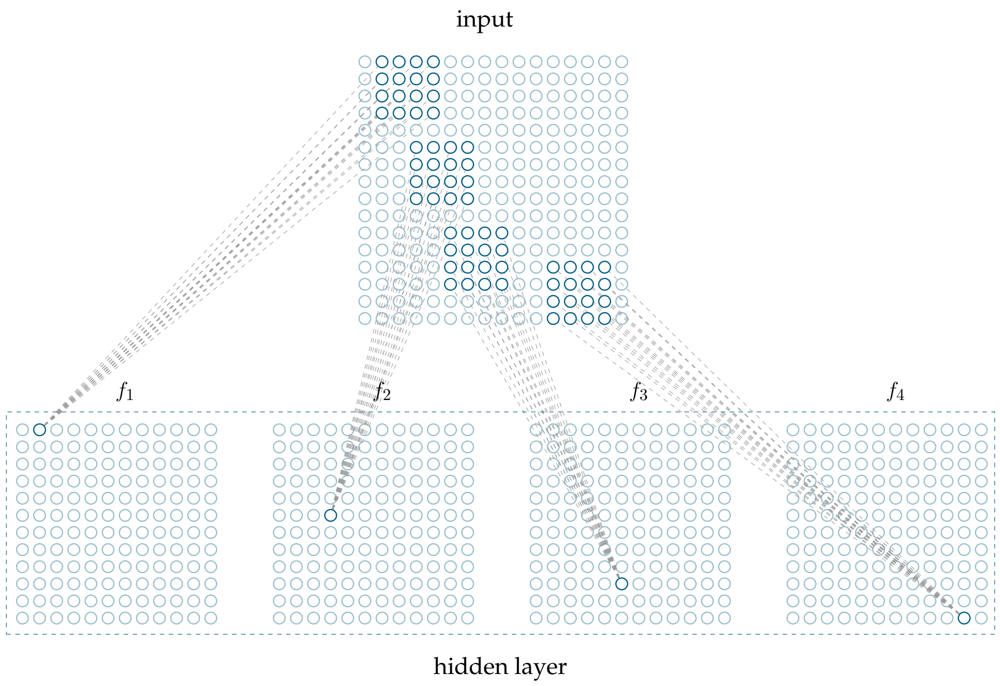


Figure 2.6: A convolution layer which generates 4 feature maps f whose sizes are 12×12. Each feature map is generated by a different 4×4 filter by convolving it over a 16×16 input (stride= 1).

2.3.2 Activation

For now, the output of the network will just be a linear transformation of the input. This fact implies that the network will not satisfy the *universal approximation theorem* [31]. In other words, the representational power of the network is constrained. To transform the network into a universal approximator, non-linearities must be introduced.

The purpose of the activation layers is to introduce non-linearities into the network. Some of the most common activation functions are shown in Figure 2.7. Those functions increase the non-linear properties of the decision function learned by the network, without affecting the receptive fields of the previous convolution layers.

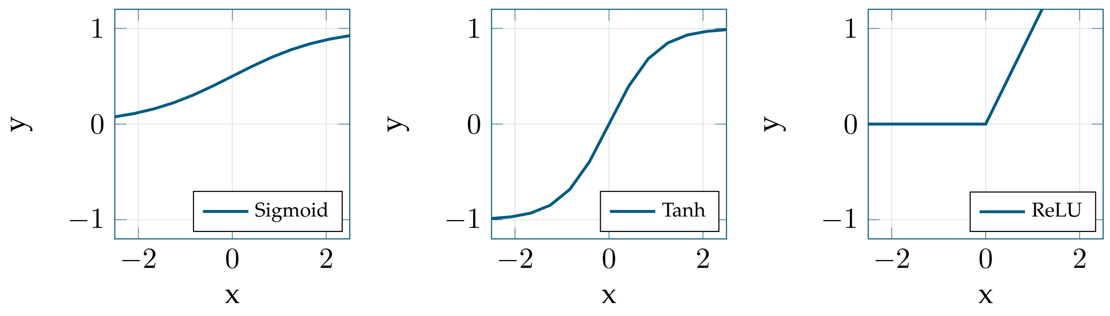
The sigmoid function has the mathematical form $\sigma(x) = 1/(1+e^{-x})$. The intuitive interpretation of this non-linearity is that the input gets squashed into the range $[0, 1]$. It has been widely used due to its biological interpretation as the firing rate of an actual neuron. However, it is not commonly used because of two major drawbacks that hinder or render impossible the learning process in some situations: gradient saturation and not zero-centered outputs.

The tanh non-linearity is similar to the sigmoid one, but it squashes the input to the range $[-1, 1]$. It can be expressed as a scaled sigmoid $tanh(x) = 2\sigma(2x) - 1$. In comparison with the sigmoid, it still saturates gradients but the output is zero-centered. In practice, the tanh has been used as a preferred replacement for the sigmoid.

The ReLU is the most common activation function for CNNs. It basically thresholds the input at zero $f(x) = max(0, x)$. It has two major advantages: gradient descent training is significantly accelerated due to its non-saturating properties and it features a low computational cost in comparison with the sigmoid and tanh which involve expensive operations. Nonetheless, ReLU units can irreversibly die (the neuron will never activate again) if the learning rate is not carefully set [32]. In order to solve this problem, other functions have been proposed, being the most popular ones the Leaky ReLU and the Parametric Rectified Linear Unit (PReLU) [33].

Activation layers are often placed right after a convolution takes place, but many combinations are possible. In this case, the output for the i, j neuron of a hidden layer H after the convolution takes place and the non-linearity, denoted as σ independently of the chosen activation function, is applied is:

$$H_{i,j} = \sigma(b + \sum_{k=0}^{f_w-1} \sum_{l=0}^{f_h-1} X_{i+k,j+l} W_{k,l}) \,.$$



(a) Sigmoid **(b)** Tanh **(c)** ReLU

Figure 2.7: Activation functions: Sigmoid (a), Tanh (b), and ReLU (c).

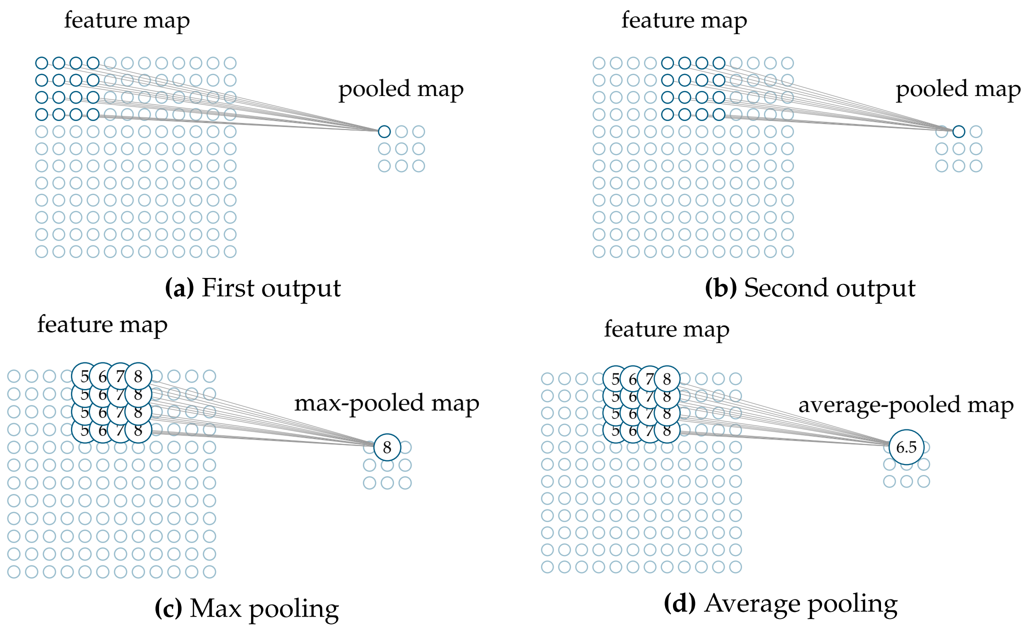
2.3.3 Pooling

In addition to the convolution and activation layers just described, CNNs also contain pooling layers placed immediately after a combination of convolution and activation ones. These layers are used to simplify the feature maps produced by the previous convolution and activation layers.

The intuitive concept supporting this kind of layer lies in the fact that once a feature has been found by the previous layers, its exact location in the feature map is not as important as its location relative to other detected features. Pooling layers take advantage of this reasoning to progressively reduce the spatial size of the representation. This simplification has many advantages: by reducing the representation size the amount of parameters to be learned is also reduced, therefore decreasing the computational cost of the whole network and also diminishing overfitting.

In detail, the pooling layer generates a condensed feature map by summarizing regions of a certain size with a predefined stride. Figures 2.8a and 2.8b show a pooling example in which a 12×12 feature map gets condensed into a 4×4 pooled map, produced by pooling regions of 4×4 neurons with a stride of 4×4. The pooling region is slided over the whole input to generate the pooled map. It is important to remark that convolution layers can generate many feature maps that are passed through activation layers to the pooling stage. In this case, pooling is applied to each feature map separately.

There are many pooling operations, being the most common the one known as *max pooling* (see Figure 2.8c). In this operation, each pooling region outputs the maximum value of the neurons for that region. Other functions such as average pooling (see Figure 2.8d) or L2-norm pooling can be used as well. Despite the fact that pooling layers have been commonly used for CNNs in almost every architecture, the current trend in research tends to reduce the size of the pooling regions or even remove the pooling layers due to the aggressive reduction of information [34][35].



(a) First output

(b) Second output

(c) Max pooling

(d) Average pooling

Figure 2.8: Pooling operation over a 12×12 feature map using a filter size of 4×4 and a stride of 4×4.

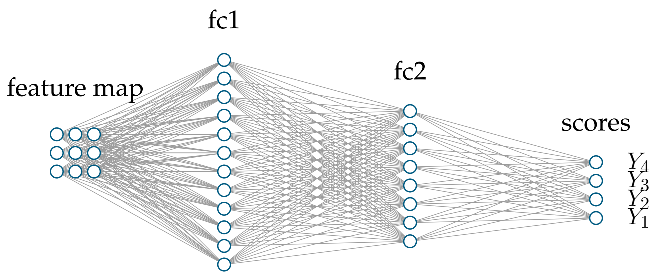


Figure 2.9: Three fully connected layers using a 3×3 feature map produced by a stack of convolution, activation, and pooling layers as input. Each connection has an associated weight that will be learned. The last fully connected layer represents the output of the network, a vector of four scores $[s_1, s_2, s_3, s_4]$ (also named logits) for a classification problem with four classes or labels.

2.3.4 Fully Connected

After a stack composed by convolution, activation, and pooling layers, the high-level reasoning of the network is deferred to a set of fully connected layers. Those layers are identical to the layers in a typical MLP. This means that neurons in a fully connected layer are connected to every neuron in the previous layer as seen in Figure 2.3.

The last fully connected layer is usually one-dimensional and contains as many neurons as classes or labels we are trying to classify. The output produced by this layer is a score per each class. Figure 2.9 shows an example of fully connected layers applied to a feature map generated by a stack of convolution, activation, and pooling layers.

2.3.5 Loss

In order to train the network – further details about the training process will be discussed in Section 2.4 – a way to penalize the deviation between the true labels of the instance that we are trying to classify and the label predicted by the network is needed. The most common way to do that is using a *softmax* function to transform the logits into probabilities and then apply a *cross-entropy* function to compute the *loss* which determines that penalty.

The softmax is a normalized exponential function that squashes the k-dimensional classifier output vector Y of real values to another k-dimensional vector $S(Y)$ of real values in the range $[0, 1]$ that add up to 1, i.e., it generates a probability distribution for the predicted classes. The softmax function is defined as:

$$S(Y_i) = e^{Y_i} / \sum_{j=0}^{k} e^{Y_j} \, .$$

The loss is then computed using the cross-entropy, a natural way to measure the difference between two probability vectors. Given a probability distribution $S(Y)$, and a one-hot encoded class label vector L in which $L_i = 1$ for the true label i for the sample and 0 otherwise, the difference can be expressed as $D(S(Y), L) = -\sum_{i=0}^{k} L_i log(S_i)$.

2.4 Training a CNN

Throughout the previous sections we introduced two key concepts in the context of CNNs: a score function which maps the inputs X to class scores Y using a set of weights W, and a loss function D which measures the quality of the current weights based on how well the predicted labels S agreed with the ground truth ones L in the training label. A third key concept arises intuitively: weights W which produce predictions for inputs X that are consistent with the training labels L have low loss D. Putting those concepts together, we can formulate training as the optimization process of finding the set of weights that minimize the loss function.

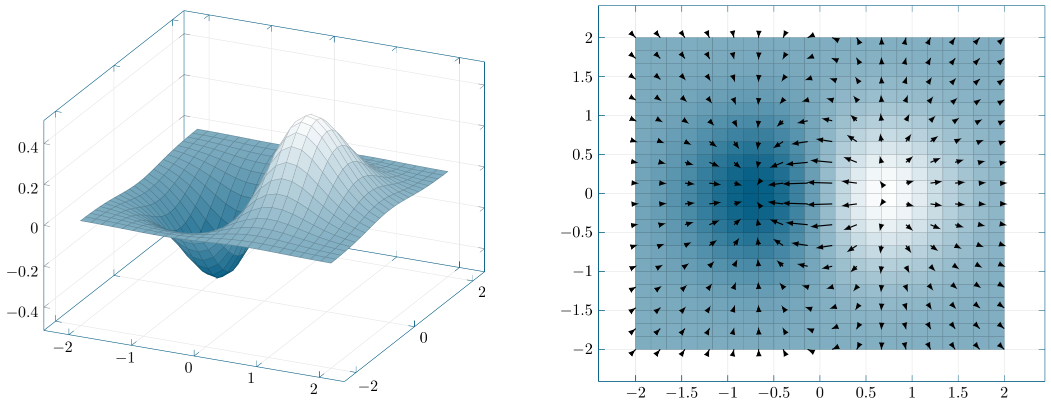
2.4.1 Gradient Descent

Gradient descent is an iterative optimization technique based on the assumption that if the function to be minimized f – which uses a set of parameters or weights W – is defined and differentiable in a neighborhood of a point x, that function decreases fastest taking a step from x in the direction of the negative gradient of x at x, $-\nabla f(x)$. Figure 2.10 shows the surface of a function and its negative gradients.

In the case of a CNN, for the sake of simplicity, we assume that the network, including all the convolution, activation, pooling, fully connected, and loss layers, models a function f which processes a training set X using a set of weights for the layers W and outputs the loss $D = f(W, X, L)$ for that input. This process is called the *forward pass*. Using that loss we can compute the gradients of each layer using backpropagation and then update the weights using those gradients of the loss functions in what is called the *backward pass*. The vanilla version of the weight updating scheme looks as follows:

$$W_{t+1} = W_t - \alpha \nabla f(W_t, X, L)\,,$$

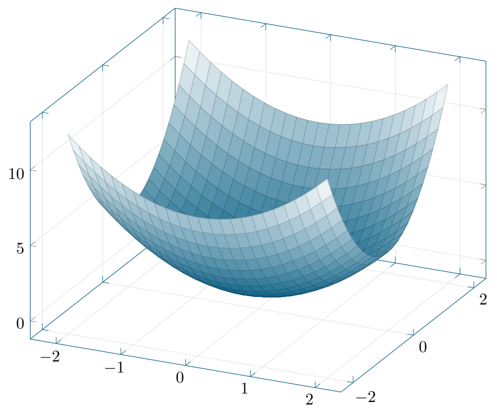
where W_t is the set of weights for iteration t and W_{t+1} is the updated one using the computed gradients and the current training example. The learning rate α determines the step size to take in the direction of the negative gradient of the loss function.



(a) Surface

(b) Gradients

Figure 2.10: Surface of the function $e^{x^2-y^2}x$ (a) – which has a minimum at $(-1/\sqrt{2}, 0)$ and a maximum at $(1/\sqrt{2}, 0)$ – together with the negative gradients shown in a top-down view (b).


(a) Surface

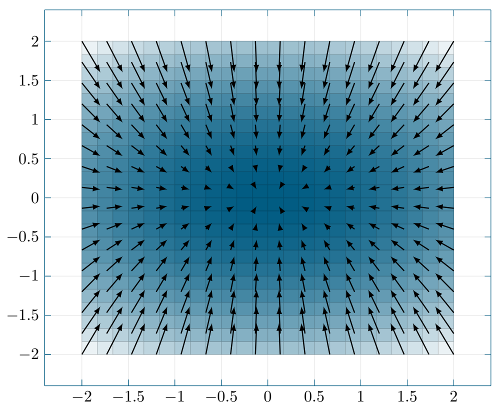
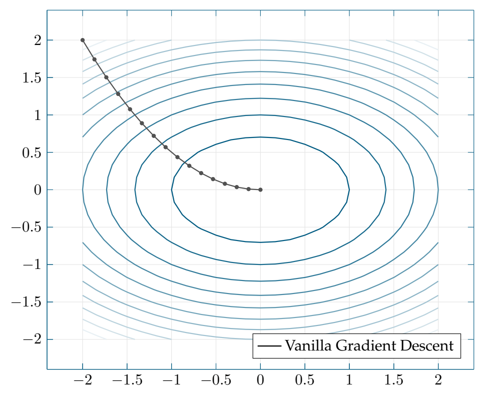

(b) Gradients

Figure 2.11: Surface of the function $x^2 + 2y^2$ (a)– which has a minimum at $(0, 0)$ – together with the negative gradients shown in a top-down view (b).

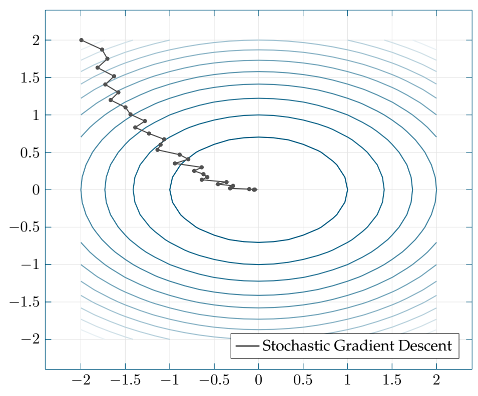
2.4.2 Stochastic Gradient Descent

Vanilla gradient descent needs to compute the gradients for the whole training set just to perform a single update or iteration. In other words, an epoch must be performed for computing the gradients. This poses a problem of utmost importance: gradient descent is intractable for big datasets that are often required for deep CNNs to avoid overfitting. Stochastic Gradient Descent (SGD) is an efficient alternative.

Instead of computing the loss and gradients for the whole training set, SGD computes an estimate of it using a mini-batch of n training examples per each iteration. Since we are only dealing with a small fraction of the training set for each step we take, we might not advance in the right direction to minimize the loss. In balance, many more steps might be needed for convergence but each one of them is much cheaper to compute. In the end, SGD is vastly more efficient than vanilla gradient descent for large datasets. When applied to the cost function whose surface is shown in Figure 2.11, the difference between both techniques can be observed in Figure 2.12.



(a) Vanilla



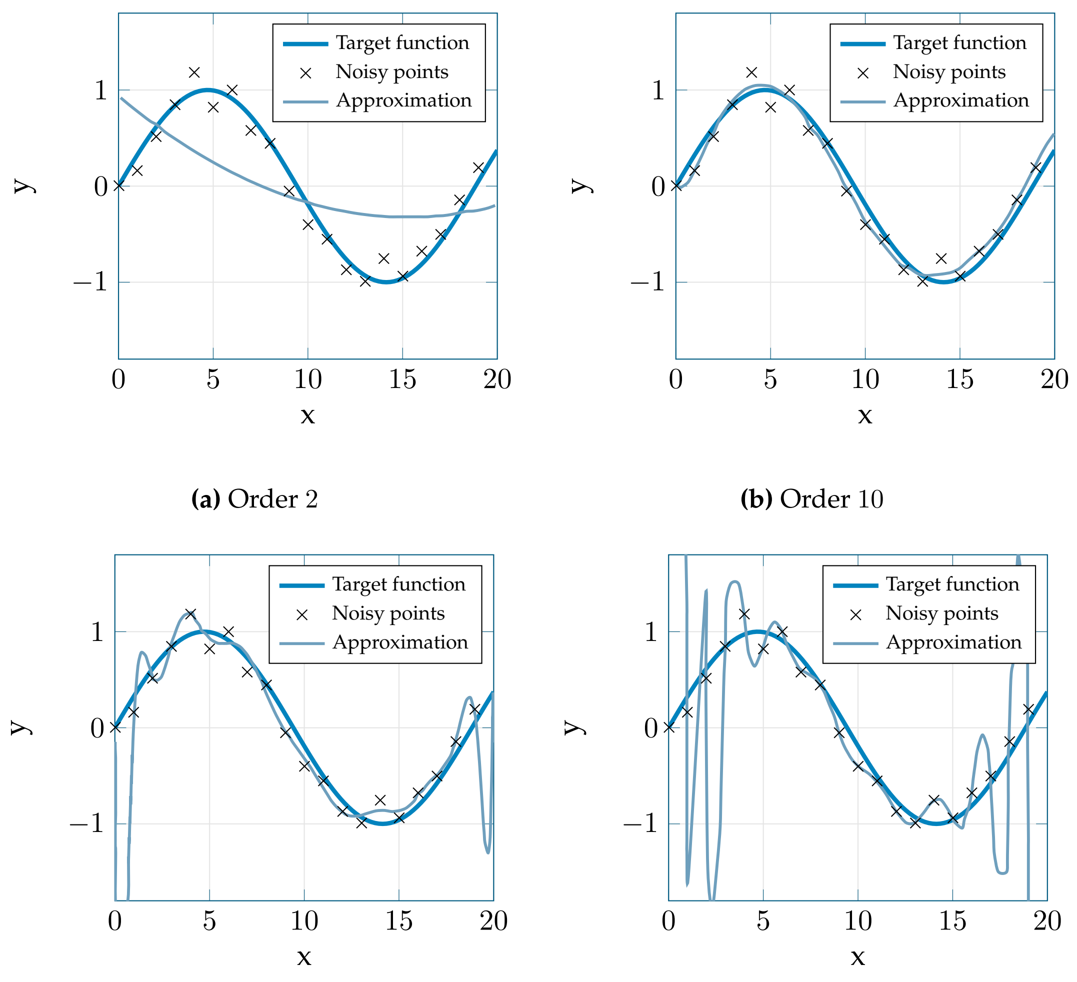
(b) Stochastic

Figure 2.12: Vanilla (a) and stochastic (b) gradient descent applied to the function $x^2 + 2y^2$. The contour plots show that SGD is less stable and takes more steps for convergence to a minimum.

2.4.3 Generalization and Overfitting

One of the most important aspects of machine learning models in general, and CNNs in particular, is how well they generalize to unseen data. When training a model, one must be very careful in order to avoid the situation in which the error in the training set is low but not that good in the test set. When this happens, the model is *overfitting* the training data, achieving poor generalization.

