Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Processing of 3D Scans using Machine Learning

Diplomová Práca

2021
Lukáš Gajdošech

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Processing of 3D Scans using Machine Learning

Diplomová Práca

Študijný program:	Aplikovaná Informatika
Študijný odbor:	2511 Aplikovaná Informatika
Školiace pracovisko:	Katedra aplikovanej informatiky
Školiteľ:	RNDr. Martin Madaras, PhD.

Bratislava, 2021
Lukáš Gajdošech

26157336

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Lukáš Gajdošech
Study programme:	Applied Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Processing of 3D Scans using Machine Learning

Annotation: Neural networks have been found as very efficient machine learning tool for image processing tasks. During the 3D scanning depth maps are created. Having a known and calibrated camera, the depth maps can be converted into a structured point cloud. Structured point clouds are similar as standard frame, they are 2D matrices of points, where every sample carries the 3D position information. The structured point clouds have the same structure as the 2D image data, with the additional 3D position information, thus the neural networks designed for 2D image processing can be used for these 3D data.

Aim: The goal of the thesis is to analyze existing neural network models that are used in image processing or signal processing in general and to propose an effective modifications, how to use these approaches to point cloud processing. Compare proposed models with the existing ones and evaluate the results. Apply proposed models for 3D image processing tasks as filtering, segmentation etc. and use them in post-processing pipeline of 3D scanner data.

Literature: (C++, python, tensorflow, neural network)
1. PlaneRCNN: 3D Plane Detection and Reconstruction from a Single Image, Chen Liu, CVPR 2019
2. Segmentation-driven 6D Object Pose Estimation, Yinlin Hu, CVPR 2019

Supervisor:	RNDr. Martin Madaras, PhD.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.
Assigned:	25.09.2017
Approved:	08.10.2019

prof. RNDr. Roman Ďurikovič, PhD.
Guarantor of Study Programme

Student

Supervisor

26157336

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Bc. Lukáš Gajdošech
Študijný program:	aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor:	informatika
Typ záverečnej práce:	diplomová
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Processing of 3D Scans using Machine Learning
Spracovanie 3D skenov pomocou strojového učenia

Anotácia: Neurónové siete sa v posledných rokoch ukázali ako vhodný a veľmi efektívny nástroj strojového učenia na spracovanie obrazových dát. Pri 3D skenovaní objektov nám vznikajú snímky, ktoré sa nazývajú hĺbkové mapy. Pri známej kamere, vieme tieto hĺbkové mapy priamo konvertovať do tzv. usporiadaných mračien bodov. Usporiadané mračná bodov sú, podobné ako štandardné snímky, 2D matice bodov, kde každá vzorka nesie informáciu o svojej 3D pozícii. Keďže usporiadané mračná bodov majú štruktúru ako 2D obrazové dáta, akurát každý prvok, okrem iného, nesie informáciu o 3D pozícii, neurónové siete na spracovanie 2D obrazu môžu byť použité na tieto 3D dáta.

Cieľ: Cieľom práce je analyzovať existujúce modely neurónových sietí, ktoré sa štandardne používajú na spracovanie obrazu alebo vo všeobecnosti na spracovanie signálu a navrhnúť efektívne modifikácie, ako tieto prístupy použiť na spracovanie mračien bodov. Vyhodnoťte navrhnuté modely a porovnajte ich s existujúcimi alternatívami. Aplikujte navrhnuté modely na úlohy spracovania 3D obrazu ako filtrovanie, segmentácia a ďalšie a použite tieto modely v metódach na spracovanie dát z 3D skenerov.

Literatúra: (C++, python, tensorflow, neural network)
1. PlaneRCNN: 3D Plane Detection and Reconstruction from a Single Image, Chen Liu, CVPR 2019
2. Segmentation-driven 6D Object Pose Estimation, Yinlin Hu, CVPR 2019

Vedúci:	RNDr. Martin Madaras, PhD.
Katedra:	FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry:	prof. Ing. Igor Farkaš, Dr.
Dátum zadania:	25.09.2017

Dátum schválenia: 08.10.2019 prof. RNDr. Roman Ďurikovič, PhD.
garant študijného programu

....
študent vedúci práce

Čestné prehlásenie: Čestne prehlasujem, že túto diplomovú prácu som vypracoval samostatne len s použitím uvedenej literatúry a za pomoci konzultácií s mojím školiteľom.

. . . .

Bratislava, 2021 Lukáš Gajdošech

Poďakovanie: Som vďačný školiteľovi Martinovi Madarasovi a aj ďalším členom firmy Skeletex za ich pomoc počas vývoja tejto práce. Ďalej by som chcel poďakovať aj všetkým priateľom, rodine, mame a milujúcej partnerke za nekonečnú podporu.

Abstract

Abstract

Keywords: machine learning

Abstrakt

Abstrakt

Kľúčové slová: strojové učenie

Contents

Chapter 1

Introduction

For an access to the repository, contact the author at `l.gajdosech@gmail.com`.

Chapter 2

Motivation

Chapter 3

Theoretical Overview

In this chapter, we briefly go over the theoretical definition of the problems we are going to solve and the required methods and techniques.

3.1 Analytical solutions

The problems we solve using the tools of *deep learning* were by large part decided by the availability of training data. Training of the network for the filtration of artefacts was conditioned on the existence of an analytical solution, providing data for supervised learning.

3.1.1 Artefact Filtering

During the process of scanning by a structured light scanner, several artefacts tend to appear. These are usually caused by the reflectance of the material surface, or various lightning conditions. We used an existing analytical solution developed by Skeletex [1]. This algorithm needs a sequence of scans of the object with rich overlaps, as it is build upon this redundancy. In other words, if a given point appears at a certain location only in a single scan out of several aligned scans, which have overlapping parts in this region, it is probably just an artefact.

On the other hand, machine learning model developed in this work is able to perform a filtration of a single-view scan. This ability is granted by the learning process, where it captures the probability distribution of artefacts, given their feature characteristics. This provides a huge advantage in special use-cases over the traditional approach. For example, we are able to filter the frames from a continuous 3D camera in real-time.

[1] http://skeletex.xyz

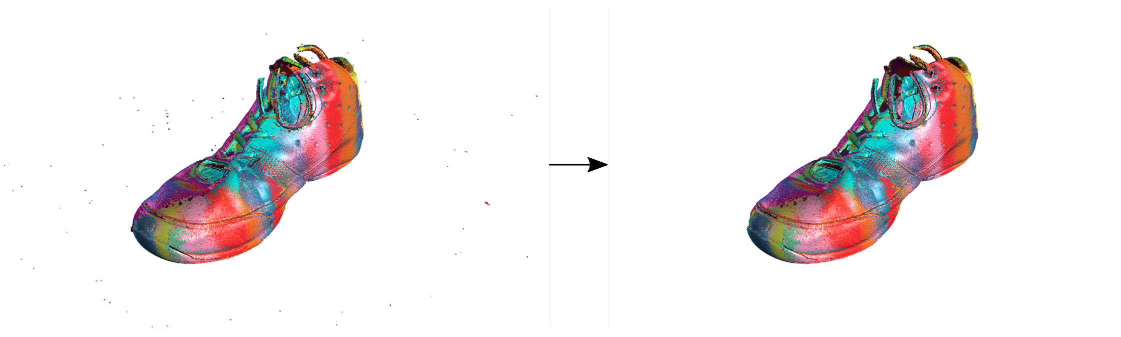
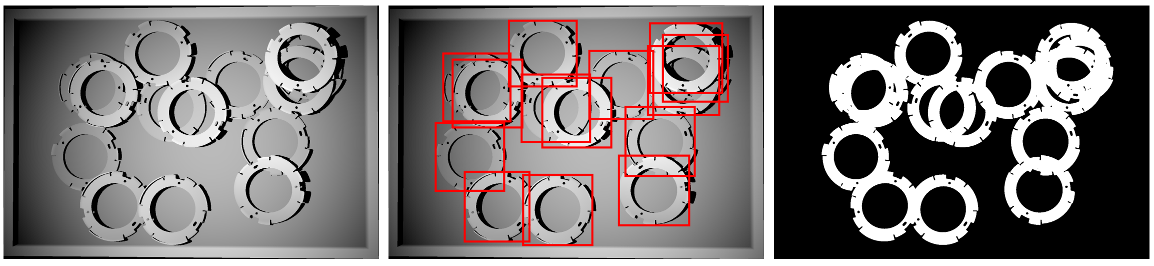


Figure 3.1: Exampled of a 3D point cloud, filtered by an analytical algorithm.