In order to illustrate the overfitting concept in a simple manner, we will reduce the problem to a polynomial approximation of a function. This experiment was replicated from the one originally carried out by Lawrence *et. al* [36]. We created a training dataset consisting of 21 noisy points uniformly sampled from the $f(x) = sin(x/3)$ function in the $[0, 20)$ domain. This dataset was then used to fit polynomial models with orders $2, 10, 16$, and 20. Figure 2.13 shows the target function, the noisy samples, and the approximations achieved by the polynomial model with increasing order.



(a) Order 2 **(b)** Order 10

(c) Order 16 **(d)** Order 20

Figure 2.13: Polynomial interpolation of noisy samples of the function $f(x) = sin(x/3)$ in the domain $[0, 20]$ and range $[-1, 1]$ as the order of the fitted model is evaluated at points $2, 10, 16$, and 20. The increasing complexity of the model tends to overfit the noisy data, thus not generalizing well for the actual function.

As we can observe, when the model has too few parameters (order 2) both training and generalization errors are high since it needs more expressiveness to fit the training data. With order 10 the model fits the training data reasonably good and also generalizes well. However, as the number of parameters increases significantly (orders 16 and 20) the model starts fitting the training data with almost no error, but it generalizes poorly, deviating too much from the target function.

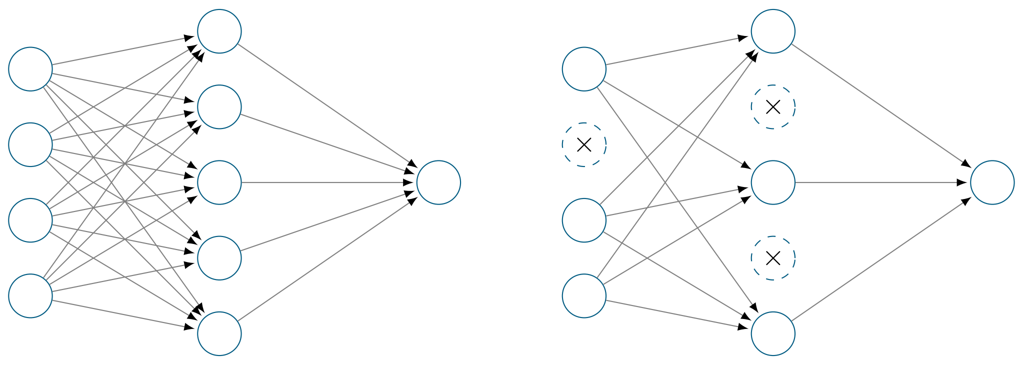
Deep neural networks such as CNNs are very expressive models thanks to their multiple non-linear hidden layers. At the same time, due to the large number of parameters, they are particularly prone to overfitting, especially with limited training data. In this regard, two critical topics arise: the selection of architectures which maximizes generalization – determining the optimal network layers, depth, and hyperparameters – and the development of techniques to prevent overfitting. The process of avoiding or preventing overfitting is called *regularization* and some of the most successful techniques include $L1$ and $L2$ regularization, maximum norm constraints, dropout, and early stopping.

Maximum Norm Constraints

One of the most simple forms of regularization consists of enforcing an absolute upper bound on the magnitude of every weight vector. This is implemented by clamping each weight vector W to satisfy $||W||_2 < c$ where c is the constraint value, which is usually set to 3 or 4. By doing this, the weight vectors cannot explode to high values even with high learning rates, thus preventing overfitting.

Dropout

Dropout [37] is arguably the most applied technique to avoid overfitting in deep neural networks. In the particular case of CNNs, dropout operations are performed between fully connected layers. The key idea is to randomly drop units and connections between those fully connected layers during training with a certain probability. The goal is to prevent units from co-adapting. In the end, dropout layers significantly reduce overfitting. Due to its simplicity and results, it has become one of the most common regularization methods. Figure 2.14 shows an example of this technique.



(a) Standard Net **(b)** After Dropout

Figure 2.14: Dropout model applied to a standard fully connected network (a), the crossed units have been dropped out randomly together with their connections (b).

$L1$, $L2$, and Elastic Net Regularization

Another common way to avoid overfitting is to add penalties for network parameters to restrict complexity as maximum norm constraints do. This process is also known as *weight decay*, and it is applied on a per layer basis.

In the case of the $L1$ regularizer, the $L1$ norm of each set of parameters is computed during the forward pass $||W||$ and $-\lambda||W||$ is added to the loss function. On the other hand, $L2$ computes the $L2$ norm $||W||^2$ during the forward pass and adds the term $-0.5\lambda||W||^2$ to the loss function. In both cases λ is a regularization coefficient that determines how dominant the regularization term is, i.e., the magnitude of the penalization. Intuitively, $L1$ makes the weight vectors become sparse during optimization, whilst $L2$ heavily penalizes peaky weight vectors over diffuse ones. It is important to notice that during the weights update, $L2$ regularization ultimately decays them linearly by adding the term $-\lambda W$ to the update equation.

In addition, both approaches can be combined by adding both $L1$ and $L2$ terms to the loss function $-(\lambda_1||W|| + \lambda_2||W||^2)$ into what is called *elastic net regularization* [38].

Early Stopping

It is a good practice to always monitor the error during training not only on the training set but also on a validation set. In this regard, the training process should be stopped, applying certain thresholds, when either the validation error is not improving significantly or either it starts to increase. This technique, which is illustrated in Figure 2.15, is known as *early stopping*.

Shuffling

In order not to bias the optimization algorithm during the descent, it is important to avoid providing the training batchs in a meaningful order. For this purpose, a common approach consists of *shuffling* the whole training set after every epoch.

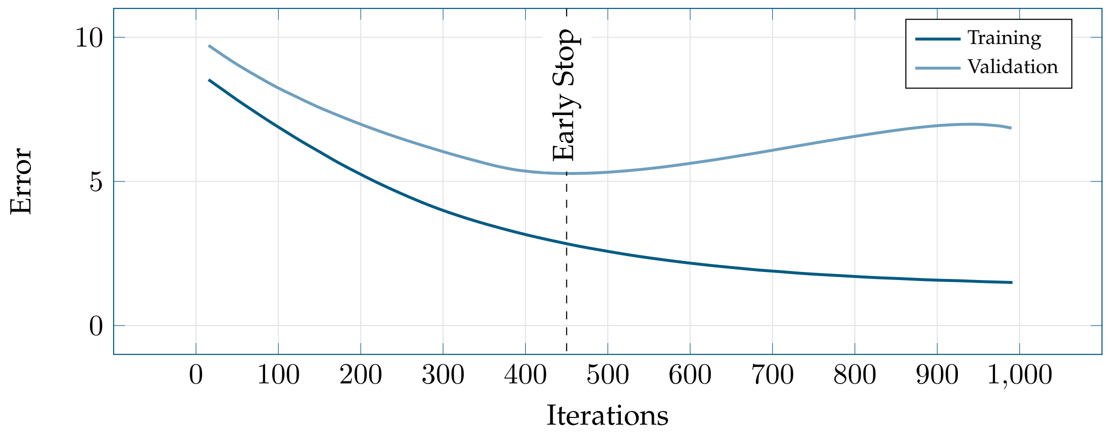
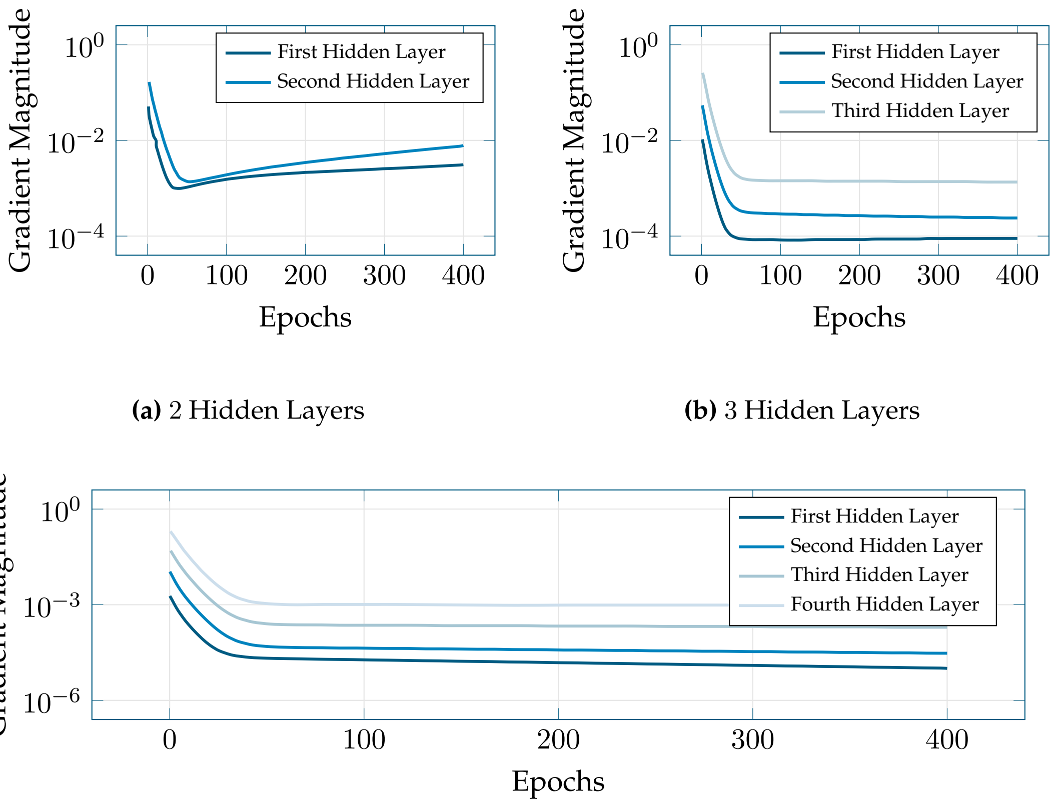


Figure 2.15: Representation of the early stopping regularization method. Both training and validation errors are monitored. At some point – near iteration 450 – validation error starts increasing while training error keeps decreasing. At that point, training must be stopped to avoid overfitting.

2.4.4 Vanishing and Exploding Gradient

Neural networks in general, and CNNs in particular are usually trained using gradient-based methods and backpropagation. Since CNNs have multiple layers, each set of weights is updated proportionally to the gradient of the loss function with respect to the current weight in each training iteration. For this purpose, backpropagation computes the gradients using the chain rule, from the last layer to the first. Between the different layers we can find activation ones such as sigmoid or tanh, whose gradients are in the range $(-1, 1)$ or $[0, 1)$. Since the chain rule multiplies those small numbers to propagate the gradient to the first layers, it decreases exponentially, rendering impossible learning features in early layers because the learning signal that reaches them is too weak [39]. Figure 2.16 illustrates the vanishing gradients problem using a multi-layer network with an increasing number of hidden layers. On the contrary, if the activation function derivatives do not saturate or can take larger values, the gradient can explode.

The vanishing gradient problem can be solved by using ReLU activation functions which do not saturate, layer-by-layer pretraining so that early layers only need to be adjusted slightly [40], and also the appearance of faster hardware like GPUs made backpropagation feasible even with weak signals. The exploding gradient can be alleviated normalizing the output of each layer so that it cannot saturate higher ones [41]. Furthermore, much effort has been devoted to initializing the weights so that they are less likely to lead to either vanishing or exploding gradients [42].



(a) 2 Hidden Layers **(b)** 3 Hidden Layers

(c) 4 Hidden Layers

Figure 2.16: Vanishing gradient in a multi-layer neural network training with the MNIST dataset. Experiments were replicated from [43].

2.4.5 Challenges and Optimizations

SGD – also known as mini-batch gradient descent – has become the algorithm of choice for training CNNs due to its efficiency and continuous success. However, there are many challenges that are yet to be solved regarding the optimization process:

- *Suboptimal local minima.* Gradient descent guarantees converging to a global minimum for convex optimization, but only to local minimums for non-convex problems such as the cost functions from a CNN. In addition, it has been identified that the main problem are saddle points [44], which are notoriously hard for SGD to escape, due to the almost zero gradient in all dimensions.
- *Learning rate value.* It is hard to choose the right initial value for the learning rate parameter. A small value will lead to painfully slow convergence, while a large one might cause overshooting and hinder convergence.
- *Learning rate schedules.* It is not clear what the step size should be, but there is a consensus about decreasing it as the training progresses [45]. Lowering the learning rate over time is known as learning rate decay. However, most of the schedules are defined in advance so they are unable to adapt to the training set.
- *Adaptive learning rate.* The learning rate is applied to all parameters with the same value. In some situations, certain features may be sparse and better results could be achieved if larger updates are performed for rarely occurring features and vice versa.

In this section we will review some of the most common optimization algorithms to address the aforementioned challenges: SGD with momentum, Nesterov accelerated Gradient (NAG), Adagrad, Adadelta, RMSprop, and Adaptive Moment Estimation.

SGD with Momentum

Momentum [46] is a method which aims to accelerate the convergence of SGD by keeping a running average of the gradient over time, and using it instead of the direction of the current mini-batch to perform the descent step. The weights are updated using the momentum term M_t as follows:

$$W_{t+1} = W_t - M_t \,.$$

The momentum term for the current iteration t is computed by adding a fraction γ of the update vector of the last step M_{t-1} to the current update vector

$$M_t = \gamma M_{t-1} + \alpha \nabla f(W_t, X, L) \,.$$

Essentially, the momentum term increases for dimensions whose gradients point consistently to the same directions over time and decreases otherwise, thus accelerating convergence and reducing oscillations.

Nesterov accelerated Gradient

NAG [47] is an improvement over momentum. Using the running average of the previous updates accelerates the descent blindly following the slope. This blind descent might get to a region in which the gradient slopes up again. NAG looks ahead by calculating the gradient with regard to the approximate future position of the weights:

$$M_t = \gamma M_{t-1} + \alpha \nabla f(W_t - \gamma M_{t-1}, X, L)\,.$$

Adagrad

Adagrad [48] is another optimizer that improves vanilla SGD by adapting the learning rate according to the weights, performing smaller updates for frequent features and larger ones and vice versa. This eliminates the need to manually tune the learning rate. It uses a different learning rate for each weight W_i at a certain time step t. The weights update rule is then expressed as follows:

$$W_{t+1,i} = W_{t,i} - \frac{\alpha}{\sqrt{G_{t,i} + \epsilon}} \nabla f(W_{t,i}, X, L)\,,$$

where $G_{t,i} \in \mathcal{R}^{d \times d}$ is a diagonal matrix where each diagonal element is the sum of the squares of the gradients with regard to W_i up to time step t. A smoothing term ϵ is introduced to avoid divisions by zero. This rule can be vectorized performing an element-wise matrix-vector multiplication:

$$W_{t+1} = W_t - \frac{\alpha}{\sqrt{G_t + \epsilon}} \odot \nabla f(W_t, X, L)\,.$$

Adadelta

The main problem of Adagrad is the continuous growing of the accumulated sum in the denominator which eventually causes the learning rate to become infinitesimally small thus rendering the algorithm unable to learn. Adadelta [49] seeks to overcome this weakness by restricting the window of accumulated past gradients to a fixed size w. That sum of gradients $E[g^2]_t$ is recursively defined as a decaying average of past gradients, and depends on the previous average and the current gradient:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) \nabla f^2(W_t, X, L)\,.$$

The weight update rule can be reformulated by simply replacing the diagonal matrix G_t with the decaying average over past squared gradients $E[g^2]_t$ (which is just the Root Mean Squared (RMS) error) as follows:

$$W_{t+1} = W_t - \frac{\alpha}{\sqrt{E[g^2]_t + \epsilon}} \odot \nabla f(W_t, X, L) = W_t - \frac{\alpha}{RMS[g]_t} \odot \nabla f(W_t, X, L)\,.$$

RMSprop [50] is another adaptive learning method which was developed independently and around the same time as Adadelta, both aiming to solve Adagrad's diminishing learning rate to infinitesimal quantities.

Adam

Adaptive Moment Estimation (Adam) [51] is an improvement over RMSprop and Adadelta that also keeps an exponentially decaying average of past gradients M'_t, apart from the exponentially decaying average of the past squared gradients M_t:

$$M'_t = \gamma' M'_{t-1} + (1-\gamma')\nabla f(W_t, X, L)\,,$$

$$M_t = \gamma M_{t-1} + (1-\gamma)\nabla f^2(W_t, X, L)\,.$$

The estimates of the first and second moment of the gradients, M_t and M'_t respectively, are biased towards zero so corrected versions are necessary:

$$\hat{M}'_t = \frac{M'_t}{1-\gamma'_t}\,, \quad \hat{M}_t = \frac{M_t}{1-\gamma_t}\,.$$

Those moments are used to update the weights using the Adam update rule:

$$W_{t+1} = W_t - \frac{\alpha}{\sqrt{\hat{M}_t + \epsilon}}\hat{M}'_t\,.$$

Comparison of Optimizers

Figure 2.17 shows a comparison of some of the most common optimizers explained above. This comparison evaluates the performance of those optimizers against each other on the MNIST dataset using *ConvNetJS*. The benchmark uses a simple two fully connected hiddden layers with 20 neurons each one and ReLU activation functions followed up by a softmax output layer. A batch size of 8 samples was used, the learning rate starts at 0.01, an $L2$ weight decay of 0.001 was also included, momentum was set to 0.9. More details about the setup can be consulted here [52].

As we can observe, Adadelta is the one with the best performance, i.e., faster convergence and the lowest loss. Adagrad converges fast during the first samples but then its performance starts degrading while NAG's behavior is the opposite. On the other hand, SGD is the slowest one whilst SGD with momentum is a tad faster and they both converge to similar final values as NAG.

However, it is important to notice that results and performance might vary significantly from one dataset to another and even different architectures may affect them. Also, hyperparameter tuning is mandatory for those solvers which depend on them in order to achieve good performance. As an empirical established rule, Adagrad and Adadelta are safer in the sense that they do not depend strongly on the hyperparameters. Nevertheless, well-tuned SGD with momentum or even vanilla SGD usually converge faster and achieve better results [52].

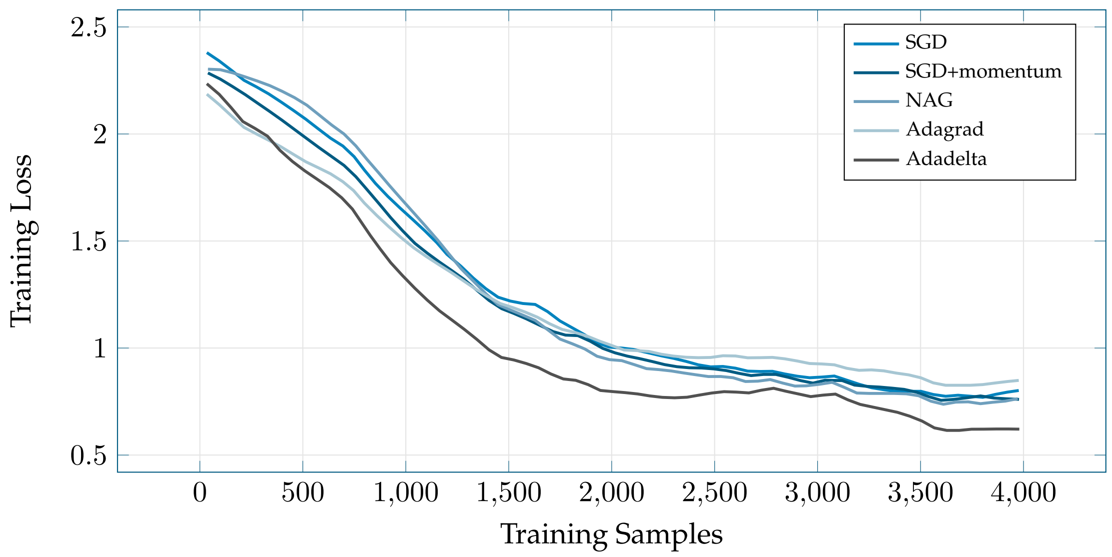


Figure 2.17: Comparison of common optimizers (SGD, SGD with momentum, NAG, Adagrad, and Adadelta) on the MNIST dataset using a network architecture with to fully connected hidden layers with 20 neurons followed by a softmax output. A batch size of 8 samples was used, the learning rate starts at 0.01, an $L2$ weight decay of 0.001 was also included, momentum was set to 0.9. The benchmark was implemented using *ConvNetJS* by Andrej Karpathy [52]. The plot compares the training loss as the number of training examples provided to the network increases.

2.5 Conclusion

In this chapter we have laid down the foundations for the rest of this work. First, we described what the convolution operator is and its formulation when applied to tensors. We also discussed the architecture of a CNN: its key concepts and layers. At last, we showed how to train this kind of neural network and we also enumerated challenges and how recent contributions to the field optimize this process.

This theoretical background provides us with all the concepts that we need to understand in order to design, implement, and train a 2DCNN. The following chapters will focus on the implementation of a 2.5DCNN – using well established deep learning frameworks to express the aforementioned concepts without needing to implement them from scratch – that will be later modified to perform 3D convolutions. However, before getting straight to the proposal of this work, we will have to review some prerequisites: materials and methods, and volumetric representations.

In that regard, the next chapter is devoted to the description of the different tools, data, and resources that will be used throughout this work to compute representations, train networks, and in general carry out any experiment. After that, we will discuss volumetric representations. As we previously stated in the beginning of this chapter, CNNs exploit data with clear topologies, i.e., grid-like structures or arrays. Raw 3D data exhibits no clear topology, e.g., unstructured point clouds provided by 3D acquisition devices. A transformation is needed to generate structured 3D tensors that can be fed to a CNN. Those matters will be discussed in the following chapters.

Chapter 3

Materials and Methods

In this second chapter we will describe the set of materials and methods employed in this work, in terms of software, data, and hardware. The chapter is organized as follows. Section 3.1 introduces the purpose of this chapter and puts the materials and methods in context. Section 3.2 reviews a set of popular deep learning frameworks and justifies our chosen one. Section 3.3 analyzes existing object databases and selects one to be used for this work. At last, Section 3.4 describes the hardware setup that was assembled and configured to perform the experiments.

3.1 Introduction

In order to carry out this project, we need the support of many different resources. First, we need tools for expressing CNNs, which would allow us to design, implement, train, and test different layers, architectures, and configurations. It is possible to implement them from scratch, but leveraging to a framework eases and streamlines the development process.

Second, we need a source of data to generate training, test, and validation sets. That data will be adapted and transformed into a volumetric representation that will be eventually provided as input to the CNNs. The choice of a dataset is not trivial due to the special needs of deep architectures.

Third, we require computational resources to experiment with the implemented CNNs to measure their performance in terms of both execution time and accuracy. Again, because of the special requirements of deep neural networks, low-end computers are not a good option so high-end solutions with specialized hardware is a must for maintaining a competitive edge in this field.

In the following sections we will review each need in detail, focusing on picking a framework that best suits our needs, an adequate dataset for our purposes, and configuring a server to provide proper computational power.

3.2 Frameworks

Deep learning architectures have significantly influenced various application domains, surpassing by a large margin state-of-the-art methods for computer vision [23], [53], natural language processing [54]–[56], and speech recognition [57]–[59]. Due to that significant performance improvement achieved compared to traditional approaches, the popularity of deep learning methods has significantly increased over the last few years. Because of that, deep learning frameworks have also experienced a considerable growth in order to enable researchers to efficiently design, implement, and deploy these kind of architectures. Although this meteoric rise has hugely benefited the deep learning community, it also leads researchers to a certain confusion and difficulties to

Framework	License	Core	Interface	OpenMP	CUDA
Torch[60]	BSD 3-Clause	C, Lua	Torch, C, C++/OpenCL	Yes	Third Party
Theano[61]	BSD 3-Clause	Python	Python	Yes	Yes
Caffe[62]	BSD 2-Clause	C++, Python	C++, CLI, Python, MATLAB	No	Yes
CNTK[63]	MIT	C++	CLI	No	Yes
TensorFlow[64]	Apache 2.0	C++, Python	C/C++, Python	No	Yes

Table 3.1: Comparison of the most popular deep learning frameworks, taking into account high-level features such as the source code license, the programming language used to write the core of the framework, the available interfaces to train, test, and deploy the models, and whether or not they support OpenMP and CUDA.

decide which framework to use. The fact is that different frameworks aim to optimize distinct aspects of the development process of deep learning architecture, i.e., choosing one framework over the others is mainly a matter of adequateness to the task at hand.

The list of available frameworks is vast and it is growing constantly as new frameworks are developed either to fill the gaps left by other's weaknesses or to provide new approaches and tools for researches to ease the development of deep architectures. Arguably, the most popular frameworks nowadays are Theano, Torch, Caffe, Computational Network Toolkit (CNTK), and TensorFlow. The list is not limited to those frameworks and also includes other less known ones such as DeepLearning4J, Neon, PyLearn, MXNet, ConvNet, CUDA-ConvNet, Deepmat, and more too numerous to mention. This section presents an overview of the five aforementioned most popular frameworks, as well as a brief comparison and discussion about their features to determine the best one for our purposes. In this regard, Table 3.1 shows a quick comparison of the five discussed frameworks, taking into account certain high-level features. This brief comparison serves as a proper starting point to describe each one of the frameworks in depth throughout the next sections.

3.2.1 Torch

Torch[60] is a scientific computing framework mainly maintained by researchers at Facebook Artificial Intelligence Research (FAIR) lab, Google DeepMind, Twitter, and also by a large list of contributors on *GitHub*[1]. It is a BSD-licensed C/Lua library that runs on Lua Just In Time (JIT) compiler with wide support for machine learning algorithms. Its goal is to have maximum flexibility and speed building scientific algorithms while making the process extremely simple thanks to an easy and fast scripting language, LuaJIT, and an underlying C/CUDA implementation. One of its main highlights is its large ecosystem of community-driven packages and documentation.

The most important core features are easiness (it uses a simple scripting language as LuaJIT as an interface to the C core routines), efficiency (the underlying C/CUDA implementation provides a strong backend for both CPU and GPU computations with NVIDIA CUDA Deep Neural Network (cuDNN)[65]), optimized routines (for linear algebra, numeric optimization, and machine learning), and fast embedding with ports to iOS, Android and Field Programmable Gate Array (FPGA) backends.

[1]http://github.com/torch/torch7

3.2.2 Theano

Theano[61], [66], [67] is a mathematical compiler for Python developed by a group of researches from the University of Montreal Computer Science and Operations (CSO) department, and maintained by them together with the contributions of a number users on *GitHub*[2]. It is a BSD-licensed framework in the Python programming language which can be used to define mathematical functions in a symbolic way, derive gradient expressions, and compile those expressions into executable functions for efficient performance in both CPU and GPU platforms. It was conceived as a general mathematical tool, but its main goal was to facilitate research in deep learning.

The main features of Theano are the following ones: powerful tools for manipulating and optimizing symbolic mathematical expressions, fast to write and execute thanks to its dependency on NumPy[68] and SciPy[69], CUDA code generators for fast GPU execution of compiled mathematical expressions, and code stability and community support due to its short stable release cycle and exhaustive testing.

3.2.3 Caffe

Caffe[62] is a deep learning framework developed and maintained by the Berkeley Vision and Learning Center (BVLC) and an active community of contributors on *GitHub*[3]. This BSD-licensed C++ library provides means to create, train, and deploy general-purpose CNNs and other deep models efficiently, mainly thanks to its drop-in integration of NVIDIA cuDNN[65] to take advantage of GPU acceleration.

The highlights of this framework are modularity (allowing easy extension by defining new data formats, layers, and functions), separation of representation and implementation (model definitions are written as configuration files in *Protocol Buffer* language), test coverage (all modules have tests which have to be passed to be accepted into the project), Python/MATLAB bindings (for rapid prototyping, besides the Command Line Interface (CLI) and C++ API), and pre-trained reference models.

3.2.4 Computational Network Toolkit

The CNTK [63] is a general purpose machine learning framework developed by Microsoft Research and recently open-sourced under the MIT license on *GitHub*[4]. Its main contribution is providing a unified framework for describing learning systems as Computational Network (CN), and the means for training and evaluating that series of computational steps. Despite the fact that it was created as a general toolkit, it mainly focuses on deep neural networks, so it can be redefined as a deep learning tool that balances efficiency, performance, and flexibility. The core is written in C++ and the CN models can be described in both C++ or Network Definition Language (NDL)/Model Editing Language (MEL).

Thanks to the directed graph representation of the computation steps, it allows to easily recreate common models and extend them or create new ones from scratch by combining simple functions in arbitrary ways. It features a significantly modularized architecture, SGD for training, auto-differentiation to obtain the gradient derivatives, and parallelization focused on GPU clusters across multiple nodes with CUDA-capable devices.

[2]`https://github.com/Theano/Theano`
[3]`http://github.com/BVLC/caffe`
[4]`https://github.com/Microsoft/CNK`

3.2.5 TensorFlow

TensorFlow[64] is a library for numerical computation using data flow graphs originally developed by researchers and engineers working on the Google Brain team for the purposes of conducting machine learning and deep neural networks research, currently an active community of *GitHub* users contribute to the source code as well using its repository [5]. Its core is written in C++ and the whole framework is being open sourced under the Apache 2.0 license. The main goal of this project is to provide an interface, in both Python and C++, for expressing artificial intelligence or machine learning algorithms, together with an efficient implementation of those algorithms which are flexible and scalable, i.e., the computations can be executed on a wide variety of devices with minimal or no changes at all.

This library features deep flexibility (every computation that can be expressed as a data flow graph can be implemented using TensorFlow), portability and scalability (training, testing, and deployment run on both CPUs and GPUs and they scale from single nodes to large-scale systems, besides it allows users to freely assign compute elements of the graph to different devices), research and production (deploying experimental systems is an easy task and requires minimal changes), and auto-differentiation (the derivatives needed for the data flow graph when expressing gradient based machine learning algorithms are computed automatically).

3.2.6 Other Frameworks

Other less known but still important frameworks that are currently being use by the research community are (among others) Keras [6], DeepLearning4j [7], Marvin [8], and MatConvNet [70] [9].

3.2.7 Picking a Framework

Picking the right framework depends on many factors, e.g., speed, extensibility, main language, the community, etc. In our case, our main concern is speed since the system is intended to be eventually deployed in an embedded platform for real-time recognition. Related to this point, it is important for us the compatibility of the framework to export and load models in embedded boards for classification. Another major concern is extensibility to create our own data layers and extend convolutions from 2D to 3D. And last, but not least, having an active community is a must for choosing a framework in order to effectively deal with problems and improve the current code base.

Taking all of this into account, we picked Caffe as our framework of choice due to its long standing community, extensibility, speed – thanks to C++ and CUDA –, and its well-tested deployment in embedded platforms. The framework was extended this framework to support 3D convolutions and pooling. In addition, we also added a data layer to load our volumetric representations that will be discussed in Chapter 4.

[5] `https://github.com/tensorflow/tensorflow`
[6] `https://github.com/fchollet/keras`
[7] `https://github.com/deeplearning4j/deeplearning4j`
[8] `https://github.com/PrincetonVision/marvin`
[9] `https://github.com/vlfeat/matconvnet`

3.3 Datasets

Deep neural network architectures are usually composed by many layers which in turn mean many weights to be learned. Because of that, there is a strong need of large-scale datasets to train those networks in order to avoid overfitting the model to the input data. Nowadays, large-scale databases of real-world 3D objects are scarce, some of them do not have that high number of objects [71][72][73], or were incomplete by the time this work was performed [74]. A possible workaround to this problem consists of using Computer Aided Design (CAD) model databases – which are virtually unlimited – and processing those models to simulate real-world data.

The *Princeton ModelNet* project is one of the most popular large-scale 3D object dataset. Its goal, as their authors state, is to provide researchers with a comprehensive clean collection of 3D CAD models for objects, which were obtained via online search engines. Employees from the Amazon Mechanical Turk (AMT) service were hired to classify over 150 000 models into 662 different categories.

At the moment, there are two versions of this dataset publicly available for download[10]: *ModelNet-10* and *ModelNet-40*. Those are subsets of the original dataset which only provide the 10 and 40 most popular object categories respectively. These subsets are specially clean versions of the complete dataset.

On the one hand, ModelNet-10 is composed of a collection of over 5000 models classified into 10 categories and divided into training and test splits. In addition, the orientation of all CAD models of the dataset was manually aligned. On the other hand, ModelNet-40 features over 9800 models classified into 40 categories, also including training and test sets. However, the orientations of its models are not aligned as they are in ModelNet-10. Figure 3.1 and Table 3.2 show the model distribution per each class of both subsets taking into account the training and test splits. Figure 3.2 shows some model examples from ModelNet-10.

[10]http://modelnet.cs.princeton.edu/

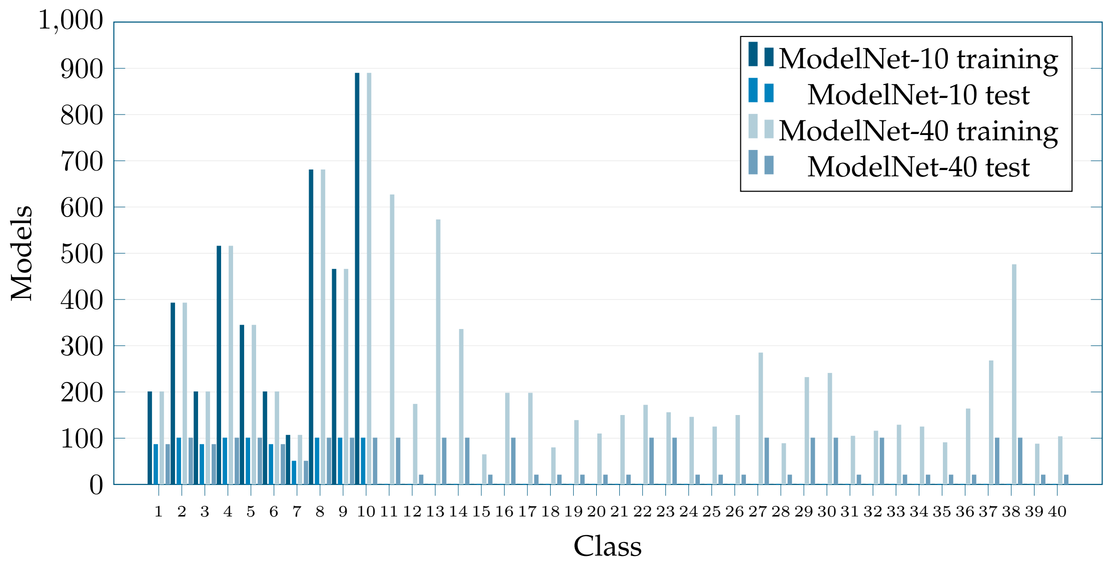


Figure 3.1: Model distribution per object class or category for both ModelNet-10 and ModelNet-40 training and test splits.

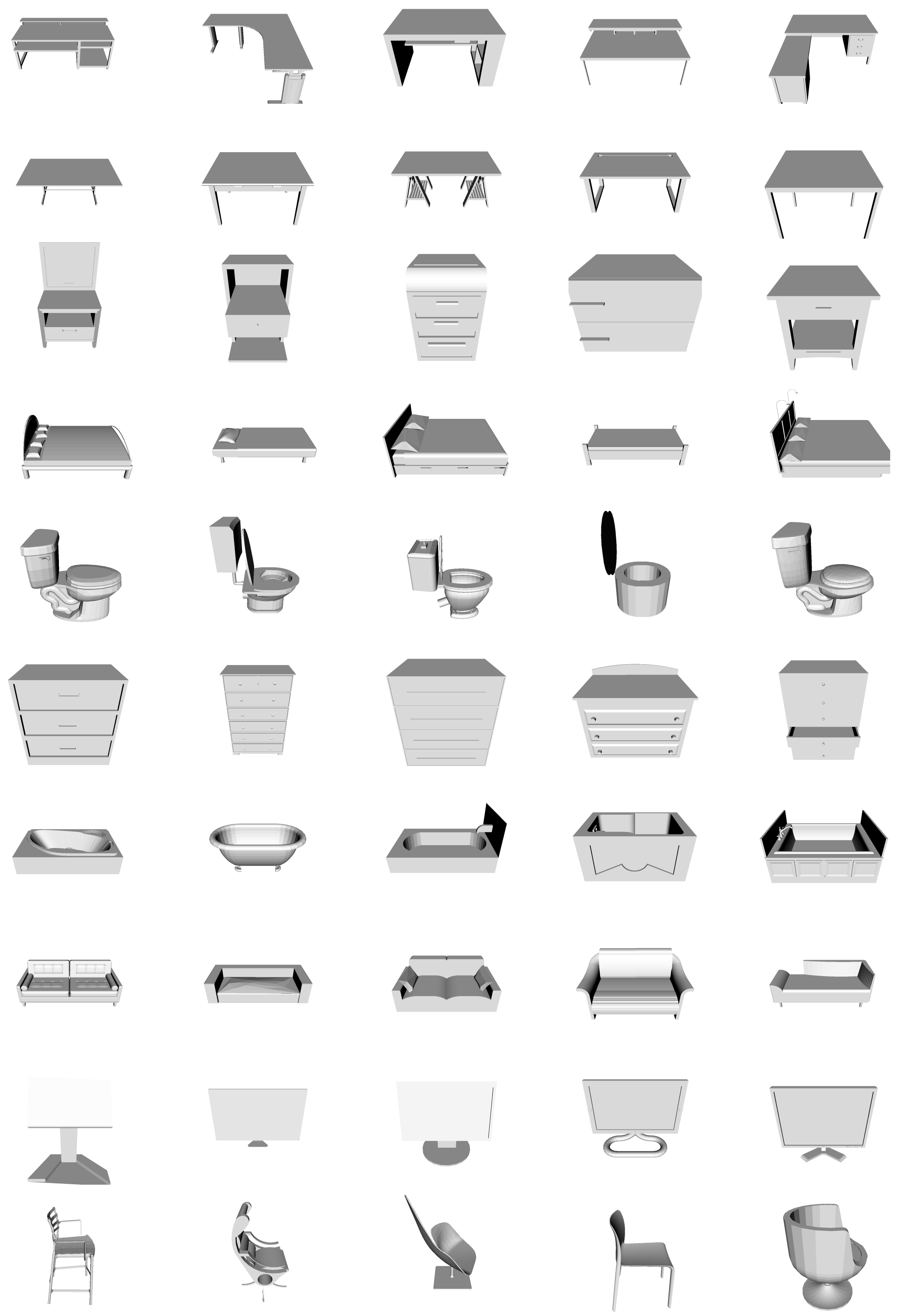


Figure 3.2: ModelNet10 samples.

Category	Training (10)	Test (10)	Training (40)	Test (40)
Desk	200	86	200	86
Table	392	100	392	100
Nighstand	200	86	200	86
Bed	515	100	515	100
Toilet	344	100	344	100
Dresser	200	86	200	86
Bathtub	106	50	106	50
Sofa	680	100	680	100
Monitor	465	100	465	100
Chair	889	100	889	100
Airplane	0	0	626	100
Bench	0	0	173	20
Bookshelf	0	0	572	100
Bottle	0	0	335	100
Bowl	0	0	64	20
Car	0	0	197	100
Cone	0	0	167	20
Cup	0	0	79	20
Curtain	0	0	138	20
Door	0	0	109	20
FlowerPot	0	0	149	20
GlassBox	0	0	171	100
Guitar	0	0	155	100
Keyboard	0	0	145	20
Lamp	0	0	124	20
Laptop	0	0	149	20
Mantel	0	0	284	100
Person	0	0	88	20
Piano	0	0	231	100
Plant	0	0	240	100
Radio	0	0	104	20
RangeHood	0	0	115	100
Sink	0	0	128	20
Stairs	0	0	124	20
Stool	0	0	90	20
Tent	0	0	163	20
TVStand	0	0	267	100
Vase	0	0	475	100
Wardrobe	0	0	87	20
XBox	0	0	103	20

Table 3.2: Model distribution per object class or category for both ModelNet-10 and ModelNet-40 training and test splits.

For this work, we will use of the ModelNet-10 subset, which contains a reasonable amount of models for both training and validation, mainly because this dataset was completely cleaned and the orientation of the models were manually aligned.

3.4 Hardware

Deep learning algorithms take tremendous computational resources to efficiently process huge amounts of data. To that end, it becomes mandatory to experiment in a powerful and energy-efficient computing solution. The *Asimov* server was assembled and configured for that purpose. It was built with the NVIDIA Digits DevBox [75] as inspiration, so most of its components were chosen based on the DevBox's ones.

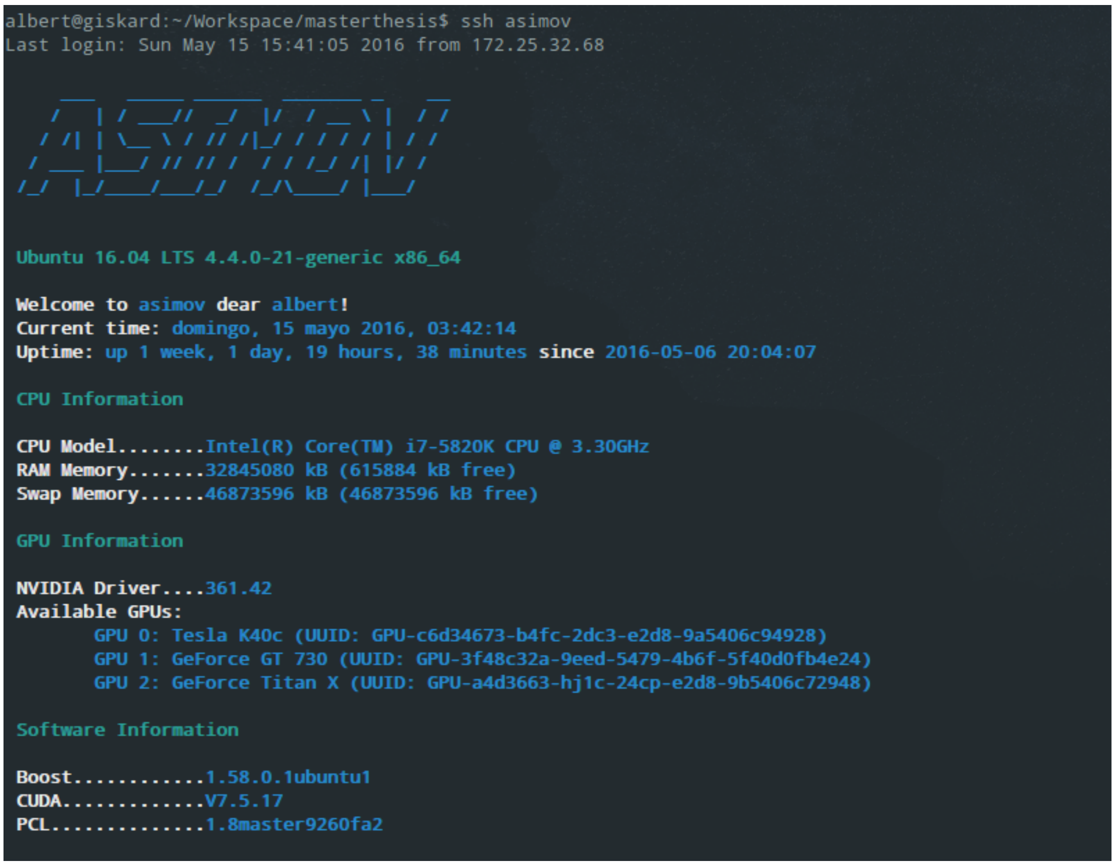
The full hardware configuration is shown in Table 3.3. The main features of the server are the three NVIDIA GPUs which were targeted at different goals. The Titan X will be devoted to deep learning, whilst a Tesla K40 was also installed for scientific computation purposes thanks to its exceptional performance with double precision floating point numbers. In addition, a less powerful GT730 was included for visualization and offloading the Titan X from that burden.

Regarding the software information, the server runs Ubuntu 16.04 LTS with Linux kernel $4.4.0 - 21-$generic for x86_64 architecture. The GPUs are running on NVIDIA driver version 361.42 and CUDA 7.5.

Other relevant software includes Caffe RC3, Point Cloud Library (PCL) 1.8 (master branch `9260fa2`), Vtk 5.6, and Boost 1.58.0.1. All libraries and tools were compiled using GNU Compiler Collection (GCC) 5.3.1 and the CMake 3.5.1 environment with release settings for maximum optimization.