3.1.2 Localization and Segmentation

Localizing objects, marking their bounding boxes and drawing their pixel-wise segmentation masks can be done manually. However, this is quite a laborous process and we currently do not have an analytical algorithm, that could provide a robust ground truth for these problems. Thus we choose a different approach here.

We have an access to *BinGenerator* utility, that simulates the work of a light-structured scanner and produces synthetic scans of bins filled with CAD parts [2]. Apart from 3D organized point clouds (depth maps), it also outputs a point-wise class assignment. This can be used as an mask for different segmentation problems, such as a binary segmentation between the parts and background (bin), or instance segmentation of the individual parts. We can derive axis aligned bounding boxes and begin by learning this easier task. Our goal is to train models on these synthetic data-sets and design them in such a way, that they are able to generalize to real-world scans. See Figure 3.2 for an example of generated depth image and the expected ground truth results. *BinGenerator* outputs even more data, such as a precise transformation for every part in the bin. This opens the possibility to train models even for harder tasks, such as for 6DOF localization in the future.



(a) depth map (b) bounding boxes (c) segmentation mask

Figure 3.2: Synthetic scan with GT for BB localization and binary segmentation.

[2]https://deep-geometry.github.io/abc-dataset/

3.2 Machine Learning

At the core of this work lies *convolutional neural networks*, which belong to the class of deep learning models. Additionally, deep learning as a whole is part of machine learning. In machine learning we are trying to make the computer *learn* specific functions to perform some task and in deep learning we take this one step further by combining various nonlinear transformations, stacking them on each other in form of layers and thus producing algorithms with greater representational capacity [13]. However, this also increases the variance in the hypothesis space and *training* of such algorithms is often more complicated, time consuming and requires a lot of data [5].

In machine learning, we design an learning algorithm, which than tries to find patterns that generalize to new data. This is essentially a form of applied statistics, where we use the computer to statistically estimate complicated functions [7]. Usually, we optimize the parameters of these parameters using gradient descent. But how to define an algorithm that learns from data and what exactly is learning? Definition by Mitchell [16] is commonly used:

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

In this work, we use methods of *supervised learning*, where the experience E takes a form of an example, which is usually just some vector of features. We define the task T by providing ground truth data - i.e. the expected result and than the algorithm tries to optimize itself to solve this task in a best way possible, according to a performance measure P, which often takes a form of the loss function.

3.2.1 Supervised learning

Let x denote the input example, which can be just a single value or more often a vector of features, or a whole multi-dimensional volume, as in the case of images. In supervised learning, we provide the ground truth value y, which is the target that we are trying to predict. This can be something simple, such as a number denoting a class to which the x belongs. The *artefact cleaning* problem is example of such binary classification, which we can model as a prediction for every point x, whether it is an artefact or true geometry. However, the target can be something more complicated, such as a set of bounding box coordinates in the case of object detection in the whole image x [20].

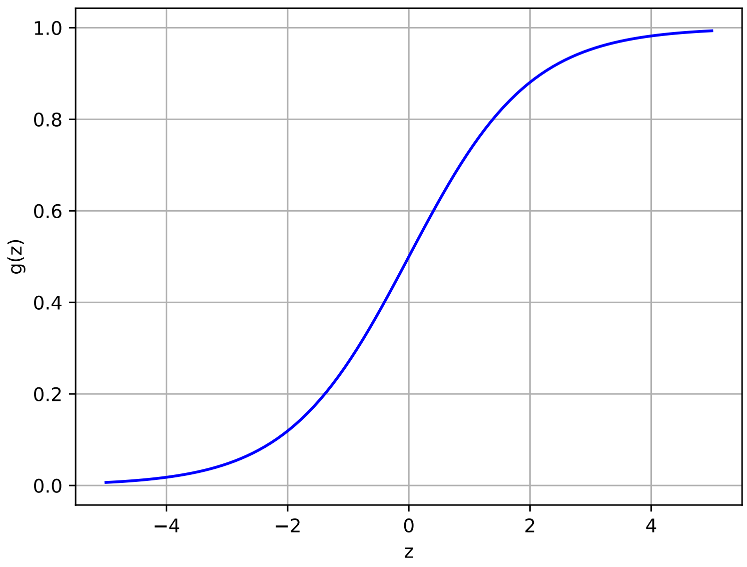
A pair (x, y) is called a training example. We usually have a lot of these examples in a form of training data-set $\{x^{(i)}, y^{(i)} \mid i \in 0, \ldots, m\}$. Our goal is to train a function $h(x^{(i)})$, that predicts the corresponding value $y^{(i)}$ as good as possible. The trained version of this function (often called a hypothesis) is the result of our endeavor. If the space of targets $y^{(i)}$ is continuous, we denote the learning problem as regression (ex. bounding box coordinates) [17]. If it takes just a fixed number of discrete values (ex. artefact cleaning), we call it a classification problem. Let us now look at a simple learning algorithm for classification, where we will see all the required components in action. Note that we could try to solve the problem of artefact cleaning with this very approach.

3.2.2 Logistic regression

Even though this method has the word *regression* in its name, it is actually used for binary classification with two classes $0, 1$, as it outputs values $h(x) \in (0, 1)$. The first step is to take the dot product between a set of trainable parameters θ and feature vector x. Usually, additional feature x_0 is added, so the parameter θ_0 can function as a bias term. With this modification the resulting dot product is calculated as (where n is the number of features): $\theta^T x) = \theta_0 x_0 + \sum_{j=1}^{n} \theta_j x_j = \theta_0 + \sum_{j=1}^{n} \theta_j x_j$ [17].

To arrive at our hypothesis $h(x)$, we additionally apply a nonlinear logistic function $g(z)$ over this dot product. This function is also known as the sigmoid function and can be used as an activation function in neural networks, as we will see later [19]. The definition of $g(z)$ and its graph on interval $z \in [-5, 5]$:

$$g(z) = \frac{1}{1 + e^{-z}}$$



Now we define our hypothesis $h_\theta(x)$ as (with θ in the index we denote that the result of this function depends on the parameters θ):

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$

To derive the gradient descent rule used in training, we will also need the derivative of the sigmoid function:

$$\begin{aligned}
g'(z) &= \frac{d}{dz}\frac{1}{1+e^{-z}} \\
&= \frac{-1}{(1+e^{-z})^2} \cdot (-(e^{-z})) \\
&= \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}} \\
&= \frac{1}{1+e^{-z}} \cdot \frac{1+e^{-z}-1}{1+e^{-z}} \\
&= \frac{1}{1+e^{-z}} \cdot \left(\frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}}\right) \\
&= \frac{1}{1+e^{-z}} \cdot \left(1 - \frac{1}{1+e^{-z}}\right) \\
&= g(z)(1-g(z))
\end{aligned}$$

3.2.3 Training

We interpret the output $h_\theta(x)$ as the probability of example x belonging to class 1, therefore the probability of the 0 class is $1 - h_\theta(x)$. Formally, we can write this using the conditional probability notation:

$$P(y = 1 \mid x; \theta) = h_\theta(x), \quad P(y = 0 \mid x; \theta) = 1 - h_\theta(x)$$