	Asimov
Motherboard	Asus X99-A [11]
	Intel X99 Chipset
	$4\times$ PCIe $3.0/2.0 \times 16(\times16, \times16/\times16, \times16/\times16/\times8)$
CPU	Intel(R) Core(TM) i7-5820K CPU @ 3.30GHz [12]
	3.3 GHz (3.6 GHz Turbo Boost)
	6 cores (12 threads)
	140 W TDP
GPU (visualization)	NVIDIA GeForce GT730 [13]
	96 CUDA cores
	1024 MiB of DDR3 Video Memory
	PCIe 2.0
	49 W TDP
GPU (deep learning)	NVIDIA GeForce Titan X [14]
	3072 CUDA cores
	12 GiB of GDDR5 Video Memory
	PCIe 3.0
	250 W TDP
GPU (compute)	NVIDIA Tesla K40c [15]
	2880 CUDA cores
	12 GiB of GDDR5 Video Memory
	PCIe 3.0
	235 W TDP
RAM	4×8 GiB Kingston Hyper X DDR4 2666 MHz CL13
Storage (Data)	(RAID1) Seagate Barracuda 7200rpm 3TiB SATA III HDD [16]
Storage (OS)	Samsung 850 EVO 500GiB SATA III SSD [17]

Table 3.3: Hardware specifications of Asimov.



```
albert@giskard:~/Workspace/masterthesis$ ssh asimov
Last login: Sun May 15 15:41:05 2016 from 172.25.32.68

    ___   _____ ______  _______ _    __
   /   | / ___//  _/  |/  / __ \ |  / /
  / /| | \__ \ / // /|_/ / / / / | / /
 / ___ |___/ // // /  / / /_/ /| |/ /
/_/  |_/____/___/_/  /_/\____/ |___/


 Ubuntu 16.04 LTS 4.4.0-21-generic x86_64

 Welcome to asimov dear albert!
 Current time: domingo, 15 mayo 2016, 03:42:14
 Uptime: up 1 week, 1 day, 19 hours, 38 minutes since 2016-05-06 20:04:07

 CPU Information

 CPU Model........Intel(R) Core(TM) i7-5820K CPU @ 3.30GHz
 RAM Memory.......32845080 kB (615884 kB free)
 Swap Memory......46873596 kB (46873596 kB free)

 GPU Information

 NVIDIA Driver....361.42
 Available GPUs:
        GPU 0: Tesla K40c (UUID: GPU-c6d34673-b4fc-2dc3-e2d8-9a5406c94928)
        GPU 1: GeForce GT 730 (UUID: GPU-3f48c32a-9eed-5479-4b6f-5f40d0fb4e24)
        GPU 2: GeForce Titan X (UUID: GPU-a4d3663-hj1c-24cp-e2d8-9b5406c72948)

 Software Information

 Boost............1.58.0.1ubuntu1
 CUDA.............V7.5.17
 PCL..............1.8master9260fa2
```

Figure 3.3: Asimov's SSH banner message and welcome screen with server info.

In addition, the server was configured for remote access using Secure Shell (SSH). The installed version is OpenSSH 7.2p2 with OpenSSL 1.0.2. Authentication based on public/private key pairs was configured so only authorized users can access the server through an SSH gateway with the possibility to forward X11 through the SSH connection for visualization purposes. Figure 3.3 shows an SSH connection to Asimov.

At last, a mirrored Redudant Array of Independent Disks (RAID)1 setup was configured with both Hard Disk Drives (HDDs) for optimized reading and redundancy to tolerate errors on a data partition to store all the needed datasets, intermediate results, and models. The Solid State Drive (SSD) was reserved for the operating system, swap, and caching.

[11] https://www.asus.com/Motherboards/X99A/specifications/
[12] http://ark.intel.com/products/82932/Intel-Core-i7-5820K-Processor-15M-Cache-up-to-3_60-GHz
[13] http://www.geforce.com/hardware/desktop-gpus/geforce-gt-730/specifications
[14] http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x
[15] http://www.nvidia.es/content/PDF/kepler/Tesla-K40-PCIe-Passive-Board-Spec-BD-06902-001_v05.pdf
[16] http://www.seagate.com/es/es/internal-hard-drives/desktop-hard-drives/desktop-hdd/?sku=ST3000DM001
[17] http://www.samsung.com/us/computer/memory-storage/MZ-75E500B/AM

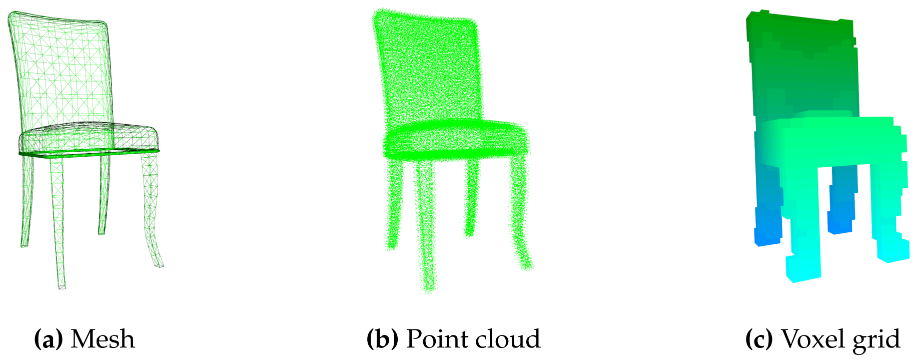
Chapter 4

Volumetric Representations

This chapter is devoted to volumetric representations for data that will be used as input for training and testing 2.5D and 3D CNNs. Firstly, we will describe what a volumetric representation is, which are the most common ones, and what are their advantages and weaknesses in Section 4.1. Next, in Section 4.2, we will review state-of-the-art representations that are currently being used together with CNNs. We will also discuss what a good representation should be and then propose novel ways of expressing this kind of data in Section 4.3. After that, in Section 4.4, we will carry out a set of experiments to determine the efficiency of those proposed representations. In the end, we will draw conclusions about the representations, we will show our expectations, and we will introduce the usage of the proposed representations in 2.5DCNNs.

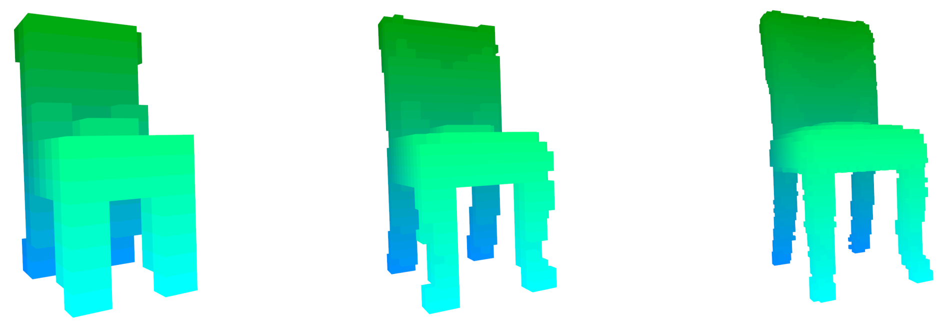
4.1 Introduction

In the previous chapter, we showed how CNNs can learn filters and recognize objects using 2D images as input. In the following chapters, we will explore the possibilities of recognizing object classes using 2.5D and 3D CNNs. For this purpose, we need volumetric data to feed the networks. Arguably, the most popular representations for volumetric data are 3D meshes or point clouds, shown in Figure 4.1a and 4.1b respectively. A mesh consists of a collection of vertices (points in a three-dimensional (3D) coordinate system), edges (connections between those vertices), and faces (closed sets of edges, usually triangles) that defines the shape of an object. A point cloud is just a set of points defined by x, y, and z coordinates in a three-dimensional coordinate system that model the surface of an object. However, those representations are unbounded and barely structured since they contain an arbitrary number of components, e.g., vertices or points, and no particular ordering is enforced for those entities.



(a) Mesh **(b)** Point cloud **(c)** Voxel grid

Figure 4.1: Common volumetric representations: polygonal mesh (a), point cloud (b), and voxel grid (c) of a chair model.

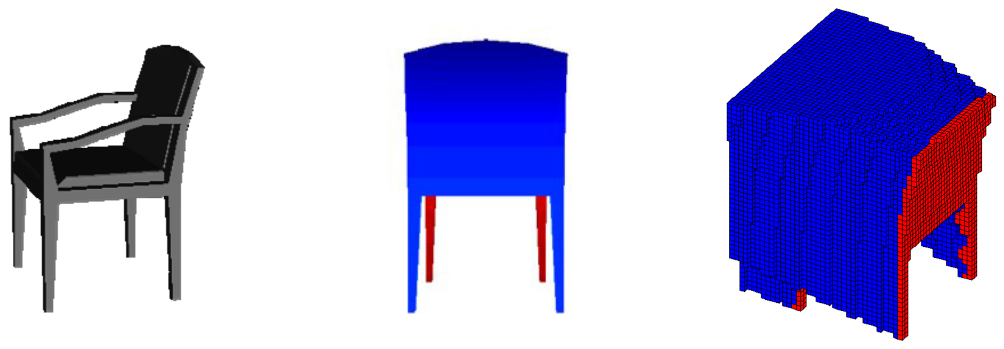


(a) $15 \times 15 \times 15$ **(b)** $30 \times 30 \times 30$ **(c)** $60 \times 60 \times 60$

Figure 4.2: Effect of the leaf size on binary voxel grids. All grids have the same cubic size: $300 \times 300 \times 300$ units. Leaf sizes vary from 5, 10, and 20 units, resulting in binary grids of $15 \times 15 \times 15$ (a), $30 \times 30 \times 30$ (b), and $60 \times 60 \times 60$ voxels (c) respectively.

This fact poses a problem since CNNs require a fixed-size representation for the input data. In order to overcome this limitation, alternative volumetric representations must be used to provide samples to the network for both training and testing. The most common volumetric representation which allows a structured and bounded definition of an object shape is the voxel grid. A voxel (word contraction of *volume element* or *volumetric pixel*) is the 3D equivalent to a 2D pixel, i.e., it is the minimal unit of a three-dimensional matrix. A volumetric object can be represented as a 3D matrix of voxels, whose positions are relative to other voxels while points and polygons must be represented explicitly by 3D coordinates. In this regard, voxels are able to efficiently represent regularly sampled 3D spaces that are also non-homogeneously filled, while meshes and point clouds are good for representing 3D shapes with empty or homogeneously filled space. It is important to notice that a voxel is just a data point in a three-dimensional grid, so its value may represent many different properties. The most popular and simple voxel grid type is the binary one (see Figure 4.1c) in which each voxel contains a binary value depending on whether the object's surface intersects or is partially contained in the voxel's volume.

Despite the fact that a binary voxel grid representation allows us to feed a CNN with volumetric data coming from different sources (point clouds provided by range sensors or polygonal meshes from 3D models can be easily converted into voxel grids) a significant amount of information from the original representation is lost. This loss depends on the resolution of the grid, i.e., the voxel size which is usually referred as the *leaf size*. Figure 4.2 shows how the resolution of the voxel grid can be tuned to obtain more accurate or more compact volumetric representations. Although an arbitrary precision can be obtained by changing the leaf size, it is hard to determine a specific size which describes with enough detail all the possible inputs for the CNN without sacrificing the compactness of the grid. Because of that, other representations which maintain the properties of the voxel grid, but include additional information in the values of the cells are needed. In this chapter we will review existing volumetric representations for learning with CNNs and we will propose alternative ones for this purpose. Our focus will be kept on balancing the accuracy of the representations and the runtime needed to compute them from other data sources, e.g., point clouds or polygonal meshes.

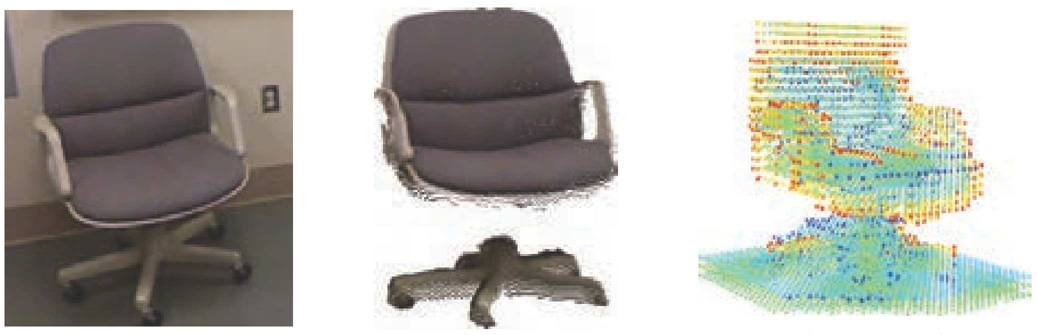


(a) Object model **(b)** Depth map **(c)** Voxel grid

Figure 4.3: *3DShapeNets* representation proposed by Wu *et al.* as shown in their paper [76]. An object (a) is captured from a certain point of view and a depth map is generated (b) which is in turn used to generate a point cloud that will be represented as a voxel grid (c) with empty voxels (in white, not represented), unknown voxels (in blue), and surface or occupied voxels (red).

4.2 Related Works

In this section we will review the most recent and popular ways of representing 3D data for CNNs. The first step was taken by Wu *et al.* [76], their work *3DShapeNets* was the first to apply CNNs to pure 3D representations. Their proposal (shown in Figure 4.3) represents 3D shapes, from captured depth maps that are later transformed into point clouds, as 3D voxel grids of size $30 \times 30 \times 30$ voxels – $24 \times 24 \times 24$ data voxels plus 3 extra ones of padding in both directions to reduce convolution artifacts – which can represent free space, occupied space (the shape itself), and unknown or occluded space depending on the point of view. Neither the grid generation process, nor the leaf size is described but the voxel grid relies on prior object segmentation.



(a) Object **(b)** Point cloud **(c)** TSDF grid

Figure 4.4: TSDF representation proposed by Song and Xiao as shown in their paper [77]. An object (a) is captured by a range sensor as a point cloud (b) and then a TSDF grid is generated (red indicates the voxel is in front of surfaces and blue indicates the voxel is behind the surface; the intensity of the color represents the TSDF value).

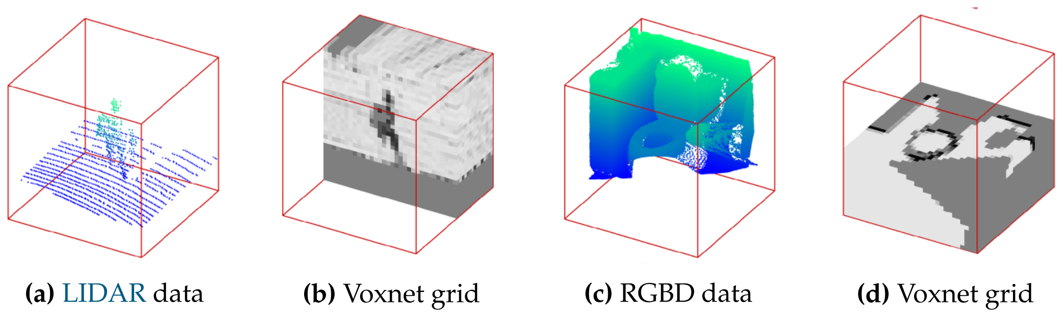

(a) LIDAR data (b) Voxnet grid (c) RGBD data (d) Voxnet grid

Figure 4.5: Volumetric occupancy grid representation used by *VoxNet* as shown in their paper [78]. For LIDAR data (a) a voxel size of 0.1m^3 is used to create a $32 \times 32 \times 32$ grid (b). For RGB-D data (c), the resolution is chosen so the object occupies a subvolume of $24 \times 24 \times 24$ voxels in a $32 \times 32 \times 32$ grid (d).

Song and Xiao [77] proposed to adopt a directional TSDF encoding which takes a depth map as input and outputs a volumetric representation. They divide a 3D space using an equally spaced voxel grid in which each cell holds a three-dimensional vector that records the shortest distance between the voxel center and the three-dimensional surface in three directions. In addition, the value is clipped by 2δ, being δ the grid size in each dimension. A $30 \times 30 \times 30$ voxels grid is fitted to a previously segmented object candidate. Figure 4.4 shows a graphical representation of this approach.

Maturana and Scherer [78] use occupancy grids in *VoxNet* to maintain a probabilistic estimate of the occupancy of each voxel to represent a 3D shape. This estimate is a function of the sensor data and prior knowledge. They propose three different occupancy models: binary, density, and hit. The binary and density models make use of raytracing to compute the number of hits and pass-throughs for each voxel. The former one assumes that each voxel has a binary state, occupied or unoccupied. The latter one assumes that each voxel has a continuous density, based on the probability it will block a sensor beam. The hit grid ignores the difference between unknown and free space, only considering hits; it discards information but does not require the use of raytracing so it is highly efficient in comparison with the other methods. They also propose two different grids for Light Detection and Ranging (LIDAR) and RGB-D sensor data. For the RGB-D case, they use a fixed occupancy grid of $32 \times 32 \times 32$ voxels, making the object of interest – obtained by a segmentation algorithm or given by a sliding box – occupy a subvolume of $24 \times 24 \times 24$ voxels. The z axis of the grid is aligned with the direction of gravity. Figure 4.5 shows the occupancy grids used by *VoxNet*.

4.3 Proposed Representations

As is clear from the previous sections, a volumetric representation to be fed to a 2.5D or 3DCNN must encode the 3D shape of an object as a 3D tensor of binary or real values. This is due to the fact that raw 3D data is sparse, i.e., a 3D shape is only defined on its surface, and CNNs are not engineered for this kind of data.

In this regard, our proposal is twofold. First, we implemented two different ways of generating the structure of the tensor – position, grid size, and leaf size – using a fixed grid and an adaptive one. Second, we developed three possible occupancy measures for the volumetric elements of the tensor. Those proposals build up on the previously reviewed representations, cherry-picking their strengths.

In this section we will describe how to adapt a dataset of choice to train the CNNs to our representation input, including ways to artificially augment the dataset for a proper training process. After that, we will show the grid generation process using the adapted data. In the end, we will discuss the computation of the occupancy of each voxel considering the three proposed alternatives.

4.3.1 Dataset Adaptation

In order to train and test the 2.5D and 3D CNNs that will be described in the following sections, we decided to use the *Princeton ModelNet* object dataset. It is a collection of CAD models of household objects in Object File Format (OFF). More details about the dataset are provided in Chapter 3. The main reason for choosing this dataset is the fact that training deep neural networks, such as a CNN, requires a huge amount of examples to obtain good generalization capabilities. Because of that fact, and considering the lack of large scale 3D datasets of real-world objects, using synthetic datasets with a significant number of instances is justified. However, those datasets should be adapted to resemble the expected input for the system coming from the sensors scanning real-world objects.

Since the final goal of the CNNs is to provide means to recognize objects onboard a mobile robotic platform which features RGB-D sensors, it is logical to transform the full mesh representation provided by the dataset into the representation provided by the sensors. RGB-D cameras output depth maps which can be used to generate 3D point clouds of the scene viewed by the camera. Then it is clear that the input of our system will be point clouds which are partial views of the actual scene, i.e., what we see depends on the Point of View (POV) of the camera. In this regard, we will transform each CAD object of the dataset into partial point clouds from different POVs.

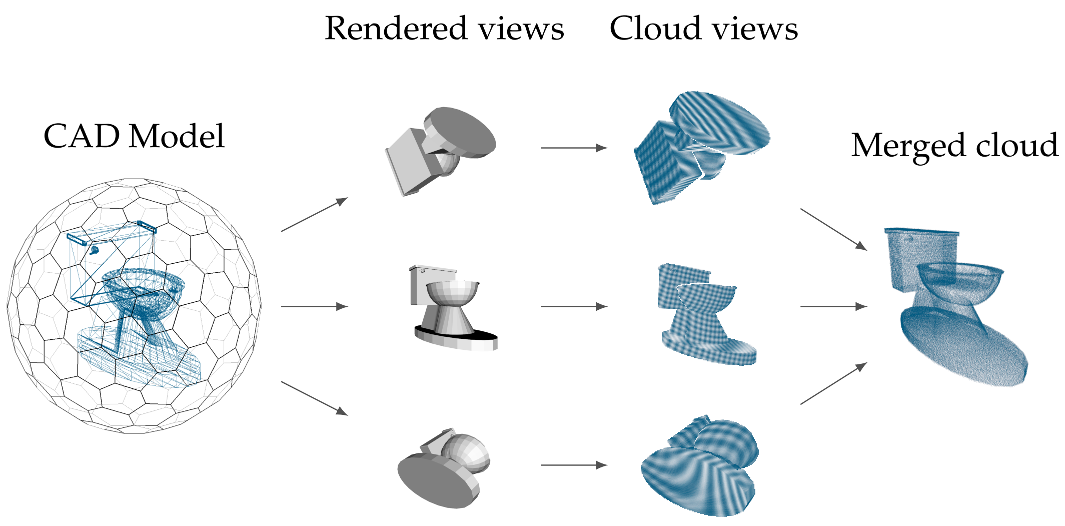


Figure 4.6: From CAD models to point clouds. The object is placed in the center of a tessellated sphere, views are rendered placing a virtual camera in each vertex of the icosahedron, the z-buffer data of those views is used to generate point clouds, and the point clouds are transformed and merged at last.

For this purpose, we converted the OFF models into Point Cloud Data (PCD) clouds using a raytracing-based process. The object is placed in the center of a 3D sphere, which is tessellated to a certain level, and a virtual camera pointing to the center of the sphere is placed in each vertex of that truncated icosahedron. Then those partial views are rendered and their z-buffer data, which contains the depth information, is used to generate point clouds from each POV. In the end, those views are translated and rotated, depending on their POV, and merged into a cloud for the full object.

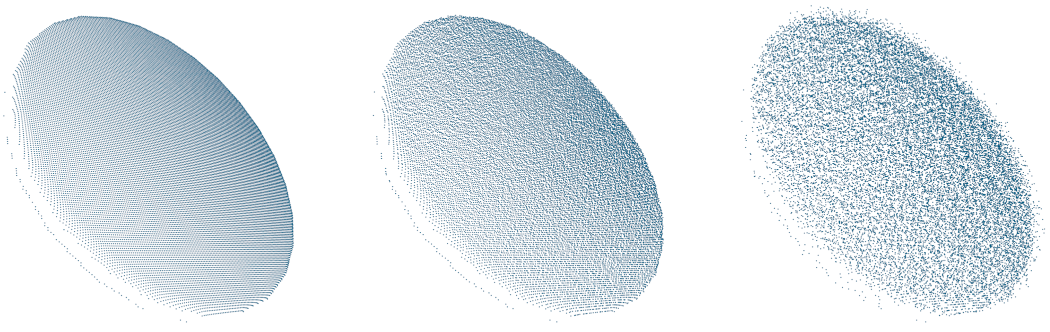
Figure 4.6 shows a diagram of the aforementioned process. For the conversion, we used the first tessellation level of the sphere, which generates 42 vertices or POVs. A resolution of 256×256 pixels was used for rendering the views. A voxel grid filter with a leaf size of $0.7 \times 0.7 \times 0.7$ units is applied to the merged cloud to equalize the point density, which is higher in certain zones due to view overlapping.

Noise

The partial views generated using the previously described process are not a good simulation of the result that we would obtain by using a low-cost RGB-D sensor. Those systems are noisy, so the point clouds produced by them are not a perfect representation of the real-world objects. Taking this fact into consideration, it is reasonable to think that it is useful to make the synthetic views noisy. The benefits are twofold: we can train our network using artificial data that resembles sensor output and the dataset can be augmented using different levels of noise to avoid overfitting.

In order to properly simulate the behavior of a sensor, a model is needed. In our case, we are dealing with low-cost RGB-D sensors such as Microsoft Kinect and Primesense Carmine. A complete noise model for those sensors, specifically for the Kinect device, must take into account occlusion boundaries due to distance between the Infrared (IR) projector and the IR camera, 8-bit quantization, 9×9 pixel correlation window smoothing, and z-axis or depth Gaussian noise [79].

We will make use of a simplification of this model, only taking into account the Gaussian noise since it is the most significant one for the generated partial views. In this regard, the synthetic views are augmented by adding Gaussian noise to the z dimension of the point clouds with mean $\mu = 0$ and different values for the standard deviation σ to quantify the noise magnitude. Figure 4.7 shows the effect of this noise over a synthetic partial view of one object of the dataset.