And more generally for any of the two values of y we can join this into a single expression:

$$P(y \mid x; \theta) = (h_\theta(x))^y (1 - h_\theta(x))^{1-y}$$

Our goal is to optimize the parameters θ in such a way, that this probability is maximized (i.e. equals 1 for every example). This is also called maximizing the likelihood of the parameters. If we have m training examples, let $L(\theta)$ denote the probability of all of them:

$$L(\theta) = \prod_{i=1}^{m} P(y^{(i)} \mid x^{(i)}; \theta) = \prod_{i=1}^{m} (h_\theta(x^{(i)}))^{y^{(i)}} (1 - h_\theta(x^{(i)}))^{1-y^{(i)}}$$

To get rid of the multiplication, we usually transform this into log likelihood $l(\theta)$. Note that now the maximum value is 0, since $log(1) = 0$:

$$l(\theta) = \log L(\theta) = \sum_{i=1}^{m} y^{(i)} \log h(x^{(i)}) + (1 - y^{(i)}) \log(1 - h(x^{(i)}))$$

Generally, we optimize our machine learning algorithms with regards to a *loss function* [3], which we wish to minimize [12]. To be concise with this habit, we can easily transform this maximization problem into minimization one, by taking the negative of max log likelihood. With this step, we arrive at the widely used *Binary Cross-Entropy loss* (BCE) function [4] (which we also use in the convolutional neural network for artefact cleaning). There are several ways how to obtain this common loss function (beginning at the term of *entropy* from information theory), but perceiving it as the negative of maximum log likelihood is quite intuitive. Now the minimal value of this loss is 0 and in the worst scenario it can possibly go to infinity. It is common to normalize the loss value with the number of training examples, so it is consistent across datasets of different sizes.

$$\begin{aligned}
\textit{Binary Cross-Entropy loss} &= BCE(\theta) \\
&= -\frac{1}{m} l(\theta) = \\
&= -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log h(x^{(i)}) + (1 - y^{(i)}) \log(1 - h(x^{(i)}))
\end{aligned}$$

One common way to minimize this loss is simply attractively moving in the opposite way of its gradient $\nabla_\theta BCE(\theta)$. We do this by finding the partial derivative according to individual parameters θ_j. We skip the derivation and write directly the result, as it is widely known and can be looked up for example in [17]. :

$$\frac{\partial}{\partial \theta_j} BCE(\theta) = -\frac{1}{m} \sum_{i=1}^{m} (y^{(i)} - h_\theta(x^{(i)})) \cdot x_j^{(i)}$$

Note: Sigmoid function is at the core of logistic regression and has a nice derivative, which allowed us to obtain this elegant update rule. As mentioned, it can also be used as an activation function in more complicated methods such as neural networks. However, this is not recommended nowadays, as it has been shown that it has several drawbacks in the context of neural nets (saturation kills gradients, non zero-centered, expensive to compute etc. [3]).

[3]https://keras.io/api/losses/

[4]https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a

Finally we can write the rule of our **gradient descent** rule:

$$\begin{aligned}\theta_j &= \theta_j - \alpha \cdot \frac{\partial}{\partial \theta_j} BCE(\theta) \\ &= \theta_j + \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} (y^{(i)} - h_\theta(x^{(i)})) \cdot x_j^{(i)} \qquad \text{(for every } j\text{)}\end{aligned}$$

Here α is an adjustable *hyperparameter* called *learning-rate*. This can be fixed, but often times it dynamically changes during the training, to increase the success of convergence [22]. We repeat this rule for every parameter θ_j. This method always look at every example in the training set on every step. To make training faster and more memory efficient (we usually cannot fit the whole training set of training images into a GPU memory), this rule is often modified into **minibatch gradient descent** or even **stochastic gradient descent** [17].

In **minibatch gradient descent** we divide the m training examples into b batches. Every batch will by identified by a *start* and *end* index of training example in the whole set. We can then update the parameters with respect only to the gradient over the examples of a single batch:

```
Repeat until convergence:
    for k=1 to b:
        $\theta_j = \theta_j + \alpha \cdot \frac{1}{m/b} \cdot \sum_{i=batch_{start}^{(k)}}^{batch_{end}^{(k)}} (y^{(i)} - h_\theta(x^{(i)})) \cdot x_j^{(i)}$        (for every $j$)
```

In an extreme case when every batch is of size 1, we get **stochastic gradient descent**. Here we update the parameters with regards to a loss on just a single training examples. Our training examples are not ordered in any particular order and therefore they basically come in a random order, stochastically affecting the values of parameters θ. This is much faster than the full **gradient descent**. However, it may never converge to the global minimum, even in the case of a convex loss function. More often than not, it will oscillate around this minimum.

```
Repeat until convergence:
    for i=1 to m:
        $\theta_j = \theta_j + \alpha \cdot (y^{(i)} - h_\theta(x^{(i)})) \cdot x_j^{(i)}$        (for every $j$)
```

The act of going over a whole training set during training, using any of the approaches above, is usually called a single *training epoch*. *Epochs* can consist of variable amount of *steps*, depending on the size of the dataset and chosen training scheme.

3.3 Convolutional Neural Networks

In general, convolution is an operation defined on two real-valued functions $f(x)$ and $w(x)$, where x can for example denote time or position in space. In convolutional neural networks, $f(x)$ is the input into convolution operation and $w(x)$ (often called a kernel) defines some sort of a average weighting over certain parts of $f(x)$. In this work, we will assume the definition of discrete convolution, taking an discrete input (ex. image) and a discrete kernel, which convolves over the input producing activation maps [7].

Compared to traditional neural networks with fully connected layers, convolutional layers which are the central building blocks of this work provide several benefits. First off, we are mostly working with outputs from a structured light scanner, which comes in a form of an organized point cloud. This can be viewed as an range image with additional information for each point apart from its location. We can perceived the individual features as the channels of the image, which together define the depth of our input volume.

For example, let us assume that the input scan is of size 1000×1000. The x and y coordinates of each point (in the camera space) is expressed implicitly by the location of point in the grid. For the z coordinate however, we need an individual channel of values, forming a depth map. Additionally, we may take the normals of each point represented by 3 values and a map of intensities, forming yet another channel.

In this case, our input volume has size $1000 \times 1000 \times 5$. If we were to build a fully connected layer of this input, each neuron would have $1000 * 1000 * 5 = 5000000$ parameters. And that is just for a single neuron! High numbers of parameters would quickly lead to over-fitting and would also contribute to increased inference time, training time and memory consumption [3]. Moreover, lot of parameters does not always lead to a better performance [9]. Our goal in the future is to run the ML models derived here directly on the structured light scanner, which often contains some light-weight processor, such as Nvidia Tegra, which has similar performance as mobile devices [24] [6].

3.3.1 Convolutional Layer

Intuitively, in our processing tasks such as artefact cleaning, we never really need a neuron to see the whole spatial extent of an incoming volume. This is where the notion of *local connectivity* comes to play. Here we connect the neuron only to a local region from the input.

The spatial size of this connectivity is often called *receptive field* of the given neuron, practically this is the size of the filter. It is important to note that the filter has always the same depth as the input volume, so connections are local in width and height, but always full along the depth axis. For example, if we had the same input volume of size $1000 \times 1000 \times 5$ and filter of size 7×7, single neuron would have $7 * 7 * 5 = 245$ weights plus a single bias, resulting in 246 parameters. In a scenario of a simple convolutional layer without additional extensions (zero-padding, stride ...) [5], one side of the output volume will in this case have size $1000 - 7 + 1 = 994$, so a single activation map would be of size 994×994 where every pixel is an output of a single neuron [3].