(a) $\sigma = 0$ **(b)** $\sigma = 0.1$ **(c)** $\sigma = 1$

Figure 4.7: Different levels of noise ($\sigma = 0$ (a), $\sigma = 0.1$ (b), and $\sigma = 1$ (c)) applied to the z-axis of every point of a table partial view.

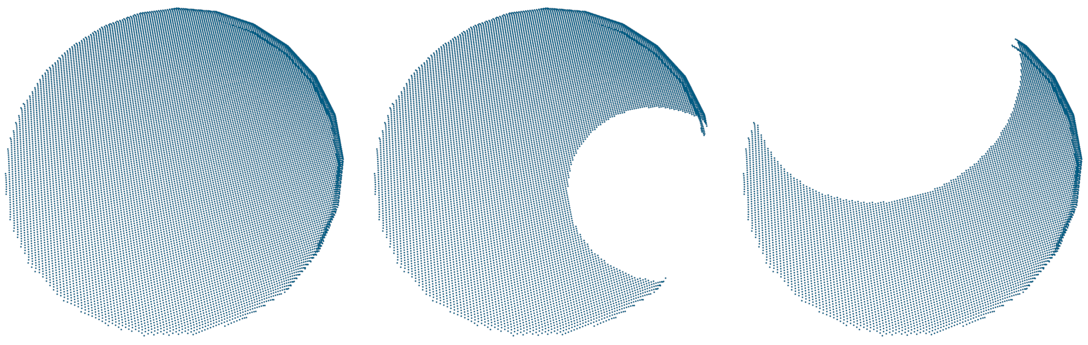
Occlusion

Adding noise is a good alternative for augmenting the dataset and, at the same time, synthetically generate training data that is somehow similar to the actual output of RGB-D sensors. However, besides modelling the sensor to improve our synthetic data, it is important to also take the environment into account. In a real-world scenario, objects are not usually perfectly isolated and easily segmented; in fact, it is common for them to be occluded by other elements of the scene.

In order to introduce more variability in our training set and avoid overfitting, we will include occluded versions of the previously generated partial views. Furthermore, the test and validation sets will also be occluded to determine the robustness of the CNNs when dealing with incomplete or missing data.

The occlusion process consists of picking a random point of the cloud with a uniform probability distribution. Then, a number of closest neighbors to that point are picked. At last, both the neighbors and the point are considered occluded surface and removed from the point cloud. The number of neighbors to pick depends on the amount of occlusion ψ we want to simulate. For instance, for an occlusion $\psi = 25\%$ we will remove neighbors until the rest of the cloud contains a 75% of the original amount of points, i.e., we will remove a 25% of the original cloud. Figure 4.8 shows the effect of the random occlusion process with different occlusion factors ψ over a synthetic partial view of a table object of the dataset.

It is important to notice the randomness of the occlusion process. This means that even with a high ψ it is possible not to remove any important surface information and vice versa. In other words, it is possible for some objects to remove a 50% of their points and still be recognizable because the removed region was not significant at all, e.g., a completely flat surface. However it is possible to render an object unrecognizable by removing a small portion of its points if the randomly picked surface is significant for its geometry. This remark is specially important when testing the robustness of the system. In order to guarantee that an appropriate measure of the robustness against missing information is obtained, a significant amount of testing sets must be generated and their results averaged so that it is highly probable to test against objects which have been occluded all over their surface across the whole testing set.



(a) $\psi = 0\%$ **(b)** $\psi = 25\%$ **(c)** $\psi = 50\%$

Figure 4.8: Different levels of occlusion ($\psi = 0\%$ (a), $\psi = 25\%$ (b), and $\psi = 50\%$ (c)) applied randomly to a table partial view.

4.3.2 Grid Generation

Once the dataset has been properly adapted to simulate real-world sensor data, we need to generate a discretized representation of the unbounded 3D data to feed the network. As we previously stated, each view will be represented as a 3D tensor. For that purpose, we need to spawn a grid to subdivide the space occupied by the point clouds. Two types are proposed: one with fixed leaf and grid sizes, and another one which will adapt those sizes to fit the data.

Fixed

This kind of grid sets its origin at the minimum x, y, and z values of the point cloud. Then the grid is spawned, with fixed and predefined sizes for both grid and voxels. After that, the cloud is scaled up or down to fit the grid. The scale factor is computed with respect to the dimension of maximum difference between the cloud and the grid. The cloud is scaled with that factor in all axes to maintain the original ratios. As a result, a cubic grid is generated as shown in Figure 4.9.

Adaptive

The adaptive grid also sets its origin at the minimum x, y, and z values of the point cloud. Next, the grid size is adapted to the cloud dimensions. The leaf size is also computed in function of the grid size. Knowing both parameters, the grid is spawned, fitting the point cloud data. As a result, a non-cubic grid is generated. As shown in Figure 4.10, all voxels have the same size, but they are not necessarily cubic.

It is important to remark that, in both cases (fixed and adaptive), the number of voxels in the grid is fixed. Figures 4.9 and 4.10 show examples for both types using $8 \times 8 \times 8$ voxels for the sake of a better visualization.

It is also important to notice that each representation serves a purpose. The fixed grid will not always fit the data perfectly so it might end up having sparse zones with no information at all (as seen in Figure 4.9a on the first column). However, it can be used right away for sliding box detection. On the contrary, the adaptive grid fits the

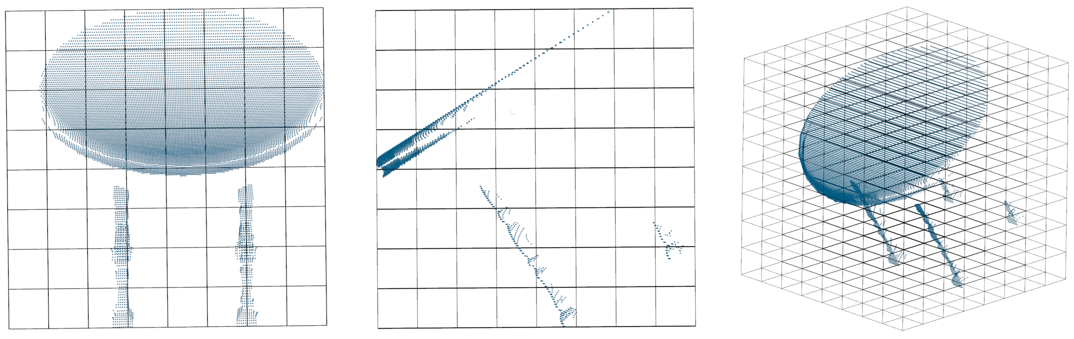
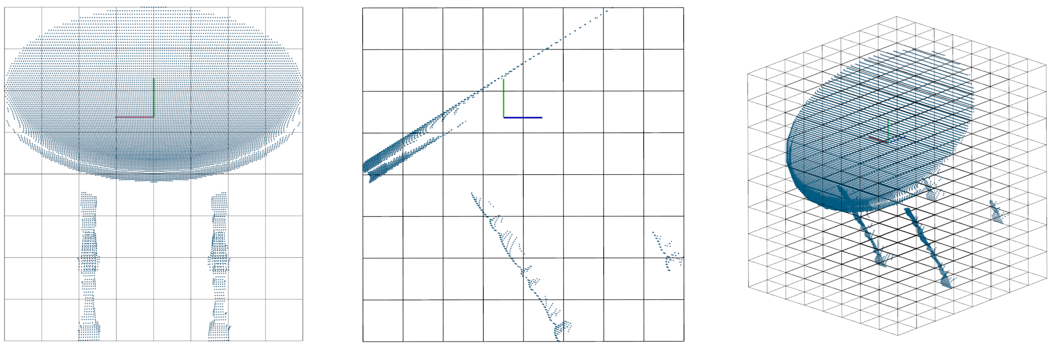

(a) Front **(b)** Side **(c)** Perspective

Figure 4.9: A fixed occupancy grid ($8 \times 8 \times 8$ voxels) with 40 units leaf size and 320 units grid size in all dimensions. The grid origin is placed at the minimum x, y, and z values of the point cloud. Front (a), side (b), and perspective (c) views of the grid over a partial view of a segmented table object are shown.

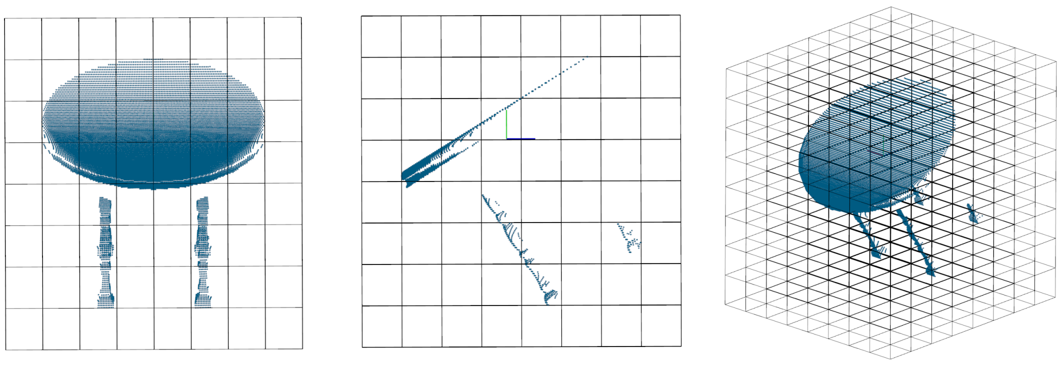
data to achieve a better representation. Nonetheless, it relies on a proper segmentation of the object to spawn the grid.



(a) Front **(b)** Side **(c)** Perspective

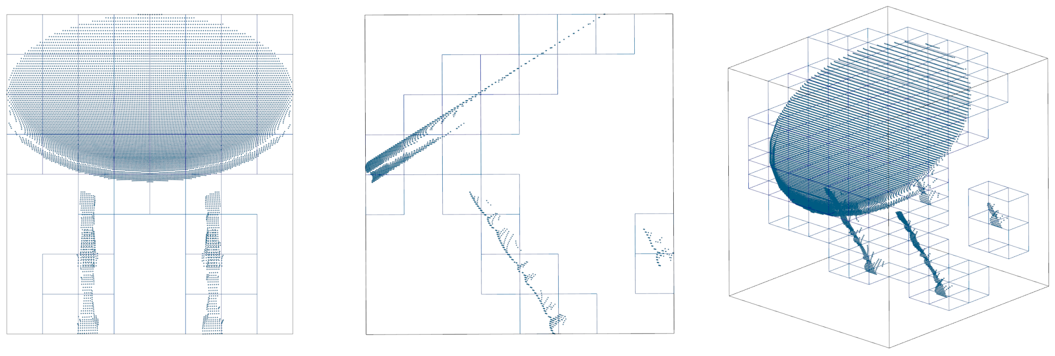
Figure 4.10: An adaptive occupancy grid ($8 \times 8 \times 8$ voxels) with adapted leaf and grid sizes in all dimensions to fit the data. The grid origin is placed at the minimum x, y, and z values of the point cloud. Front (a), side (b), and perspective (c) views of the grid over a partial view of a segmented table object are shown.

An important feature to add to both representations is the so called *padding*. It is a good practice to pad the input data with zeros to avoid convolution artifacts when the filter interacts with the boundaries [80]. In our case, padding helps the CNN retain the spatial dimensions of the input. In this regard, when using a filter size of $3 \times 3 \times 3$ it becomes obvious that a padding of 1 voxel will avoid the CNN altering the spatial dimensions of the input. The same applies when using a $5 \times 5 \times 5$ filter and a padding of 2 voxels. Figure 4.11 shows a padded example of the previously shown adaptive grid in Figure 4.10.



(a) Front **(b)** Side **(c)** Perspective

Figure 4.11: An adaptive occupancy grid ($8 \times 8 \times 8$ voxels) with adapted leaf and grid sizes in all dimensions to fit the data. One cell padding is enforced so that the grid contains a halo of empty cells around the actual data. Front (a), side (b), and perspective (c) views of the grid over a partial view of a segmented table object.



(a) Front **(b)** Side **(c)** Perspective

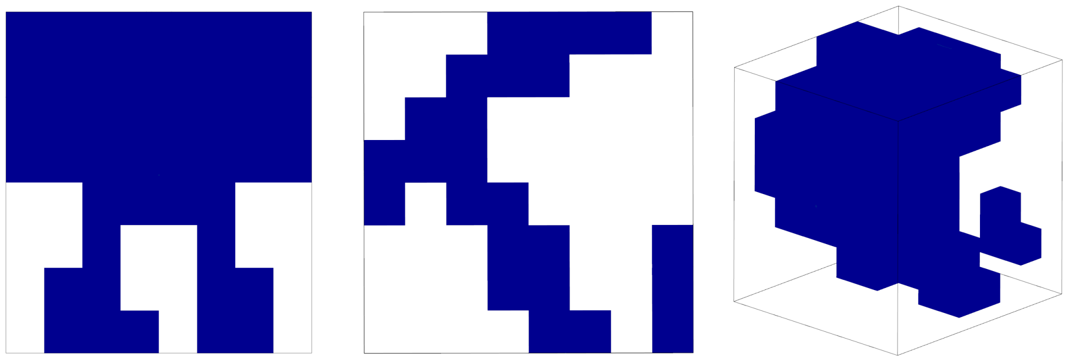
Figure 4.12: Occupied voxels in an adaptive $8 \times 8 \times 8$ grid generated over a partial view point cloud. Those voxels with points inside are shown in a wireframe representation. Empty voxels are omitted. Occupied voxels must be filled with values which represent the contained shape.

4.3.3 Tensor Computation

After spawning the grid to generate a discrete space, we need to determine the values for each cell or voxel of the 3D tensor. In order to do that, we must encode the geometric information of the point cloud into each occupied cell (see Figure 4.12). In other words, we have to summarize as a single value, the information of all points which lie inside a certain voxel. One way to do that is using occupancy measures. For that purpose, we propose three different alternatives: binary occupancy, normalized density, and surface intersection.

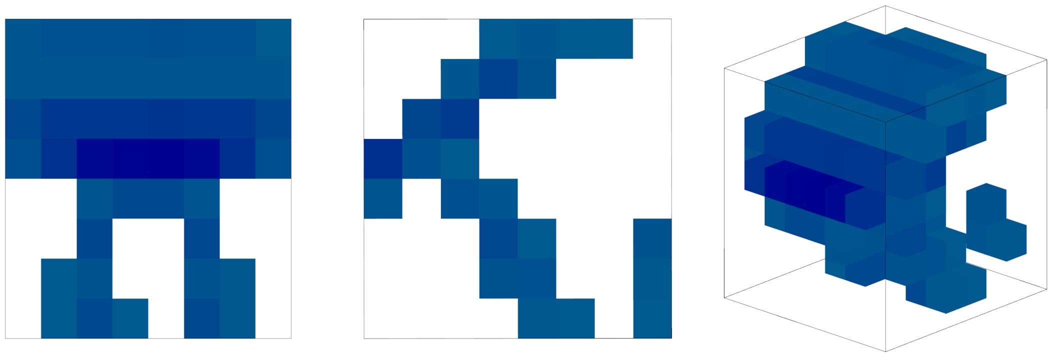
Binary

The binary tensor is the simplest representation that can be conceived to encode the shape. Voxels will hold binary values, they will be considered occupied if at least a point lies inside, and empty otherwise. Figure 4.13 shows an example of this tensor.



(a) Front **(b)** Side **(c)** Perspective

Figure 4.13: Binary tensor computed over a point cloud of a partial view of an object (shown in Figure 4.12). Occupied voxels are shown in blue, empty voxels are omitted for the sake of simplicity.



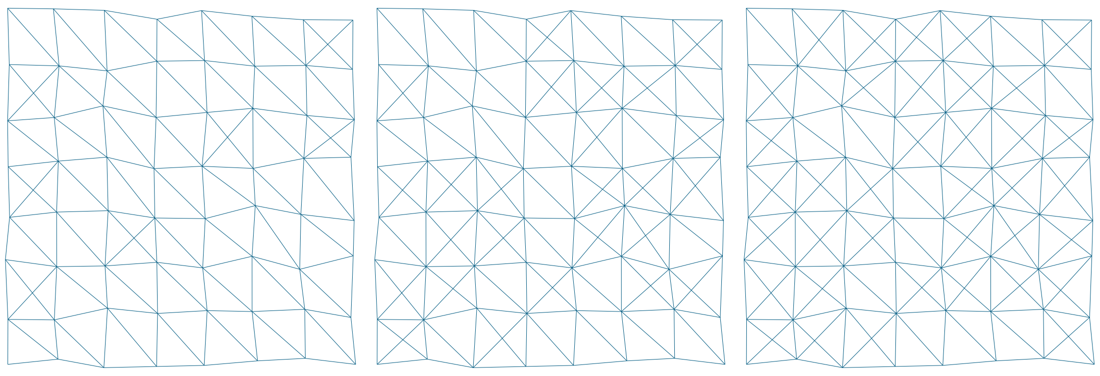
(a) Front **(b)** Side **(c)** Perspective

Figure 4.14: Normalized density tensor over a point cloud of a partial view of an object (shown in Figure 4.12). Denser voxels are darker and sparse ones are shown in light blue. Empty voxels were removed for visualization purposes.

Normalized Density

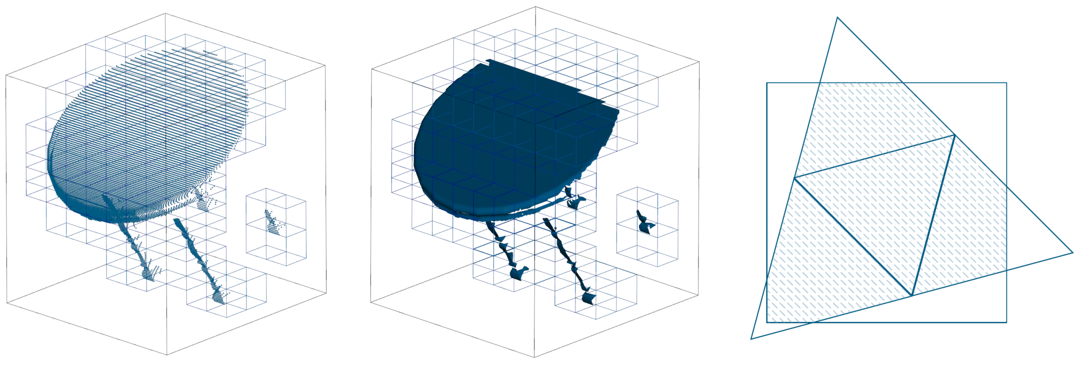
Binary representations are simple and require low computational power. However, complex shapes may get oversimplified so useful shape information gets lost. This representation can be improved by taking into account more shape information. A possible alternative consists of computing the point density inside each voxel, i.e., counting the number of points that fall within each cell.

It is important to notice that point density directly depends on the cloud resolution which in turn depends on many factors involving the camera and the scene, e.g., it is common for RGB-D to generate denser shapes in closer surfaces. To alleviate this problem, we can normalize the density inside each voxel dividing each value by the maximum density over the whole tensor. An example of normalized density tensor is shown in Figure 4.14.



(a) Low density **(b)** Medium density **(c)** High density

Figure 4.15: Triangulation with varying point densities. The total surface or area remains the same despite the fact that the number of vertices is increasing.



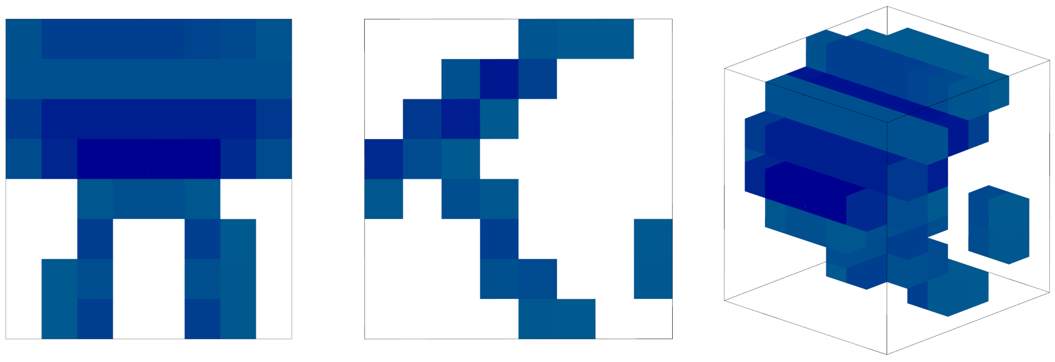
(a) Point cloud **(b)** Triangulated mesh **(c)** Triangle intersections

Figure 4.16: Surface intersection tensor calculation steps. A partial view point cloud (a) is triangulated to generate a mesh (b). Then the value of each voxel corresponds to the area of intersection of the mesh and that particular voxel; for each voxel the intersecting triangles are clipped and the areas of the resulting polygons are added up (c).

Surface Intersection

Despite the fact that the normalized density representation is an improvement over the binary tensor, it is still sensitive to varying resolutions depending on the point of view and other factors. To overcome this problem, we can leverage to higher level shape information such as the very same surface. We can compute the amount of shape surface that intersects – falls within – a certain voxel and use it as an occupancy measure. By doing this, we gain robustness against density variations (see Figure 4.15).

Unfortunately, by increasing the robustness of the representation, we also incur in a computational cost penalty because of the need of triangulating the point cloud to obtain that higher level of surface information. For a certain partial view, a triangle mesh based on projections of the local neighborhoods is obtained by applying a greedy surface triangulation algorithm proposed by Marton *et al.* [81]. Once the surface has been triangulated, we compute the area of intersection between the mesh and each



(a) Front **(b)** Side **(c)** Perspective

Figure 4.17: Surface intersection tensor over a point cloud of a partial view of an object (shown in Figure 4.12). Those voxels with more surface intersection area are darker. Empty voxels were removed for visualization purposes.

voxel. Those triangles which partially intersect a voxel are clipped and then the area of the resulting polygon is computed. Figure 4.16 illustrates the described process.

An example of surface intersection tensor is shown in Figure 4.17. As we can observe, the surface-based tensor exhibits more details than the previous ones and represents better the original point cloud.

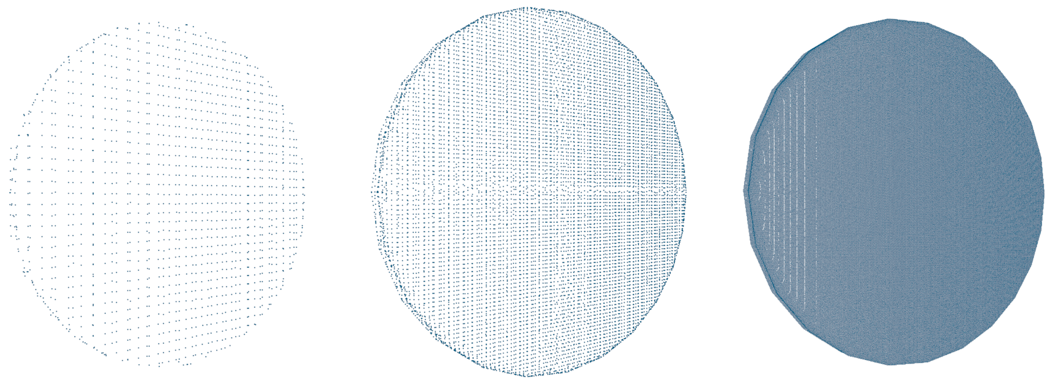
4.4 Experimentation

One of the main concerns of our target system, besides accuracy, is performance. In order to be able to process sensor data in real-time, the calculation of the volumetric representation must be efficient and it should scale properly. To determine the performance of our implementations, we carried out a set of experiments. Those tests consisted of two different benchmarks, each one measuring the influence of two distinct variables: the cloud point count and the grid size.