Apart from local connections, another technique to reduce the number of parameters and promote in-variance of features across different spatial locations is the scheme of *parameter sharing*. In the example above, we have $994 * 994 = 988036$ neurons, each with 246 parameters. However, in convolutional layers we make a reasonable assumption, that if some local feature is useful at some location (x_1, y_1), it is probably useful to compute the same feature at a different location (x_2, y_2) with the same filter. Therefore we are going to constrain all the 994×994 neurons (sometimes also called a single *depth slice*) producing a single activation map to use the same set of learned weights. Suddenly, the whole convolutional layer has only 246 parameters.

Receptive field is one hyper parameter, convolutional layers have also a *depth parameter*, which denotes the number of filters we would like to train [25]. So for example, if we were to apply a convolutional layer with *filter size* $= 7 \times 7$ and *depth* $= 32$ over our $1000 \times 1000 \times 5$ input volume, the resulting volume would have spatial size $994 \times 994 \times 32$, where each of the 32 activation maps is produced by single $7 \times 7 \times 5$ filter. This layer has $246 * 32 = 7872$ learnable parameters [3].

After the description of fundamental characteristic of a convolutional layers, we arrive back at the notion of convolution from the beginning of this chapter. Since all the neurons in the same depth slice are using identical set of weights, we compute the forward pass of a single depth slice as a convolution of the filter with the input volume. Mathematically, this is like taking an element-wise dot product between the sub-region of input volume and the filter. In even simpler maths, we can implement this as a dot product between the weights of filter unrolled into a single column vector and a matrix, where each row is formed by unrolled values of a sub-region from the input volume. For example, let us consider a $3 \times 3 \times 1$ input $\mathbf{X}$ and $2 \times 2 \times 1$ filter $\mathbf{K}$, the convolution

[5] https://towardsdatascience.com/types-of-convolutions-in-deep-learning-717013397f4d

operation is often denoted with $*$:

$$\mathbf{X}*\mathbf{K} = \begin{pmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{pmatrix} * \begin{pmatrix} k_1 & k_2 \\ k_3 & k_4 \end{pmatrix} = \begin{pmatrix} x_1 & x_2 & x_4 & x_5 \\ x_2 & x_3 & x_5 & x_6 \\ x_4 & x_5 & x_7 & x_8 \\ x_5 & x_6 & x_8 & x_9 \end{pmatrix} * \begin{pmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \end{pmatrix}$$

These types of layers are also called 2D convolutions. This can be confusing at first, since the input is in fact a 3D volume, but the output is a single value and the filter slides only in two dimensions, always having the same depth as input, as mentioned above. In Figure 3.3, we present an illustration of a convolutional layer with *depth* $=$ 1 and *receptive field* $=$ 3, over a 2-channel (depth, intensity) example input image. Note that in reality, each pixel would be represented by a single neuron, here we scale them up for the sake of clarity.

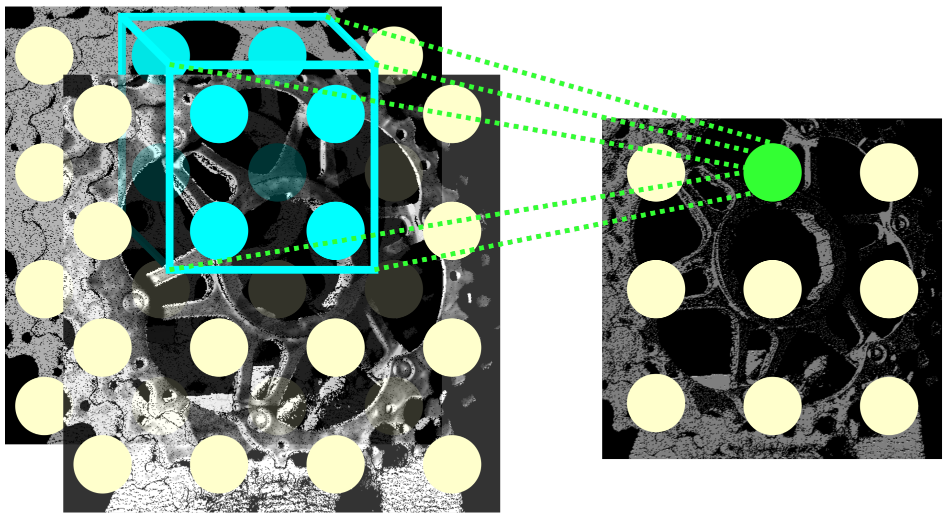


Figure 3.3: Example of a convolution layer operation over an input volume of size $4\times4\times2$, where *depth* $=1$ and *filter size* $=2\times2$. Notice, how the filter extents the full depth of input and produces a single value. After sliding the filter over whole image, we got a single activation map.

3.3.2 Zero-padding

As we saw above, the convolution operation with a filter of size $F \times F$ scales down the both the input's width and height by $F-1$, in case of an odd F (we rarely use a filter of even dimensions). To prevent this, we can pad the input with $\lfloor F/2 \rfloor$ zeros from all four sides. This is yet another *hyperparameter* of the convolutional layer, used to control the spatial size of the output.

3.3.3 Nonlinearity

The activation of each neuron after taking the dot product is often run thru a non-linear function. This increases the *hypothesis space* of our model, giving it greater representational capacity. Without these non-linearities, we would be just stacking linear transformations (dot products), which is often not enough. These functions are also called activation functions. In the past, one of the widely used activation was the sigmoid function $f(x) = \frac{1}{1+e^{-x}}$, which takes the output of the neuron x and squashes into the $[0, 1]$ interval. However, the training time with gradient descent is consistently faster when using a simple Rectified Linear Unit (ReLU) activation function $f(x) = max(0, x)$ [13]. This function is therefore widely used in practice, but it is always best to experiment with different activation functions for every problem, therefore treating it as an adjustable *hyperparameter*.

3.3.4 Pooling Layer

It is common to insert pooling layers after convolutional layers, which reduce the spatial size of the volume. It is used to reduce the number of parameters, control overfitting and filtering noisy activations, by taking only a single value from a *receptive field*. Other beneficial effects are contributed to this operation, for example granting the network an invariance to small translations and granting the model a more global, aggregated features [15]. This operation works independently on every depth slice (activation map), commonly with a *receptive field* of size $F = 2 \times 2$ applied to every other pixel (denoted as a *stride hyperparameter* with a value of 2). This effectively halves the width and height of the volume. It is important to note that since the pooling works independently on the individual depth slices, it does not reduce the depth of the volume (for that exists another technique described below [23]). We can downscale the regions of size F with some common aggregating function, for example taking the average of those values, resulting in an average-pooling operation. However, the most common operation is to take the maximal value, i. e. doing max-pooling [3]. It is widely adopted that by taking the maximal value, we are left only with the most representative value from the subregion, as it is the value of the pixel with a highest neural response [18].

But there are also downsides to pooling, especially in the context of tasks that require precise localization, such as segmentation. In the process of pooling, we lost the spatial information within the *receptive field* and by downsampling the volume, we practically cannot output a segmentation mask with the same size as the input.

Therefore we often try to do a reverse operation in a later layers of the network. The most simple operation is unpooling, which is often connected with usage of the switch variables. In this approach we remember the location of chosen activations during the pooling operation and then we place them back into their original locations during upscaling, while filling the other locations with zero, or some other value, see Figure 3.4. There are also more complicated techniques, which tries to mimic the inverse function of pooling, we will describe those in the next chapter along the architectures where they are used [18]. Do note that pooling is not the only way to change size of the activation maps in the network. We can always use some common techniques from image processing, or just do a convolution with a *stride* > 1, which basically skips some pixels while sliding the filter over input volume, resulting in a similar decrease of the spatial size.