On the one hand, we tested how performance is affected when increasing the point count of a cloud. For this purpose, we used a partial view of the previously shown table object. That view was originally generated using a resolution of 1080×1080 pixels for the render. This generated a cloud with approximately $300k$ points. We then downsampled this cloud to reduce the point count as shown in Figure 4.18 using uniform sampling. For this benchmark, we used an adaptive grid with $16 \times 16 \times 16$ voxels and 2 padding cells.

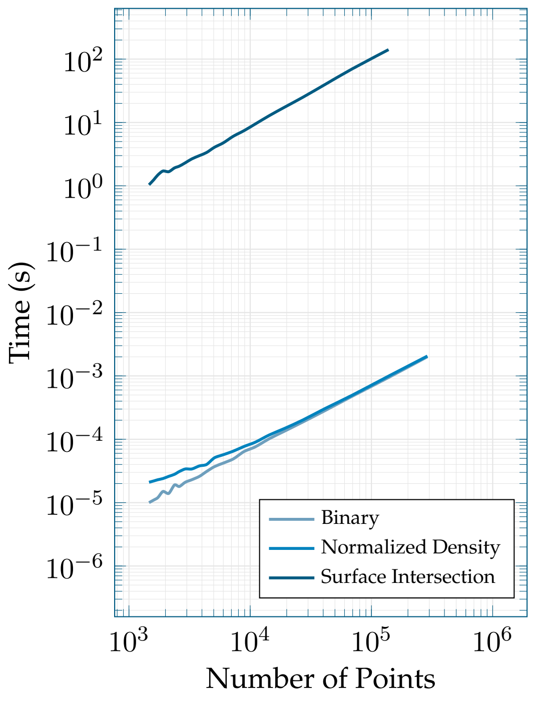
On the other hand, we also tested the performance scaling when increasing the grid size. We started from a small grid of $8 \times 8 \times 8$ voxels and then increased its size by 1 voxel until we reached the maximum size of $64 \times 64 \times 64$ voxels. For this experiment we used one of the downsampled clouds of the previous experiment with an intermediate point count of roughly $12k$ points. The grid was also adaptive.

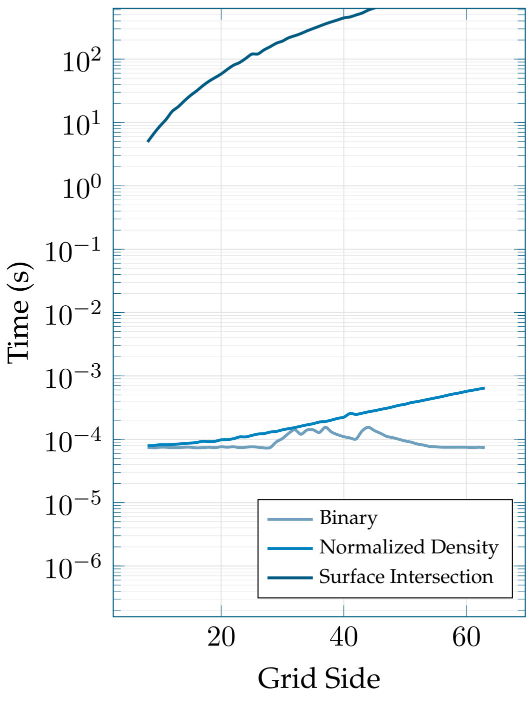
It is important to remark that the difference, in terms of computational cost, between using the fixed and the adaptive grids is almost negligible since both implementations make use of essentially the same operations. Furthermore, the grid spawning process has constant complexity. Because of that, there is no need to test the scaling of those processes.



(a) Low count **(b)** Medium count **(c)** High count

Figure 4.18: Partial view point cloud with varying point count to test occupancy computation performance scaling as the number of points increases. Clouds with low (a), medium (b), and high (c) point counts are shown.





(a) **(b)**

Figure 4.19: Results of the experimentation carried out to test the performance scaling of volumetric representations. The first plot (a) shows the scaling of the three occupancy computation methods (binary, normalized density, and surface intersection) when the number of points of the cloud increases (using an adaptive grid of $16 \times 16 \times 16$ voxels with 2 padding cells). The second plot (b) shows the scaling of the same three methods using a cloud with a fixed number of points but increasing the grid size.

4.4.1 Results and Discussion

The results of the experimentation are shown in Figure 4.19. As expected, the binary method is the fastest one in all experiments, followed by the normalized density one. The surface intersection method exhibits a computational cost which is orders of magnitude above both of them.

As we can observe in Figure 4.19a, when the number of points increases, the three methods scale linearly. It is important to notice that, for small clouds, the normalized density method is slower than the binary one due to the necessity of normalizing the whole grid. However, once the execution time becomes neglectable, there is no difference between those methods. In addition, and despite its linear scaling, the surface intersection method is orders of magnitude slower than the other ones due to the triangulation and clipping processes.

Regarding to the grid size scaling, the first thing we can observe in Figure 4.19b is the constant cost of the binary method. This is due to the fact that the implementation loops over the point cloud, but finding their corresponding voxel is computed in constant time. In this case, the normalized density grid does not behave as the binary one because the normalization process loops over all voxels so the execution time of that process scales at the same rate as the grid size. The surface intersection method scales badly with the grid size since we have to check each voxel individually to determine which triangles intersect them.

4.5 Conclusion

In this chapter we have proposed and analyzed multiple volumetric representations for 3D data that will be used by our CNNs later in this work. First, we described what a volumetric representation is and why do we need them to feed a CNN. Later, we reviewed common volumetric representations used for CNNs in the literature. After that, we determined what a good representation should be and presented a set of alternatives for our data. We also explained how to adapt our dataset of choice to those representations. At last, we carried out an empirical study to analyze the efficiency of the implemented representations.

This study provided us insight about the performance of those representations in terms of execution time needed to be computed. We found out a tradeoff between sophistication and computational cost. In other words, those representations which provide more accurate or more robust information require extra processes that add up run time. In conclusion, the density grid offered a good balance between representation accuracy and computational cost. In the extremes, the binary grid is plain but extremely fast whilst the surface intersection one is richer but its cost might be prohibitive. Those proposals will be further tested after implementing 2.5D and 3D CNNs to determine which ones are more suitable for our problem in terms of accuracy.

In the next chapter we will discuss 2.5DCNNs as a previous step towards pure 3DCNNs. This first approach will be useful to start gaining insight about the performance of the different representations and to start designing a proper architecture before diving deep into the complexity of 3DCNNs.

Chapter 5

3D CNN for Object Recognition

In this chapter we gather all the knowledge and conclusions extracted from previous ones to design, implement, and train a 3D CNN. In Section 5.1, we will first introduce the concept of 2.5D and 3D CNNs and why do we expect a gain by moving from 2D to 3D. After that, in Section 5.2, we will explore the related works about 2.5D CNNs and its evolution to 3D CNNs. Next, in Section 5.3, we will design, implement, and train a 2.5D and a 3D CNNs using full object models. We will also conduct a detailed study of the effect of noise and occlusion over the accuracy of the networks. At last, we will draw conclusions about the experiments in Section 5.4.

5.1 Introduction

In the previous chapters, we showed how CNNs are able to recognize objects in 2D images by learning convolution filters to detect features. We also introduced the possibility of adding a new dimension to this problem by using 3D data instead of plain images. In order to do that, we stated that we need a way to encode that 3D information into a volumetric and structured representation that can be provided as input to the CNN. We also designed and implemented a set of representations after reviewing the state of the art. Those representations were tested to determine their performance and scalability. All of that provided us with all we need to start designing, implementing, and training 3D CNNs. But before diving deep into that, we will first lay down the differences between 2D CNNs and 3D ones, using 2.5D CNNs as an intermediate step.

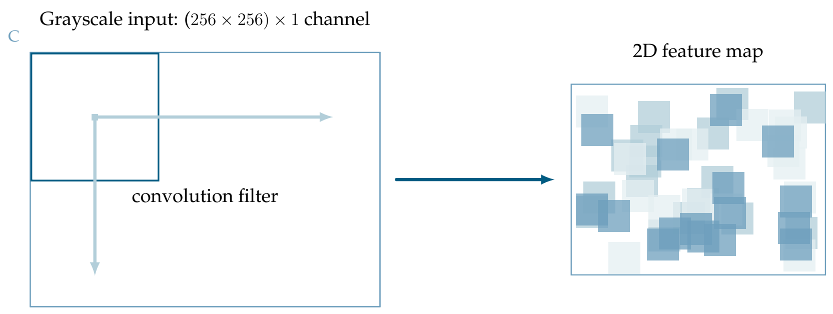


Figure 5.1: Applying a 2D convolution to a single-channel 2D input, in this case a grayscale image, results in a 2D feature map. The filter (in dark blue) has a fixed width and height and it is slided across the width and height of the input image, producing a 2D feature map as a result of the matrix multiplications of the filter and the input.

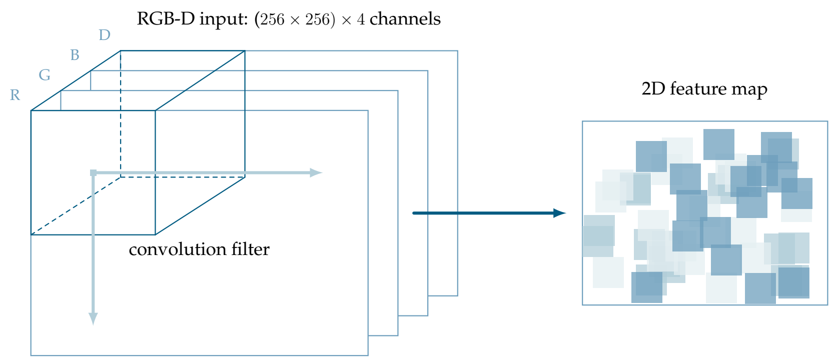


Figure 5.2: Applying a 2D convolution to a multi-channel input, in this case RGB-D with four channels, results in a 2D feature map. The filter (in dark blue) has a fixed width and height, but it extends through the full depth of the input volume. During the forward pass, the filter is slided across the width and height of the input volume, producing a 2D activation or feature map.

5.1.1 2.5D Convolutional Neural Networks

Figure 5.1 shows an example of applying a 2D convolution to an image with a single channel. In that case, sliding the kernel over the whole image as described in Chapter 2 produces a 2D feature map. However, most images do not have a single channel but three (usually red, green, and blue). In that case, the filter is extended through all channels but it is still slided across the width and height of the input volume, not across the channels or depth.

This behavior can be exploited to feed the CNN with 2.5D data by adding a depth channel to a common RGB image, creating an RGB-D input volume with four channels. The convolution filter will then be extended through the four channels, creating a somewhat volumetric kernel as shown in Figure 5.2. Furthermore, we could take a volumetric representation like the ones proposed in Chapter 4 and slice it in the depth dimension into channels. Then we could feed that representation directly to a CNN. However, this approach will not slide the filter across the depth of the input volume.

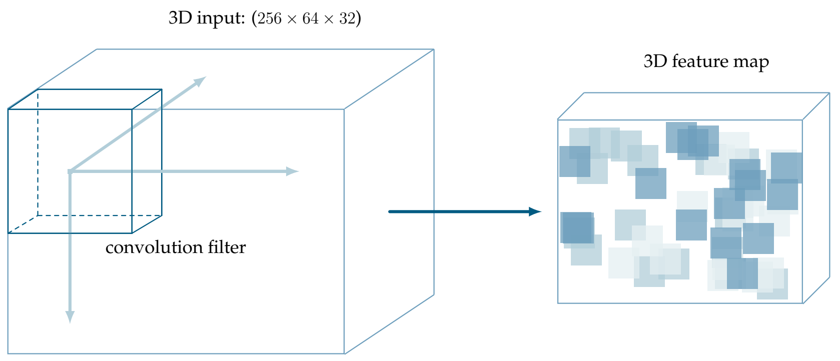


Figure 5.3: Applying a 3D convolution to a single-channel volumetric input, in this case a $256 \times 64 \times 32$ grid, results in a3D feature map. The filter (in dark blue) has a fixed width, height, and depth. During the forward pass, the filter is slided across the width, height, and depth of the input volume, producing a 3D activation map.

5.1.2 3D Convolutional Neural Networks

On the other hand, 3D convolutions use kernels with fixed width, height, and depth. That filters are slided across the width, height, and depth of an input volume. By doing that, 3D convolutions model spatial information better than 2.5D counterparts which only extend the kernel over the depth dimension but do not slide it over that dimension thus losing spatial information. As shown in Figure 5.3, a 3D convolution is applied to a single-channel input volume producing a volumetric feature map.

5.2 Related Works

After clarifying the differences between 2D, 2.5D, and 3D convolutions, we will review the literature to analyze state-of-the-art 2.5D and 3D approaches. Due to the successful applications of CNNs to 2D image analysis, several researchers decided to increase the dimensionality of the input by adding depth information as an additional channel to conform 2.5D CNNs.

Socher *et al.* [82] introduced a model based on a combination of CNNs and Recursive Neural Networks (RNNs) to learn features and classify RGB-D images. That model aims to learn low-level and translation invariant features with the CNN layers, those features are then given as inputs to fixed-tree RNNs to compose higher order features. Alexandre *et al.* [83] explore the possibility of transferring knowledge [84][85] between CNNs to improve accuracy and reducing training time when classifying RGB-D data. Hoeft *et al.* [86] proposed a four-stage CNN architecture, derived from the work of Schulz and Behnke [87], to semantically segment RGB-D scenes, providing the depth channel as feature maps representing components of a simplified histogram of oriented depth operator. Wang *et al.* [88] combined a CNN, to extract representative image features from RGB-D, with a Support Vector Machine (SVM) to classify objects in those images. Schwarz *et al.* [89] went one step beyond. They presented a system for object recognition and pose estimation using RGB-D images and transfer learning between a pre-trained CNN for image categorization and another CNN to classify colorized depth images. The features are then classified into instances and categories by SVMs and the pose is estimated via using another RBF kernel SVM.

In spite of the fact that those methods extend the traditional CNN, they do not employ a pure volumetric representation and therefore they do not make full use of the geometric information in the data. What is more, they do not use 3D convolutions. This is why they fall in the 2.5D CNNs category. In order to improve 2.5D CNNs, several authors proposed pure volumetric approaches or the so called 3D CNNs. These architectures apply spatially 3D convolutions fully utilizing geometric data.

The seminal work of Wu *et al.* [76] introduced a system that supports joint object recognition and shape completion from 2.5D depth maps that are transformed into a 3D shape representation which consists of a probability distribution of binary values on a 3D voxel grid. A Convolutional Deep Belief Network (CDBN) is used to recognize categories, complete 3D shapes, and predict next best views if the recognition is uncertain. Maturana and Scherer [78] proposed a 3D CNN for landing zone detection from LIDAR data. In that work, they also introduced a volumetric representation for that data using a density occupancy grid. Later, they extended that work creating VoxNet [90] a 3D CNN architecture for real-time object classification using volumetric occupancy grids to represent point clouds.

Other remarkable works are the multi-view system by Su *et al.* [91], the panoramic network by Shi *et al.* [92], and the orientation-based voxel nets by Sedaghat *et al.* [93].

5.3 Model-based CNN

In order to design, implement, and test our object recognition system, we are going to take a model-based approach which will be trained with full model clouds reconstructed from the extracted partial views. In that regard, we need two components: a proper 2.5D network architecture and data.

On the one hand, the network architecture of choice is shown in Figure 5.4. This network is a slightly tuned version of the one introduced in PointNet [94], which was inspired in VoxNet and 3D ShapeNets. It has been implemented using Caffe (see Section **??**). The model definition can be consulted in Appendix A.

On the other hand, we need data to train the network and validate the results. For this approach we will merge the point clouds generated from the partial views – as shown in Chapter 4 – to create full model clouds. That model clouds will be transformed into the corresponding volumetric representations to feed the CNN using a custom data layer implemented in Caffe.

5.3.1 Experimentation

In order to assess the performance of the proposed model-based CNN we carried out an extensive experimentation to determine the accuracy of the model and its robustness against occlusions and noise – situations that often occur in real-world scenes. For that purpose we started using the normalized density grids (see Chapter 4) since they offer a good balance between efficiency and representation. We also investigated the effect of both fixed and adaptive grids using different sizes. Further experimentation was performed to compare the normalized density grids with the binary ones.

The networks were trained for a maximum of 5000 iterations – weights were snapshotted every 100 iterations so in the end we selected the best sets of them as if we were early stopping – using Adadelta as optimizer with $\delta = 1 \cdot 10^{-8}$. The regularization term or weight decay in Caffe was set to $5 \cdot 10^{-3}$. A batch size of 32 training samples was chosen. The Caffe solver file which we used to train the networks can be consulted in Appendix A.

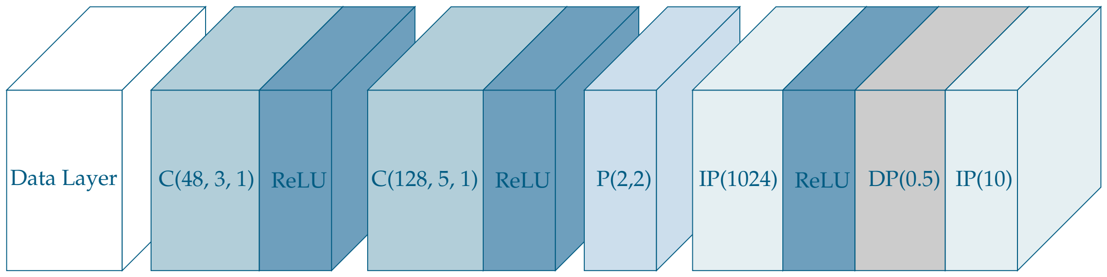


Figure 5.4: 2.5D Convolutional Neural Network architecture used for the experiments. This network is an extension of the one presented in PointNet [94]. It consists of a convolution layer – 48 filters, 3×3 filter with stride 1 –, a ReLU activation, another convolution layer – 128 filters, 5×5 filters with stride 1 –, followed by a ReLU activation, a pooling layer – 2×2 max. pooling with stride 2 –, a fully connected or inner product layer with 1024 neurons and ReLU activation, a dropout layer – 0.5 rate –, and an inner product layer with 10 neurons as output. The network accepts 3D tensors as input.

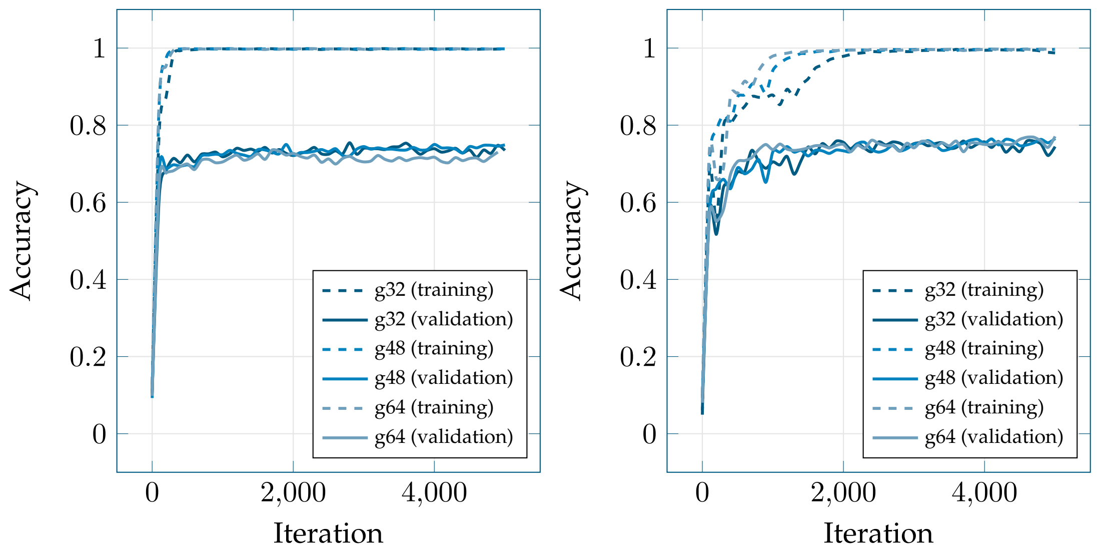
Data was divided into training and validation sets as provided by Modelnet10. The training set was shuffled upon generation. The validation set was processed to add different levels of noise and occlusions as shown in Chapter 4 in order to test the robustness of the network.

5.3.2 Results and Discussion

Figure 5.5 shows the accuracy results of the network for both grid types and increasing sizes. The peak accuracies for the fixed grids are ≈ 0.75, ≈ 0.76, and ≈ 0.73 for sizes 32, 48, and 64 respectively. In the case of the adaptive one, the peak accuracies are ≈ 0.77, ≈ 0.78, and ≈ 0.79 for the sizes 32, 48, and 64 respectively.

Taking those facts into account, we can extract two conclusions. First, the adaptive grid is able to achieve a slightly better peak accuracy in all cases; however, the fixed grid takes less iterations to reach accuracy values close to the peak in all cases. Second, there is no significant difference in using a bigger grid size of 64 voxels instead of a smaller one of 32.

The most important fact that can be observed in the aforementioned figures is that there is a considerable gap between training and validation accuracy in all situations. As we can observe, all networks reach maximum accuracy for the training set whilst the validation one hits a glass ceiling at approximately 0.80. We hypothesize that the network suffers overfitting even when we thoroughly applied measures to avoid that. The most probable cause for that problem is the reduced number of training examples. In the case of ModelNet10 the training set consists of only 3991 models. Considering the complexity of the CNN, it is reasonable to think that the lack of a richer training set is causing overfitting.



(a) Fixed Grid **(b)** Adaptive Grid

Figure 5.5: Evolution of training and validation accuracy of the model-based CNN using both fixed (a) and adaptive (b) normalized density grids. Different grid sizes ($32, 48$, and 64) were tested.

Another visualization format that would allow us to gain insight about the behavior of our network is the confusion matrix. Table 5.1 shows a confusion matrix of the validation results achieved by the best performing network in our experiments. As we can observe, there is a lot of confusion between desks and tables (see Figure 5.6). In addition, night stands get usually misclassified as dressers (see Figure 5.7). The other notable fact is that many sofas are misclassified as beds (see Figure 5.8). If we take a closer look at the confused classes it is reasonable to think that the network is not able to classify them properly because some samples are extremely similar.