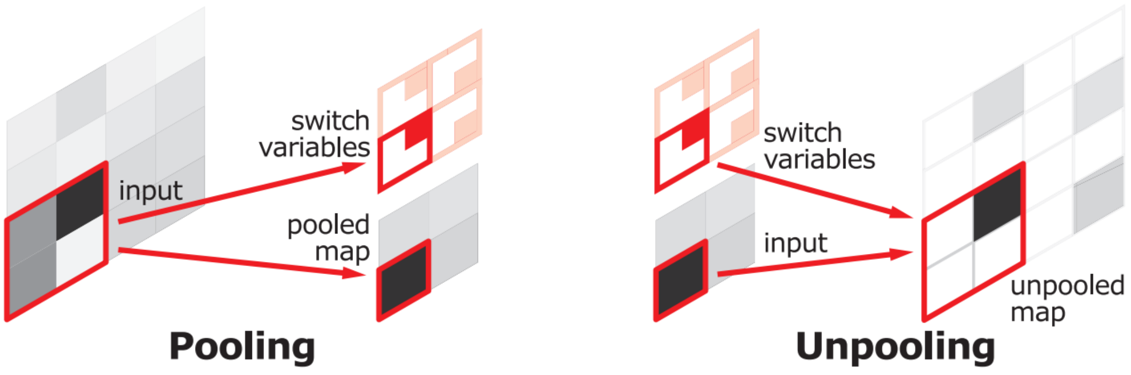


Figure 3.4: Illustration of pooling and unpooling operations taken from [18].

3.3.5 Fully convolutional network (FCN)

Neurons at the first levels of a convolutional network contain information for small region of images, since their *receptive field* is just few pixels. As we add more layers, image size keeps decreasing and the number of channels usually increase and even though the *filter sizes* are still kept small, higher levels inherently *see* much larger portion of the original image. It is generally understood that the initial layers learn simple, low level concepts, such as and edges and colors and the later levels exploit this features to learn higher level concepts, such as different objects [25].

Image classification task (assigning single class to an input) made the neural networks popular. Here we are interested in obtaining a fixed length vector at the end, denoting probabilities of discrete classes defined before training. To do this, we flatten the spatial tensor from a last convolutional layer and map it to our result vector using a fully connected layer. This process destroy all the spatial information [6].

For all the tasks in this work (filtration, segmentation, detection), we need to retain the spatial information, so no fully connected layers are ever used. Apart from keeping spatial information, this approach has another benefit.

When we use a fully connected layer, there need to be a fixed amount of neurons, each being connected to every neuron from the previous layer by a separate trainable parameter. Therefore the output volume of the last convolutional layer must have a fixed dimensions as well. This effectively forces us to determine a spatial size of the input feature images when modeling the network. On the other hand, fully convolutional networks naturally operates on an input of arbitrary size [15]. The number of parameters of a learnable filter is fixed, but we can slide it over volume of any size. These types of networks are also translation invariant by their nature, since they operate on local regions, they are able to detect a certain object at any place, or filter an artefact anywhere in the input.

The only limitation of a fully convolutional network is a minimal size of the input. Since we are downsampling in the process (usually by factor of 2) and applying convolutions afterwards, the volume can never reach a spatial size that is smaller than dimensions of the filter to be applied. In fact, fully connected layers can also be viewed as convolutions with kernels that cover the entire input volume. This means we can convert any network with a fully connected layers to a convolutional network and the

[6]https://divamgupta.com/image-segmentation/2019/06/06/deep-learning-semantic-segmentation-keras

fixed input size of the original network now becomes the minimal size of the resulting convolutional network. Moreover for inputs of greater size, this network produces a spatial output map, sometimes called a *heat map*. The last convolutional layer with a big *receptive field* does the final assignment in a tiled fashion [15].

Let us examine this on a concrete example 3.5. Suppose we have a network for classification into 3 classes with a very basic architecture. The last $17 \times 17 \times 4$ volume is flattened into one-dimensional vector, effectively creating a fully connected layer with 1156 units, that is connected to a last output layer with 3 neurons, each connected to every previous neuron by 1156 weights.

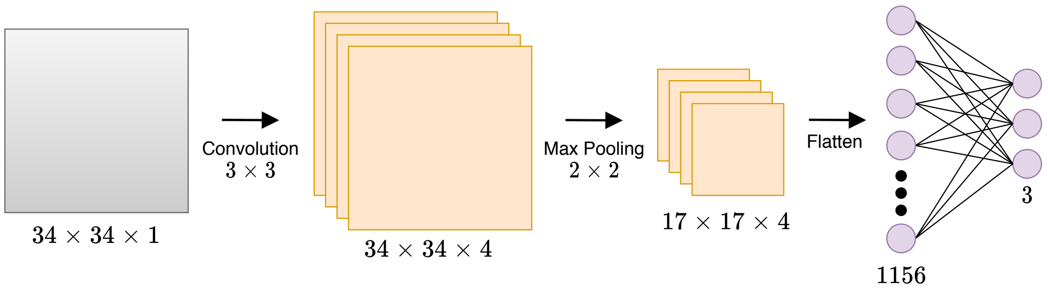


Figure 3.5: Example of a very basic network used to classify the input image of fixed size into 3 classes.

We can very easily transform this into a fully convolutional layer with the following layers, written here with all *hyperparameters*:

$$\begin{gathered} conv\_layer(kernel = 3 \times 3,\ filters = 4,\ zero\_padding = true) \\ \downarrow \\ max\_pool(kernel = 2 \times 2, stride = 2) \\ \downarrow \\ conv\_layer(kernel = 17 \times 17,\ filters = 3,\ zero\_padding = false) \end{gathered}$$

Applying this network on an image of original fixed size $34 \times 34 \times 1$ produces 3 activation maps at the end (see Figure 3.6), each consisting of just a one pixel, or more generally a $1 \times 1 \times k$ volume, with $k = 3$ in this example. We can interpret the pixel value of each activation map as a probability of belonging to the given task.

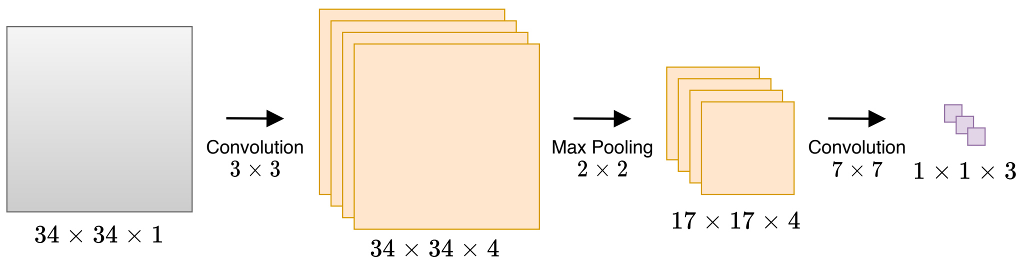


Figure 3.6: Original network from 3.5 reworked into a FCN. The last filter contain $17 * 17 * 4 = 1156$ weights, just as the fully connected neurons in the original net.

Now we are able to run inference on image of arbitrary bigger spatial size with this network. For example, by giving it an image of size $256 \times 256 \times 1$, we obtain a $112 \times 112 \times 3$ volume, these are basically three *heat maps*, each for a given classification class from the original problem. This is the same effect as if we were to tile the $256 \times 256 \times 1$ pixel by pixel into windows of size $34 \times 34 \times 1$. This would result in $112 * 112 = 12544$ such windows and we could ran the original network from 3.5 on every single one of them, obtaining the same result.

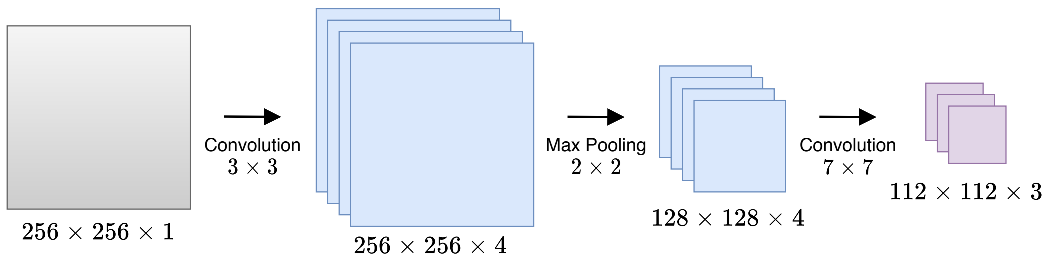


Figure 3.7: Caption

Not only this is more flexible, but as described in [15], performance of this approach is higher, due to a high amortization over the overlapping regions. Figure 3.8 provides an illustration of this transformation on a particular image. These are numerous techniques how to convert these coarse output heat maps and use them for pixel-wise segmentation of the original image, or use them for precise object localization, as we will describe in later chapters.

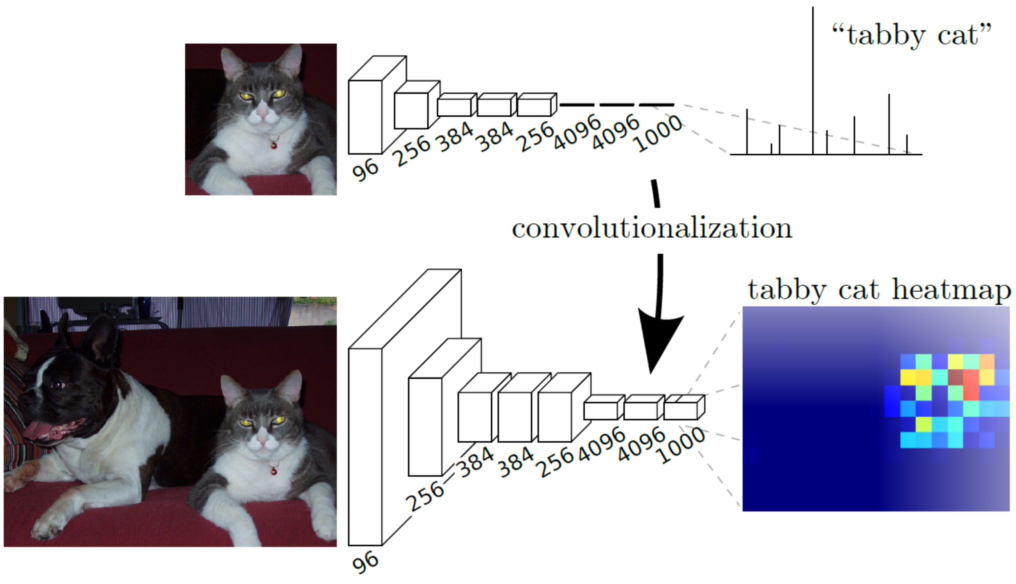


Figure 3.8: By transforming fully connected layers to convolution layers, we can use classification network to output spatial heat maps [15].

3.3.6 1×1 convolutional layers

Even convolutions with a filters of spatial size 1×1 makes a perfect sense. Since the kernel always goes through a whole depth of the input volume, these types of convolutions effectively act as taking a dot product over the single spatial location over all depth slices. We can use this in a role of other fully connected layers in the examples above, as we can apply these on the $1 \times 1 \times n$ volumes [3].

1×1 convolutions are also used just as a simple means of dimension reduction. After doing a standard convolution, it is possible that some of the activation maps share common information and we can aggregate these into fewer depth slices. This works as a form of a compression, where we apply these convolutions to reduce the number of channels before applying a more expensive convolutional layers with larger filters [23].

Chapter 4

Related Work

4.1 CNN Origins

4.1.1 LeNet

4.1.2 AlexNet

4.1.3 ResNet

4.2 Semantic Segmentation

4.2.1 FCNN

4.2.2 Deconvolution

4.2.3 U-Net

4.2.4 LU-Net

4.3 Object Localization

4.3.1 You Only Look Once

4.3.2 RetinaNet

Chapter 5

Software Specification

Chapter 6

Pipeline Proposal

Chapter 7

Research

7.1 Data-set

7.2 Architecture

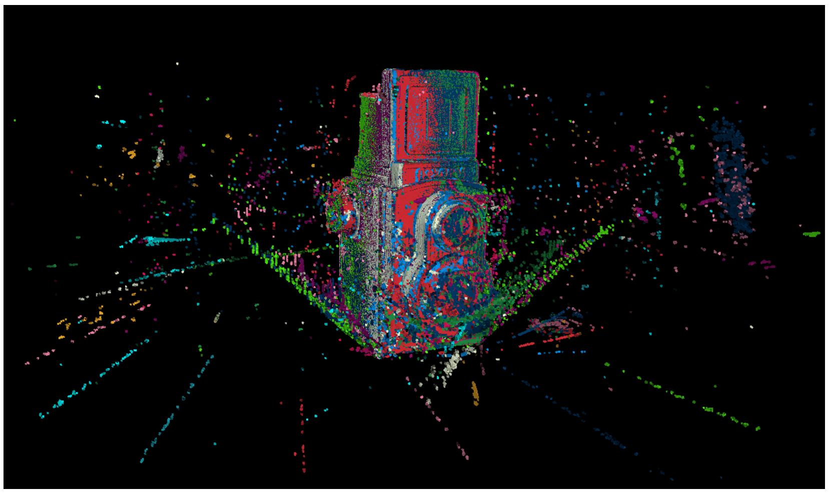
7.3 Results

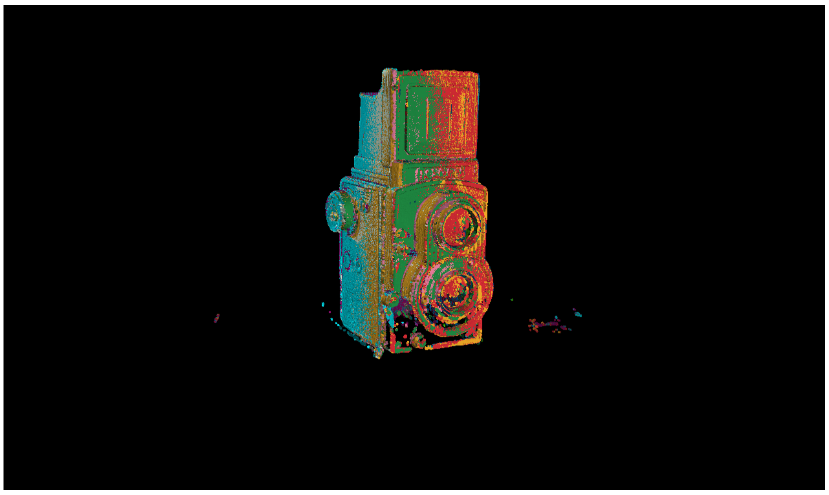
7.3.1 Artefact Filtration

Here we evaluate our filtration CNN on set of data-sets, unseen during the training.

Dataset	# scans	Analytic (s)	FCNN (s)	(min, max, avg) **IoU**
Flexaret	19	4.32	(4.35, 45.46)	(0.4313, 0.942, 0.768)

Table 7.1: Quantitative comparasion between the results of analytic filtration (ground-truth) and our CNN solution. We are using IoU metric for evaluation.