Desk	Table	Nstand	Bed	Toil.	Dresser	Bath.	Sofa	Moni.	Chair
52	9	1	4	0	5	1	5	0	9
25	69	0	1	0	0	0	0	1	4
1	2	60	1	4	8	0	0	2	8
4	0	0	80	0	0	3	11	1	1
1	0	3	1	84	0	1	3	2	5
3	0	14	0	0	61	0	1	6	1
0	1	0	3	0	0	34	8	3	1
1	0	1	4	1	2	0	88	1	2
1	1	1	1	0	5	1	1	87	2
1	2	1	2	1	1	0	1	1	90

Table 5.1: Confusion matrix of the validation results achieved by the best set of weights for the adaptive grid with a grid size of 64 voxels. Darker cells indicate more predictions while lighter ones indicate less.

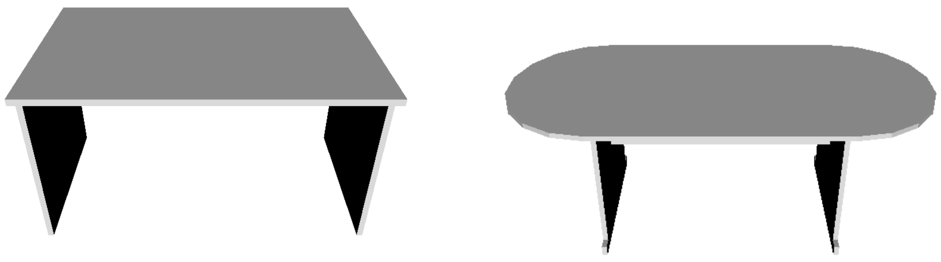


Figure 5.6: A desk class sample together with a table class one.

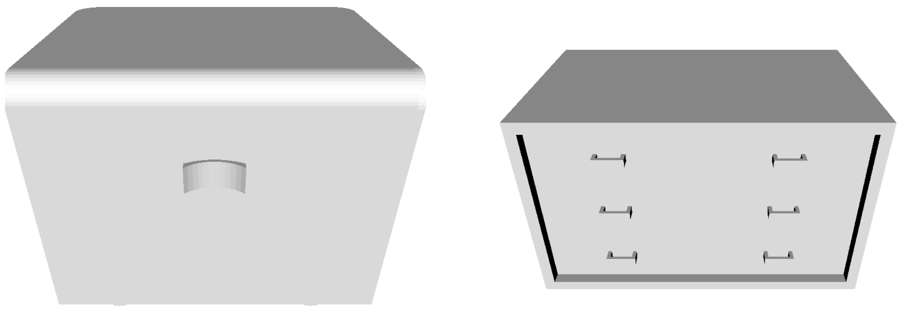


Figure 5.7: A night stand class sample together with a dresser one.

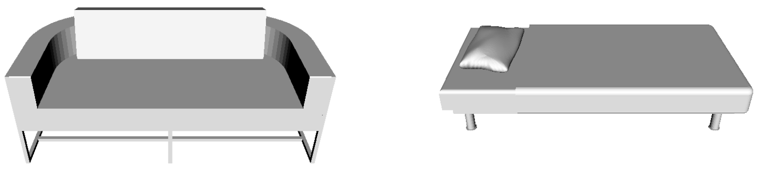
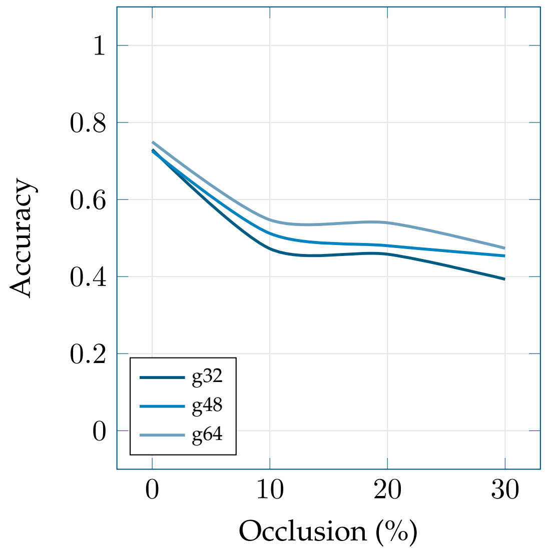
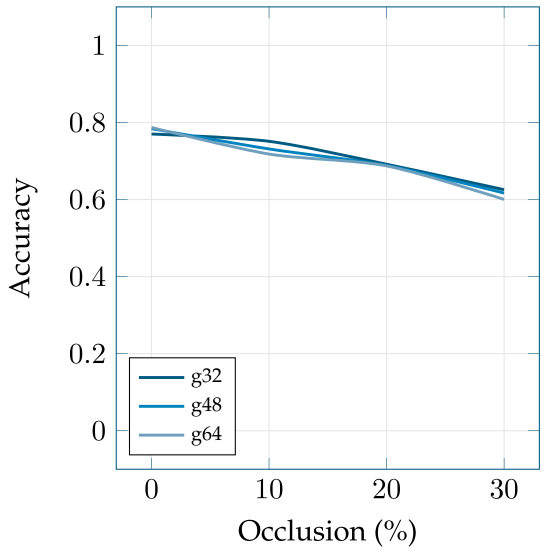


Figure 5.8: A sofa class sample together with a bed class one.

Occlusion Resilience Concerning the robustness against occlusion, we took the best networks after training and tested them using the same validation sets as before but introducing occlusions in them (up to a 30%). Figure 5.9 shows the accuracy of both grid types with different sizes as the amount of occlusion in the validation model increases. As we can observe, occlusion has a significant and negative impact on the fixed grid – bigger grid sizes are less affected – going down from ≈ 0.75 accuracy to $0.40 - 0.50$ approximately in the worst and best case respectively when a 30% of the model is occluded. On the contrary, the adaptive grid does not suffer that much – it goes down from ≈ 0.78 to ≈ 0.60 in the worst case – and there is no significant difference between grid sizes. In conclusion, the adaptive grid is considerably more robust to occlusion than the fixed one.

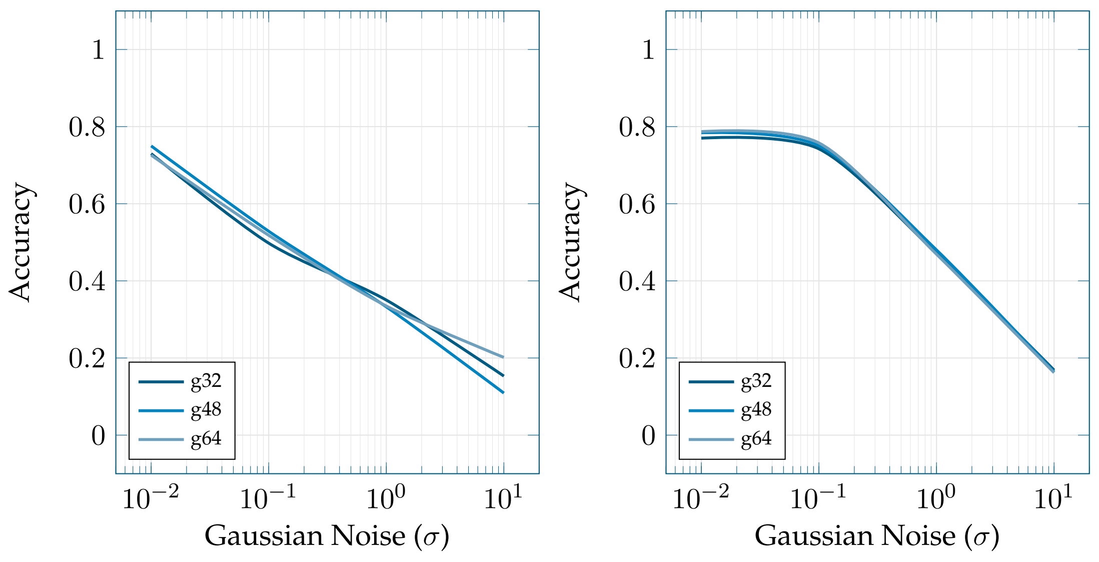


(a) Fixed Grid



(b) Adaptive Grid

Figure 5.9: Evolution of validation accuracy of the model-based CNN using both fixed (a) and adaptive (b) normalized density grids as the amount of occlusion in the validation models increases from 0% to 30%. Three grid sizes were tested ($32, 48,$ and 64).



(a) Fixed Grid **(b)** Adaptive Grid

Figure 5.10: Evolution of validation accuracy of the model-based CNN using both fixed (a) and adaptive (b) normalized density grids as the standard deviation of the Gaussian noise introduced in the z-axis of the views increases from 0.001 to 10. The common grid sizes were tested $(32, 48,$ and $64)$.

Noise Robustness Regarding the resilience to noise, we also tested the best networks obtained from the aforementioned training process using validation sets with different levels of noise (ranging from $\sigma = 1 \cdot 10^{-2}$ to $\sigma = 1 \cdot 10^{1}$). Figure 5.10b shows the results of those experiments. It can be observed that adding noise has a significant impact on the fixed grid, even small quantities, reducing the accuracy from ≈ 0.75 to ≈ 0.60, ≈ 0.4, and ≈ 0.2 for $\sigma = 1 \cdot 10^{-1}, \sigma = 1 \cdot 10^{0}$, and $\sigma = 1 \cdot 10^{1}$ respectively. On the other hand, the adaptive one shows remarkable robustness against low levels of noise (up to $\sigma = 1 \cdot 10^{-1}$), barely diminishing its accuracy.

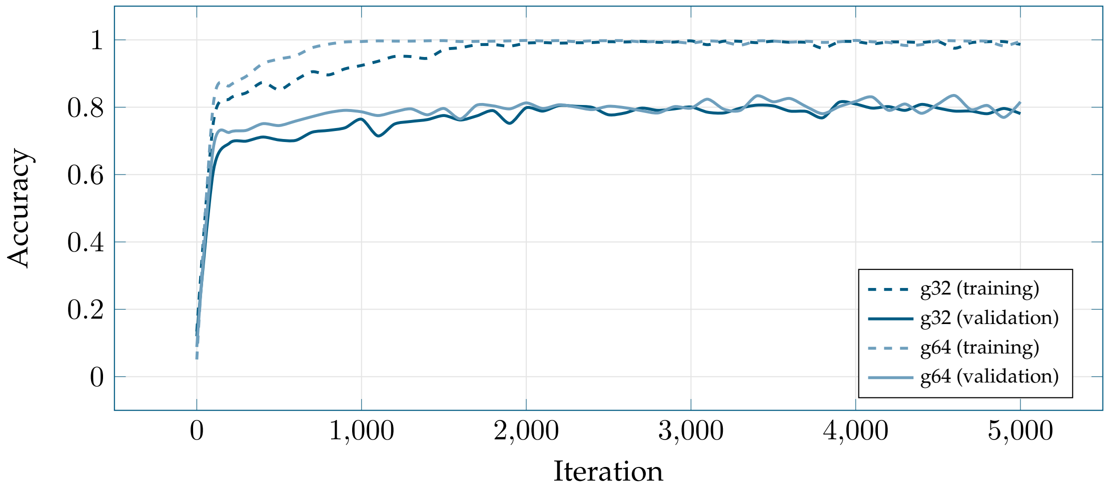
In the end, both grids suffer huge penalties in accuracy when noise levels higher than $\sigma = 1 \cdot 10^{-}1$ are introduced, being the adaptive one less affected. The grid size has little to no effect in both cases, only in the fixed grid bigger sizes are slightly more robust when intermediate to high levels of noise are introduced. In conclusion, the adaptive grid is significantly more resilient to low levels of noise, and slightly outperforms the fixed one when dealing with intermediate to high ones.

Binary Occupancy After testing the performance of the normalized density grid, we also trained and assessed the accuracy of the binary one in the same scenarios. This test intended to show whether there is any gain in using representations which include more information about the shape – at a small penalty to execution time.

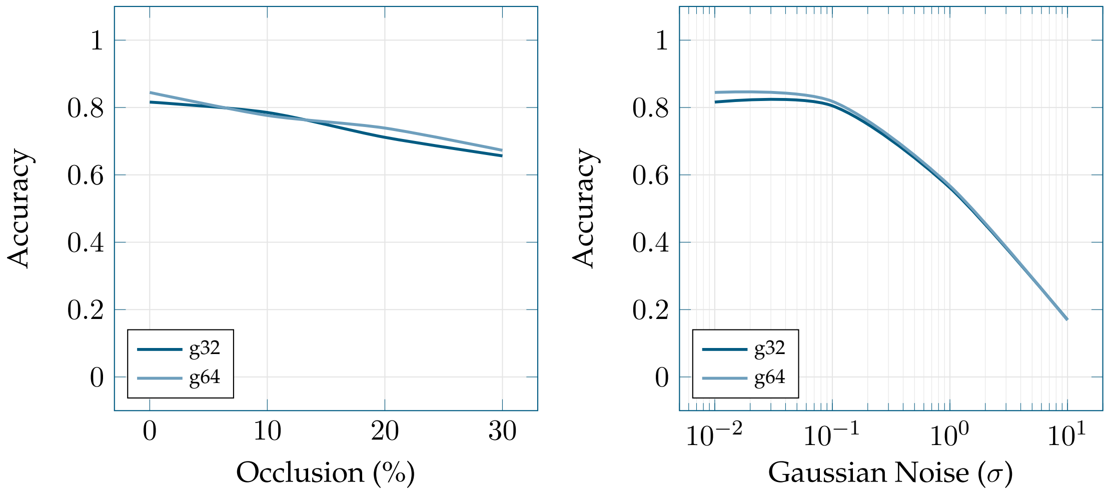
For this experimentation we picked the best performer in the previous sections: the adaptive grid. We also discarded the intermediate size (48 voxels) since there was no significant difference between it and the others. Figure 5.11a shows the accuracy results of the network trained using binary grids. As we can observe, there is no significant difference between grid sizes neither. However, using this representation we achieved a peak accuracy of approximately 0.85, using 64 voxels grids, which is better to some extent than the normalized density one shown in Figure 5.5.

Occlusion and noise tolerance (shown in Figures 5.11b and 5.11c respectively) is mostly similar to the robustness shown by the normalized density adaptive grid (see Figures 5.9b and 5.10b) except from a small offset caused by the higher accuracy of the binary grid network.

In conclusion, the less-is-better effect applies in this situation and turns out that the simplification introduced by the binary representation helps the network during the learning process. It is pending to check if this statement is still valid if the validation accuracy is not bounded by network overfitting.



(a) Training



(b) Occlusion **(c)** Noise

Figure 5.11: Evolution of training and validation accuracy of the model-based CNN using adaptive binary grids (a). Evolution of validation accuracy for the best network weights after training as the amount of occlusion in the validation set increases (b) and different levels of noise are introduced (c).

3D Convolutions At last, we extended Caffe to support pure 3D convolutions and pooling operations. In fact, this implementation is generalized to nD convolutions and pooling layers, so the system could be easily extended to support sequences of 3D inputs for instance. We kept the same network architecture shown in Figure 5.4 but extended its convolution and pooling layers to three dimensions. The model definition can be consulted in Appendix A.

We then trained the network using adaptive binary grids (since they were the ones with the best performance throughout the experimentation), and monitored the validation and training set accuracies during that process. The result is shown in Figure 5.12. As we can observe, the network does not perform as good as the 2.5D ones.

We trained the network for five times more iterations (25000) and the training accuracy slowly increased up to ≈ 0.65. However, the validation accuracy was stuck on ≈ 0.4 throughout the whole process.

In conclusion, porting the 2.5D network directly to 3D using the same datasets that produced good results in the former one does not achieve a comparable outcome in the latter. We hypothesize different causes for this problem.

On the one hand, the data representation might not be adequate for such fine-grained convolutions and bigger sizes or occupancy schemes might improve accuracy.

On the other hand, the complexity of the network skyrocketed after including that extra dimension to the convolutions, considerably increasing the number of weights and thus making the network harder to train with so few examples. This last assumption is backed up by the fact that the training accuracy keeps increasing while the validation one is stuck. This would eventually lead to a perfect fit to the training set but low accuracy on the validation examples. In other words, due to the excessive complexity of the network and the lack of training examples, the network suffers overfitting.

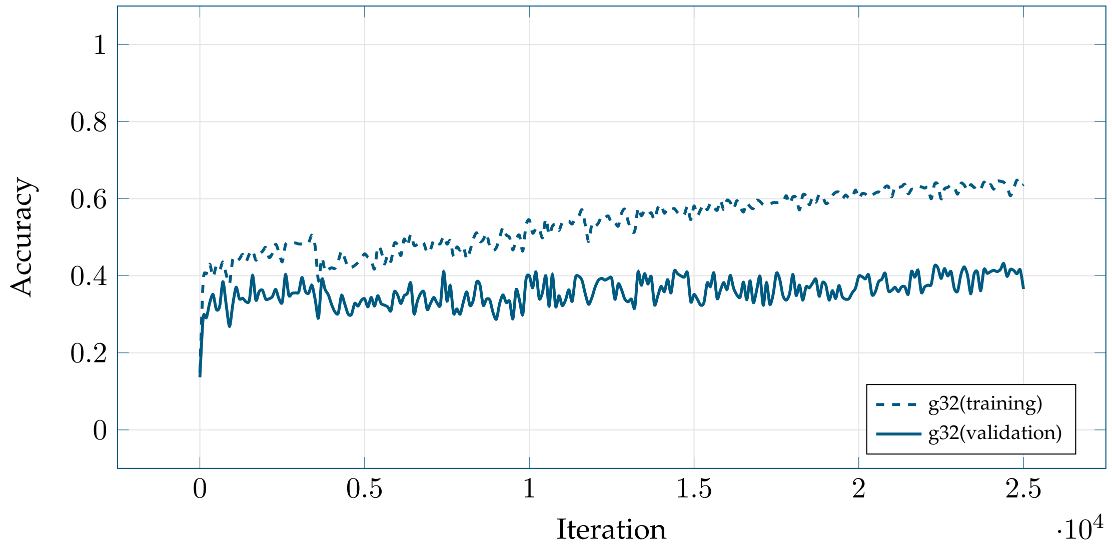


Figure 5.12: Evolution of training and validation accuracy of the model-based CNN with 3D convolutions, using adaptive binary grids with size $32 \times 32 \times 32$ voxels.

Grid Size	Fixed Density 2.5D	Adaptive Density 2.5D	Adaptive Binary 2.5D	Adaptive Binary 3D
$32 \times 32 \times 32$	0.75	0.77	0.80	0.43
$48 \times 48 \times 48$	0.76	0.78	N/A	N/A
$64 \times 64 \times 64$	0.73	0.79	0.85	N/A

Table 5.2: Summary of the experimentation results.

5.4 Conclusion

In this chapter we have discussed the differences between 2D, 2.5D, and 3D convolutions. We also reviewed state-of-the-art CNNs which make use of 2.5D and 3D convolutions. After that, we designed, implemented, and tested a 2.5D CNN using the volumetric representations proposed in Chapter 4. That network was trained and tested using fixed and adaptive grids with normalized density and binary occupancy measures. In addition, we tested the robustness of the network in adverse conditions by introducing occlusions and noise to the validation sets. After that, we extended the 2.5D network to 3D using the same data. A summary of the experimentation results is shown in Table 5.2.

To sum up, we determined that the adaptive grid slightly outperforms the fixed one in normal conditions. The same happens with the grid size, obtaining marginally better results with bigger sizes. However, when it comes down to noise and occlusion robustness, the adaptive grid exceeds the accuracy of the fixed grid by a large margin for low levels of occlusion and noise, whilst for intermediate and high levels the impact on both grids is somewhat similar. The grid size had barely any effect on those tests. In other words, the adaptive grid is better than the fixed one and it is preferable to use a bigger grid size if the performance impact can be afforded.

It is important to remark that the binary occupancy measure performed better than the normalized density one, both using adaptive grids, while maintaining similar resilience against noise and occlusions. The best network trained with normalized density grids reached a peak accuracy of ≈ 0.79 while the best binary one achieved approximately a 0.85 accuracy on the validation set.

Nevertheless, all networks exhibited a considerable amount of overfitting. This was due to the fact that the dataset has few training examples considering the complexity of the network and the classes are not completely distinguishable (for instance, many samples from the table class can be easily confused as desks). In this regard, the dataset must be augmented introducing noise, translations, rotations, and variations of the models to avoid overfitting and learn better those models that can be easily misclassified.

At last, we extended the network to support 3D convolutions and trained the same architecture as before using adaptive binary grids. The results were negative in the sense that the overfitting problem was accentuated due to the increased complexity of the network. By adding a third dimension to the convolutions, much more weights have to be learned, making the network prone to overfitting. In conclusion, despite the favorable properties of 3D convolutions, just extending a 2.5D network that works reasonably well, with a particular dataset, to perform 3D convolutions might not perform better as expected since training becomes harder.

Chapter 6

Conclusion

This chapter discusses the main conclusions extracted from the work presented in this Thesis. The chapter is organized in three sections: Section 6.1 presents and discusses the final conclusion of the work presented in this document. Next, Section 6.2 lists the publications derived from the presented work. Finally, Section 6.3 presents future works: open problems and topics that remain for further research.

6.1 Conclusions

In this Thesis, we first reviewed the state of the art of object recognition methods. After that study, we can state that deep learning is surpassing traditional pipelines based on hand-crafted descriptors. In addition, we observed that 3D-based methods usually outperform 2D-based ones thanks to the additional dimension of information. This review laid down the motivation for this work: explore the possibilities of 3D object class recognition using CNNs.

After that, we carried out a detailed study about the theoretical background of CNNs. We introduced the intuitive ideas behind that kind of network, and the differences between CNNs and fully connected ones. Furthermore, we described the basic architecture of a CNN and discussed training theory, considerations, and best practices that would be later applied in the implemented system.

As a prerequisite before diving deep into design and implementation phases, we conducted a study to select essential materials and methods for our work. In that regard, we selected ModelNet as the 3D object dataset to train our network, Caffe as the framework to develop our system, and we assembled and configured Asimov, a server to provide hardware support to this work.

Next, we discussed how to transform 3D data into volumetric representations with grid-like structures to feed CNNs. We reviewed existing proposals and stated what characterizes a good representation. After that study, we proposed several methods to represent 3D data as 3D tensors of real-valued components. In addition, we showed how to convert the dataset of choice to those representations simulating real-world sensor data. We also carried out a brief experimentation to analyze the efficiency and scaling of the proposed methods. On the one hand, there was no significant difference between the grid spawning methods. On the other hand, a tradeoff was detected between the information considered by the representation and time needed to compute it. We determined that the density-based representation offered a good balance in both terms, efficiency and quality.