(a) unfiltered


(b) ground truth

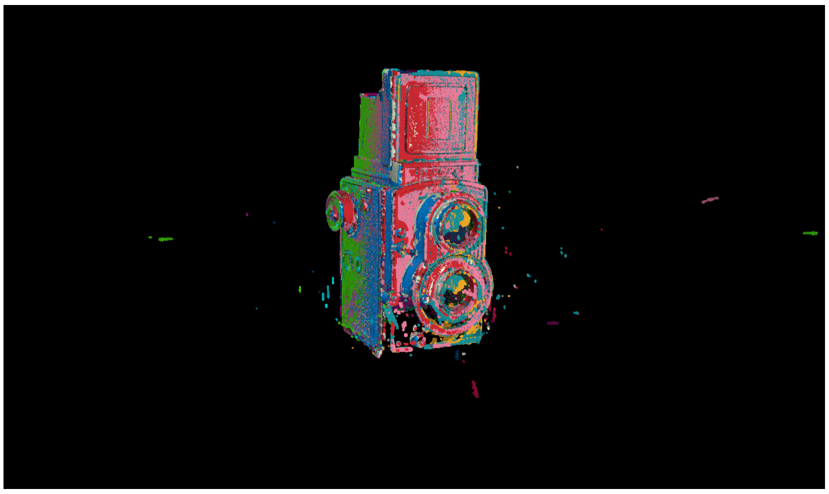
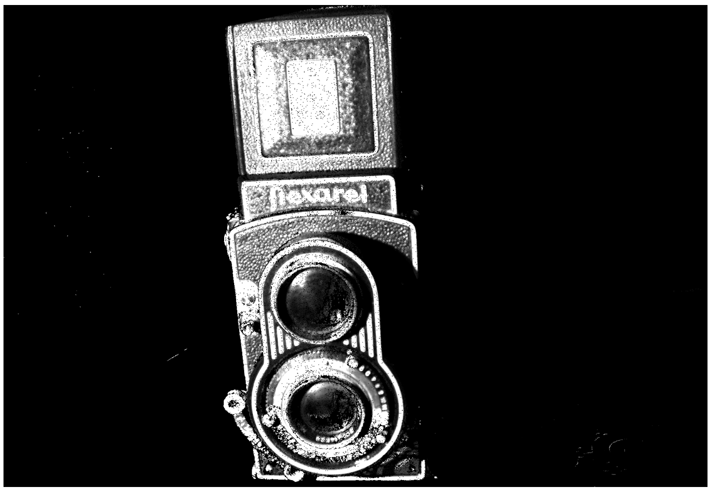
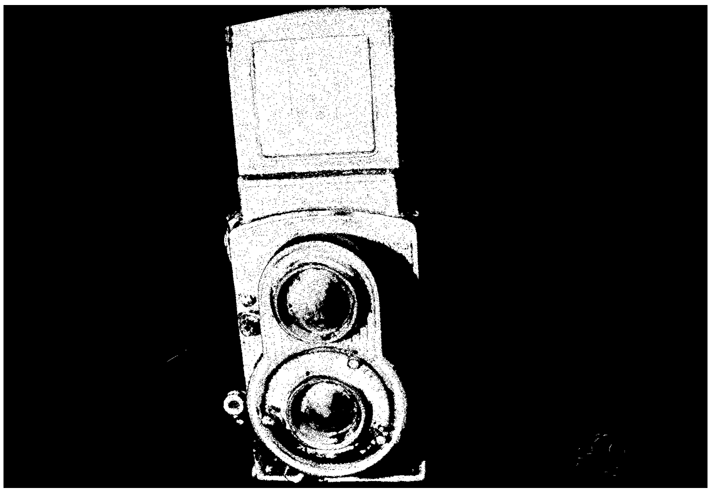

(c) prediction

Figure 7.1: Qualitative comparison of point clouds (each scan has different color)


(a) raw (intensity map)


(b) ground truth

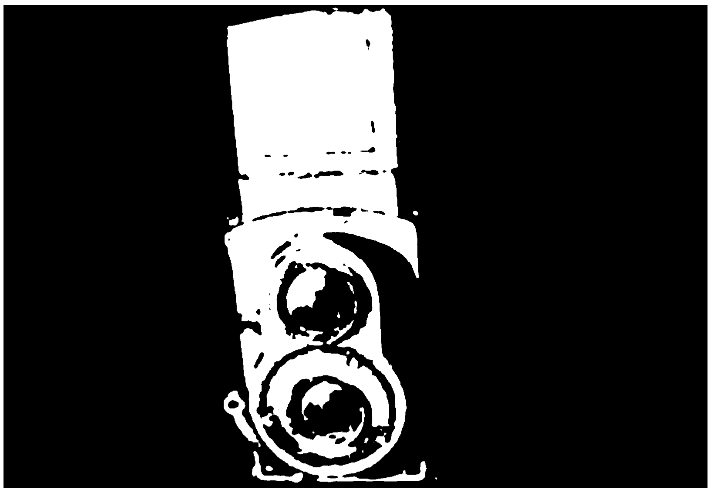
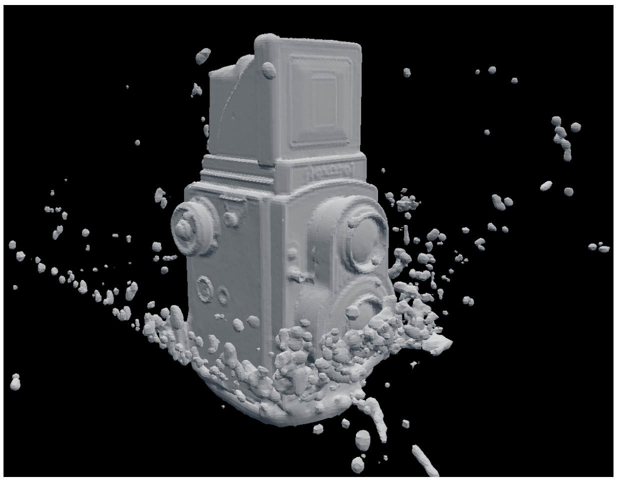
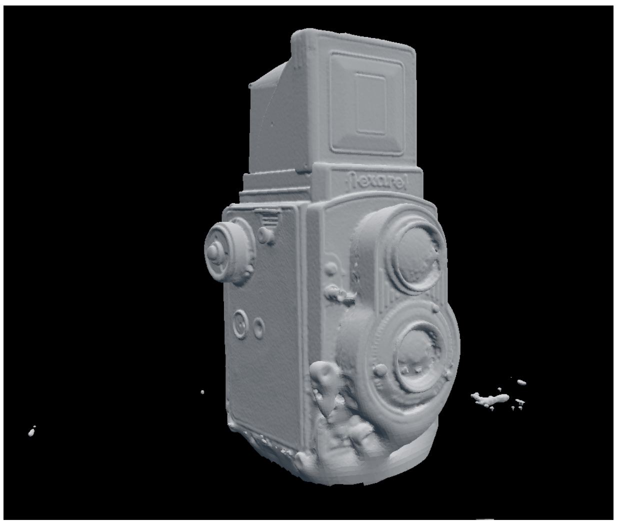

(c) prediction

Figure 7.2: Qualitative comparison of filtration masks.