At last, we described the difference between 2D, 2.5D, and 3D CNNs. We stated that a significant gain was expected from 3D convolutions with respect to 2.5D ones since the filters are also convolved in the depth dimension. We then designed, implemented, and tested a 2.5D CNN using the proposed volumetric representations. An in-depth study was carried out to determine their performance in adverse scenarios (noise and occlusions). The adaptive grid outperformed the fixed one, whilst the binary occupancy measure obtained better accuracy than the normalized density one – the surface triangulation method was excluded due to its prohibitive cost. In the end, a 2.5D CNN using adaptive binary grids of $64 \times 64 \times 64$ voxels achieved a 85% success rate when classifying object classes in the ModelNet10 dataset. In addition, that network showed significant robustness against low levels of noise and occlusion that characterize real-world scenarios. However, the final accuracy was limited due to overfitting. In the end, that successful network was extended to support 3D convolutions and trained again using the dataset with adaptive binary grids. The results showed that the 3D CNN was significantly harder to train than its 2.5D counterpart. In fact, its inherent complexity aggravated the overfitting problem so the network was not able to achieve good recognition rates.

6.2 Publications

The following publication was achieved as a result of the research carried out during the Master's Thesis:

A. Garcia-Garcia, F. Gomez-Donoso, J. Garcia-Rodriguez, S. OrtsEscolano, M. Cazorla, J. Azorin-Lopez. **PointNet: A 3D Convolutional Neural Network for Real-Time Object Class Recognition**. International Joint Conference on Neural Networks, IJCNN 2016, Vancouver, Canada.

6.3 Future Work

Due to the time constraints imposed on this project, many possible improvements and ideas were left out for future works. Here we summarize them to conclude this Thesis.

- The surface triangulation method was left out of the experimentation due to its prohibitive computational cost. Improving its efficiency by means of redesigning the algorithmic structure or developing a parallel implementation could allow us to experiment with that representation that might outperform the binary and density ones.
- In the same way, improving the efficiency of the binary and density representations with a parallel implementation using CUDA unlock its potential application to real-time problems.
- In order to avoid overfitting, the dataset must be augmented introducing noise, and variations of the existing models (rotations, translations, crops). Then the same architecture can be tested and check if that reduces overfitting.
- Preliminary experiments were carried out using a view-based 3D CNN instead of a model-based one. However, the initial results were not good enough. Determining the source of the problem for that approach and tackling it might provide better results than a model-based architecture. What is more, it could be directly applied to real-world data.
- The current system can be included in a full recognition pipeline, taking scenes from an RGB-D sensors, segmenting the objects, and applying the network to recognize them.

Appendix A

Caffe Models

A.1 Solver

```
train_net: "[training.prototxt]"
test_net: "[test.prototxt]"
test_net: "[train_test.prototxt]"
test_interval: 100
test_iter: 908
test_iter: 3991
base_lr: 1.0
weight_decay: 0.005
lr_policy: "fixed"
display: 100
max_iter: 25000
snapshot: 1000
snapshot_prefix: "[snapshots folder]"
solver_mode: GPU
solver_type: ADADELTA
delta: 1e-8
```

A.2 2.5D CNN Training Model

```
name: "PointNet"
layer{
  name: "data"
  type: "OCCData"
  occ_data_param {
    source : "[manifest tile]"
    batch_size: 32
    voxel_grid_size: [input grid size]
    leaf_size: [input leaf size]
    shuffle: 0
    occupancy: [occupancy type]
  }
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
}

layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 48
    kernel_size: 3
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}

layer {
name: "relu1"
type: "ReLU"
bottom: "conv1"
top: "conv1"
```

```
}

layer {
  name: "conv2"
  type: "Convolution"
  bottom: "conv1"
  top: "conv2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 128
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}

layer {
name: "relu2"
type: "ReLU"
bottom: "conv2"
top: "conv2"
}

layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv2"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}

layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "pool1"
  top: "ip1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 1024
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}

layer {
name: "relu3"
type: "ReLU"
bottom: "ip1"
top: "ip1"
}

layer {
  name: "drop1"
  type: "Dropout"
  bottom: "ip1"
  top: "drop1"
  dropout_param {
    dropout_ratio: 0.5
  }
}

layer {
  name: "ip2"
  type: "InnerProduct"
  bottom: "drop1"
  top: "ip2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 10
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}

layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip2"
  bottom: "label"
  top: "loss"
  loss_weight: 1
}
```

A.3 3D CNN Training Model

```
name: "PointNet"
layer{
  name: "data"
  type: "OCCData"
  occ_data_param {
    source : "[manifest tile]"
    batch_size: 32
    voxel_grid_size: [input grid size]
    leaf_size: [input leaf size]
    shuffle: 0
    occupancy: [occupancy type]
  }
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
}

layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 48
    kernel_size: 3
    kernel_size: 3
    kernel_size: 3
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}

layer {
name: "relu1"
type: "ReLU"
bottom: "conv1"
top: "conv1"
}

layer {
  name: "conv2"
  type: "Convolution"
  bottom: "conv1"
  top: "conv2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 128
    kernel_size: 5
    kernel_size: 5
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}

layer {
name: "relu2"
type: "ReLU"
bottom: "conv2"
top: "conv2"
}

layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv2"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
    kernel_size: 2
    kernel_size: 2
    stride: 2
  }
}
```

```
layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "pool1"
  top: "ip1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 1024
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}

layer {
name: "relu3"
type: "ReLU"
bottom: "ip1"
top: "ip1"
}

layer {
  name: "drop1"
  type: "Dropout"
  bottom: "ip1"
  top: "drop1"
  dropout_param {
    dropout_ratio: 0.5
  }
}

layer {
  name: "ip2"
  type: "InnerProduct"
  bottom: "drop1"
  top: "ip2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 10
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}

layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip2"
  bottom: "label"
  top: "loss"
  loss_weight: 1
}
```

Appendix B

Source Code

The source code of this Master's Thesis is available online in *GitHub* [1] licensed under the GNU General Public License v3.0 (GNU GPLv3), the most widely used free software license with a strong copyleft requirement. The repository holds a C++ project organized as follows.

```
masterthesis-src
  ├─ include
  ├─ scripts
  ├─ src
  ├─ CMakeLists.txt
  ├─ README.txt
  └─ .gitignore
```

B.1 Dependencies

The following dependencies must be met in order to compile the project:

- g++ ≥ 4.8
- CMake ≥ 2.6
- Boost ≥ 1.5
- Point Cloud Library ≥ 1.7
- Vtk ≥ 5.8

B.2 Compilation

The project can be compiled using CMake and the provided `CMakeLists.txt` file as follows (assuming that the current working directory `pwd` is the root directory of the project):

```
mkdir build
cd build
cmake ..
make -jX
```

[1] https://github.com/Blitzman/masterthesis-src

Bibliography

[1] Alberto Garcia-Garcia. *Towards a real-time 3D object recognition pipeline on mobile GPGPU computing platforms using low-cost RGB-D sensors*. Bachelor Thesis. 2015.

[2] Alberto Garcia-Garcia, Sergio Orts-Escolano, Jose Garcia-Rodriguez, et al. "Interactive 3D object recognition pipeline on mobile GPGPU computing platforms using low-cost RGB-D sensors". In: *Journal of Real-Time Image Processing* (2016).

[3] Miguel Cazorla, José Garcıa-Rodrıguez, José Marıa Canas Plaza, et al. "SIRMAVED: Development of a comprehensive monitoring and interactive robotic system for people with acquired brain damage and dependent people". In: ().

[4] Alexander Andreopoulos and John K. Tsotsos. "50 Years of object recognition: Directions forward". In: *Computer Vision and Image Understanding* 117.8 (2013), pp. 827–891.

[5] David G. Lowe. "Distinctive image features from scale-invariant keypoints". In: *International journal of computer vision* 60.2 (2004), pp. 91–110.

[6] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. "Surf: Speeded up robust features". In: *Computer vision–ECCV 2006*. Springer, 2006, pp. 404–417.

[7] Michael Calonder, Vincent Lepetit, Christoph Strecha, et al. "Brief: Binary robust independent elementary features". In: *Computer Vision–ECCV 2010* (2010), pp. 778–792.

[8] Stefan Leutenegger, Margarita Chli, and Roland Y. Siegwart. "BRISK: Binary robust invariant scalable keypoints". In: *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE. 2011, pp. 2548–2555.

[9] Ethan Rublee, Vincent Rabaud, Kurt Konolige, et al. "ORB: an efficient alternative to SIFT or SURF". In: *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE. 2011, pp. 2564–2571.

[10] Alexandre Alahi, Raphael Ortiz, and Pierre Vandergheynst. "FREAK: Fast retina keypoint". In: *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. Ieee. 2012, pp. 510–517.

[11] David G. Lowe. "Object recognition from local scale-invariant features". In: *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*. Vol. 2. Ieee. 1999, pp. 1150–1157.

[12] Yulan Guo, Mohammed Bennamoun, Ferdous Sohel, et al. "3D object recognition in cluttered scenes with local surface features: A survey". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 36.11 (2014), pp. 2270–2287.

[13] Jean Ponce, Svetlana Lazebnik, Fredrick Rothganger, et al. "Toward true 3D object recognition". In: *Reconnaissance de Formes et Intelligence Artificielle*. 2004.

[14] Paul J. Besl and Ramesh C. Jain. "Three-dimensional object recognition". In: *ACM Computing Surveys (CSUR)* 17.1 (1985), pp. 75–145.

[15] Jim P. Brady, Nagaraj Nandhakumar, and Jake K. Aggarwal. “Recent progress in object recognition from range data”. In: *image and vision computing* 7.4 (1989), pp. 295–307.

[16] Farshid Arman and Jake K. Aggarwal. “Model-based object recognition in dense-range imagesa review”. In: *ACM Computing Surveys (CSUR)* 25.1 (1993), pp. 5–43.

[17] Richard J. Campbell and Patrick J. Flynn. “A survey of free-form object representation and recognition techniques”. In: *Computer Vision and Image Understanding* 81.2 (2001), pp. 166–210.

[18] George Mamic and Mohammed Bennamoun. “Representation and recognition of 3D free-form objects”. In: *Digital Signal Processing* 12.1 (2002), pp. 47–76.

[19] Yann Le Cun, Yoshua Bengio, and Geoffrey E. Hinton. “Deep learning”. In: *Nature* 521.7553 (2015), pp. 436–444.

[20] Paul J. Werbos. “Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences”. PhD thesis. Harvard University, 1974.

[21] Yann Le Cun. “A learning scheme for asymmetric threshold networks”. In: *Proceedings of Cognitiva* 85 (1985), pp. 599–604.

[22] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors”. In: *Cognitive modeling* 5 (1988), p. 3.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[24] Stjepan Marčelja. “Mathematical description of the responses of simple cortical cells”. In: *JOSA* 70.11 (1980), pp. 1297–1300.

[25] Alex Berg, Jia Deng, and Fei-Fei Li. *ImageNet large scale visual recognition challenge 2010*. 2010.

[26] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. “A fast learning algorithm for deep belief nets”. In: *Neural computation* 18.7 (2006), pp. 1527–1554.

[27] Yoshua Bengio, Pascal Lamblin, Dan Popovici, et al. “Greedy layer-wise training of deep networks”. In: *Advances in neural information processing systems* 19 (2007), p. 153.

[28] Pierre Sermanet, Koray Kavukcuoglu, Sandhya Chintala, et al. “Pedestrian detection with unsupervised multi-stage feature learning”. In: *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*. IEEE. 2013, pp. 3626–3633.

[29] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. “Large-scale deep unsupervised learning using graphics processors”. In: *Proceedings of the 26th annual international conference on machine learning*. ACM. 2009, pp. 873–880.

[30] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. “Deep Learning”. Book in preparation for MIT Press. 2016. URL: `http://www.deeplearningbook.org`.

[31] Balázs Csanád Csáji. “Approximation with artificial neural networks”. In: *Faculty of Sciences, Etvs Lornd University, Hungary* 24 (2001), p. 48.

[32] Bing Xu, Naiyan Wang, Tianqi Chen, et al. “Empirical evaluation of rectified activations in convolutional network”. In: *arXiv preprint arXiv:1505.00853* (2015).

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, et al. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2015, pp. 1026–1034.

[34] Benjamin Graham. "Fractional max-pooling". In: *arXiv preprint arXiv:1412.6071* (2014).

[35] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, et al. "Striving for simplicity: The all convolutional net". In: *arXiv preprint arXiv:1412.6806* (2014).

[36] Steve Lawrence, Lee C. Giles, and Ah Chung Tsoi. "Lessons in neural network training: Overfitting may be harder than expected". In: *AAAI/IAAI*. Citeseer. 1997, pp. 540–545.

[37] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, et al. "Dropout: A simple way to prevent neural networks from overfitting". In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[38] Hui Zou and Trevor Hastie. "Regularization and variable selection via the elastic net". In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 67.2 (2005), pp. 301–320.

[39] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *International conference on artificial intelligence and statistics*. 2010, pp. 249–256.

[40] Dumitru Erhan, Yoshua Bengio, Aaron Courville, et al. "Why does unsupervised pre-training help deep learning?" In: *The Journal of Machine Learning Research* 11 (2010), pp. 625–660.

[41] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[42] Philipp Krähenbühl, Carl Doersch, Jeff Donahue, et al. "Data-dependent Initializations of Convolutional Neural Networks". In: *arXiv preprint arXiv:1511.06856* (2015).

[43] *Why are deep neural networks hard to train?* `http://neuralnetworksanddeeplearning.com/chap5.html`. Accessed: 09-05-2016.

[44] Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, et al. "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization". In: *Advances in neural information processing systems*. 2014, pp. 2933–2941.

[45] Christian Darken, Joseph Chang, and John Moody. "Learning rate schedules for faster stochastic gradient search". In: *Neural networks for signal processing*. Vol. 2. Citeseer. 1992.

[46] Ning Qian. "On the momentum term in gradient descent learning algorithms". In: *Neural networks* 12.1 (1999), pp. 145–151.

[47] Yurii Nesterov. "A method of solving a convex programming problem with convergence rate O (1/k2)". In: *Soviet Mathematics Doklady*. Vol. 27. 2. 1983, pp. 372–376.

[48] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *The Journal of Machine Learning Research* 12 (2011), pp. 2121–2159.

[49] Matthew D. Zeiler. "ADADELTA: an adaptive learning rate method". In: *arXiv preprint arXiv:1212.5701* (2012).

[50] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural Networks for Machine Learning* 4 (2012), p. 2.

[51] Diederik Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[52] *ConvNetJS trainer demo on MNIST*. http://cs.stanford.edu/people/karpathy/convnetjs/demo/trainers.html. Accessed: 07-05-2016.

[53] Olga Russakovsky, Jia Deng, Hao Su, et al. "Imagenet large scale visual recognition challenge". In: *International Journal of Computer Vision* 115.3 (2015), pp. 211–252.

[54] Ronan Collobert and Jason Weston. "A unified architecture for natural language processing: Deep neural networks with multitask learning". In: *Proceedings of the 25th international conference on Machine learning*. ACM. 2008, pp. 160–167.

[55] Ronan Collobert, Jason Weston, Léon Bottou, et al. "Natural language processing (almost) from scratch". In: *The Journal of Machine Learning Research* 12 (2011), pp. 2493–2537.

[56] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Domain adaptation for large-scale sentiment classification: A deep learning approach". In: *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. 2011, pp. 513–520.

[57] Li Deng, Geoffrey Hinton, and Brian Kingsbury. "New types of deep neural network learning for speech recognition and related applications: An overview". In: *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE. 2013, pp. 8599–8603.

[58] Li Deng, Jinyu Li, Jui-Ting Huang, et al. "Recent advances in deep learning for speech research at Microsoft". In: *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE. 2013, pp. 8604–8608.

[59] Alan Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. "Speech recognition with deep recurrent neural networks". In: *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE. 2013, pp. 6645–6649.

[60] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. "Torch7: A matlab-like environment for machine learning". In: *BigLearn, NIPS Workshop*. EPFL-CONF-192376. 2011.

[61] James Bergstra, Frédéric Bastien, Olivier Breuleux, et al. "Theano: Deep learning on gpus with python". In: *NIPS 2011, BigLearning Workshop, Granada, Spain*. 2011.

[62] Yangqing Jia, Evan Shelhamer, Jeff Donahue, et al. "Caffe: Convolutional architecture for fast feature embedding". In: *Proceedings of the ACM International Conference on Multimedia*. ACM. 2014, pp. 675–678.

[63] Amit Agarwal, Eldar Akchurin, Chris Basoglu, et al. *An Introduction to Computational Networks and the Computational Network Toolkit*. Tech. rep. MSR-TR-2014-112. 2014. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=226641.

[64] Martın Abadi, Ashish Agarwal, Paul Barham, et al. "TensorFlow: Large-scale machine learning on heterogeneous systems, 2015". In: *Software available from tensorflow. org* (2015).

[65] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, et al. "cudnn: Efficient primitives for deep learning". In: *arXiv preprint arXiv:1410.0759* (2014).

[66] James Bergstra, Olivier Breuleux, Frédéric Bastien, et al. "Theano: a CPU and GPU Math Expression Compiler". In: *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Oral Presentation. Austin, TX, June 2010.

[67] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, et al. *Theano: new features and speed improvements*. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop. 2012.

[68] Stefan Van Der Walt, Chris S. Colbert, and Gael Varoquaux. "The NumPy array: a structure for efficient numerical computation". In: *Computing in Science & Engineering* 13.2 (2011), pp. 22–30.

[69] Eric Jones, Travis Oliphant, and Pearu Peterson. "{SciPy}: open source scientific tools for {Python}". In: (2014).

[70] Andrea Vedaldi and Karel Lenc. "MatConvNet: Convolutional neural networks for matlab". In: *Proceedings of the 23rd Annual ACM Conference on Multimedia Conference*. ACM. 2015, pp. 689–692.

[71] Kevin Lai, Liefeng Bo, Xiaofeng Ren, et al. "A large-scale hierarchical multi-view rgb-d object dataset". In: *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE. 2011, pp. 1817–1824.

[72] Ashutosh Singh, Jin Sha, Karthik S. Narayan, et al. "Bigbird: A large-scale 3d database of object instances". In: *Robotics and Automation (ICRA), 2014 IEEE International Conference on*. IEEE. 2014, pp. 509–516.

[73] Bo Li, Yijuan Lu, Chunyuan Li, et al. "Shrec14 track: extended large scale sketch-based 3D shape retrieval". In: *Eurographics Workshop on 3D Object Retrieval*. 2014, pp. 121–130.

[74] Sungjoon Choi, Qian-Yi Zhou, Stephen Miller, et al. "A Large Dataset of Object Scans". In: *CoRR* abs/1602.02481 (2016).

[75] *DIGITS DevBox: User Guide*. `http://docs.nvidia.com/deeplearning/pdf/DIGITS_DEVBOX_User_Guide.pdf`. Accessed: 19-05-2016.

[76] Zhirong Wu, Shuran Song, Aditya Khosla, et al. "3D ShapeNets: A Deep Representation for Volumetric Shape Modeling". In: *Proc. CVPR, to appear*. Vol. 1. 2. 2015, p. 3.

[77] Shuran Song and Jianxiong Xiao. "Deep Sliding Shapes for Amodal 3D Object Detection in RGB-D Images". In: *CoRR* abs/1511.02300 (2015). URL: `http://arxiv.org/abs/1511.02300`.

[78] Daniel Maturana and Sebastian Scherer. "3D Convolutional Neural Networks for Landing Zone Detection from LiDAR". In: ICRA. 2015.

[79] Michael Gschwandtner, Roland Kwitt, Andreas Uhl, et al. "BlenSor: blender sensor simulation toolbox". In: *Advances in Visual Computing*. Springer, 2011, pp. 199–208.

[80] Patrice Y. Simard, Dave Steinkraus, and John C. Platt. "Best practices for convolutional neural networks applied to visual document analysis". In: *null*. IEEE. 2003, p. 958.

[81] Zoltan C. Marton, Radu B. Rusu, and Michael Beetz. “On Fast Surface Reconstruction Methods for Large and Noisy Datasets”. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. Kobe, Japan, 2009.

[82] Richard Socher, Brody Huval, Bharath Bath, et al. “Convolutional-recursive deep learning for 3d object classification”. In: *Advances in Neural Information Processing Systems*. 2012, pp. 665–673.

[83] Luís A. Alexandre. “3D object recognition using convolutional neural networks with transfer learning between input channels”. In: *Proc. the 13th International Conference on Intelligent Autonomous Systems*. 2014.

[84] Sinno Jialin Pan and Qiang Yang. “A survey on transfer learning”. In: *Knowledge and Data Engineering, IEEE Transactions on* 22.10 (2010), pp. 1345–1359.

[85] Dan C. Cireşan, Ueli Meier, and Jürgen Schmidhuber. “Transfer learning for Latin and Chinese characters with deep neural networks”. In: *Neural Networks (IJCNN), The 2012 International Joint Conference on*. IEEE. 2012, pp. 1–6.

[86] Nico Höft, Hannes Schulz, and Sven Behnke. “Fast Semantic Segmentation of RGB-D Scenes with GPU-Accelerated Deep Neural Networks”. In: *KI 2014: Advances in Artificial Intelligence*. Springer, 2014, pp. 80–85.

[87] Hannes Schulz and Sven Behnke. “Learning Object-Class Segmentation with Convolutional Neural Networks.” In: *ESANN*. 2012.

[88] Jianhua Wang, Jinjin Lu, Weihai Chen, et al. “Convolutional neural network for 3D object recognition based on RGB-D dataset”. In: *Industrial Electronics and Applications (ICIEA), 2015 IEEE 10th Conference on*. IEEE. 2015, pp. 34–39.

[89] Max Schwarz, Hannes Schulz, and Sven Behnke. “RGB-D object recognition and pose estimation based on pre-trained convolutional neural network features”. In: *Robotics and Automation (ICRA), 2015 IEEE International Conference on*. IEEE. 2015, pp. 1329–1335.

[90] Daniel Maturana and Sebastian Scherer. “VoxNet: A 3D Convolutional Neural Network for Real-Time Object Recognition”. In: (2015).

[91] Hang Su, Subhransu Maji, Evangelos Kalogerakis, et al. “Multi-view convolutional neural networks for 3d shape recognition”. In: *Proc. ICCV*. 2015.

[92] Baoguang Shi, Song Bai, Zhichao Zhou, et al. “DeepPano: Deep panoramic representation for 3-D shape recognition”. In: *Signal Processing Letters, IEEE* 22.12 (2015), pp. 2339–2343.

[93] Nima Sedaghat, Mohammadreza Zolfaghari, and Thomas Brox. “Orientation-boosted Voxel Nets for 3D Object Recognition”. In: *arXiv preprint arXiv:1604.03351* (2016).

[94] Alberto Garcia-Garcia, Francisco Gomez-Donoso, Jose Garcia-Rodriguez, et al. “PointNet: A 3D Convolutional Neural Network for Real-Time Object Class Recognition”. In: *Neural Networks (IJCNN), The 2016 International Joint Conference on*. IEEE. 2016.