(a) mesh reconstruction without filtration


(b) mesh from ground-truth analytical filtration

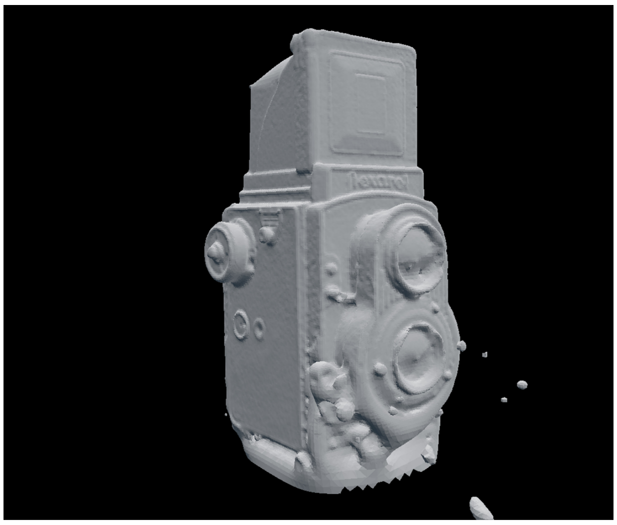

(c) mesh from proposed FCNN filtration

Figure 7.3: Qualitative comparison of resulting meshes
(obtained using *Poisson surface reconstruction).*

7.3.2 Object Localization

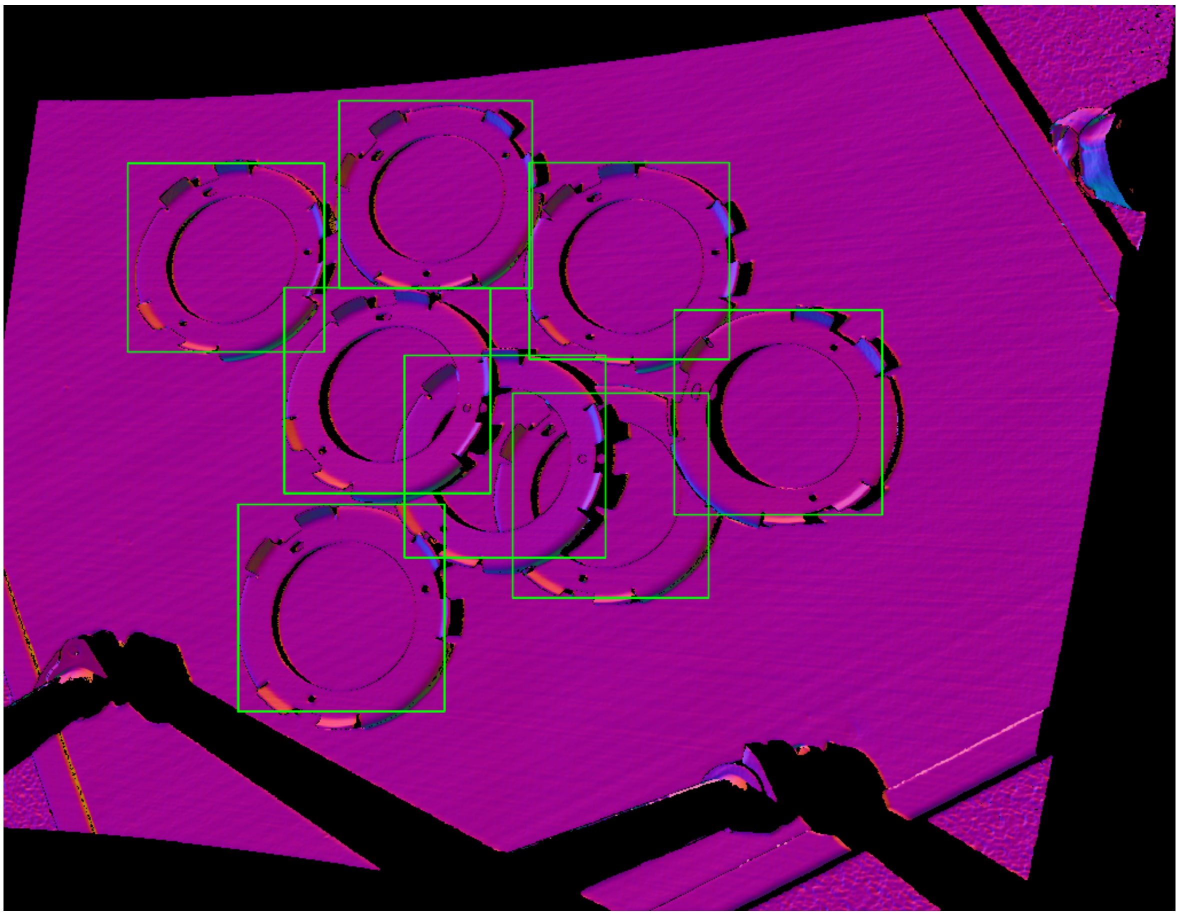


Figure 7.4: Bounding box detection on a real scan.

Chapter 8

Implementation Details

Chapter 9

Conclusion

Bibliography

[1] Atzmon, M., Maron, H., and Lipman, Y. Point convolutional neural networks by extension operators, 2018.

[2] Biasutti, P., Lepetit, V., Aujol, J.-F., Brédif, M., and Bugeau, A. Lu-net: An efficient network for 3d lidar point cloud semantic segmentation based on end-to-end-learned 3d features and u-net, 2019.

[3] Byun, A. CS231n lecture notes - convolutional neural networks for visual recognition. `https://cs231n.github.io/`, 2017.

[4] Charles, R., Su, H., Kaichun, M., and Guibas, L. PointNet: deep learning on point sets for 3D classification and segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (07. 2017), pp. 77–85.

[5] Chollet, F. *Deep Learning with Python.* Manning, Nov. 2017.

[6] Coleman, C. A., Narayanan, D., Kang, D., Zhao, T., Zhang, J., Nardi, L., Bailis, P., Olukotun, K., Ré, C., and Zaharia, M. Dawnbench : An end-to-end deep learning benchmark and competition.

[7] Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[8] Gron, A. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, 1st ed. O'Reilly Media, Inc., 2017.

[9] Hasanpour, S. H., Rouhani, M., Fayyaz, M., and Sabokrou, M. Lets keep it simple, using simple architectures to outperform deeper and more complex architectures, 2018.

[10] Havaei, M., Davy, A., Warde-Farley, D., Biard, A., Courville, A., Bengio, Y., Pal, C., Jodoin, P.-M., and Larochelle, H. Brain tumor segmentation with deep neural networks. *Medical Image Analysis 35* (Jan 2017), 18–31.

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

[12] Jadon, S. A survey of loss functions for semantic segmentation, 2020.

[13] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems* (2012), F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25, Curran Associates, Inc., pp. 1097–1105.

[14] Lin, T.-Y., Goyal, P., Girshick, R., He, K., and Dollár, P. Focal loss for dense object detection, 2018.

[15] Long, J., Shelhamer, E., and Darrell, T. Fully convolutional networks for semantic segmentation, 2014.

[16] Mitchell, T. M. *Machine Learning.* McGraw-Hill, New York, 1997.

[17] Ng, A. CS229 lecture notes - supervised learning. `https://see.stanford.edu/Course/CS229`, 2012.

[18] Noh, H., Hong, S., and Han, B. Learning deconvolution network for semantic segmentation, 2015.

[19] Nwankpa, C., Ijomah, W., Gachagan, A., and Marshall, S. Activation functions: Comparison of trends in practice and research for deep learning. *CoRR abs/1811.03378* (2018).

[20] Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. You only look once: Unified, real-time object detection, 2016.

[21] Ronneberger, O., Fischer, P., and Brox, T. U-Net: convolutional networks for biomedical image segmentation. vol. 9351, pp. 234–241.

[22] Ruder, S. An overview of gradient descent optimization algorithms, 2017.

[23] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions, 2014.

[24] Zebin, T., Scully, P. J., Peek, N., Casson, A. J., and Ozanyan, K. B. Design and implementation of a convolutional neural network on an edge computing smartphone for human activity recognition. *IEEE Access 7* (2019), 133509–133520.

[25] Zeiler, M. D., and Fergus, R. Visualizing and understanding convolutional networks, 2013.

[26] Zitnick, C. L., and Dollár, P. Edge boxes: Locating object proposals from edges. In *Computer Vision – ECCV 2014* (Cham, 2014), D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds., Springer International Publishing, pp. 391–405.