Deep Learning-based Computer Vision Pipeline to improve Situational Awareness of an Autonomous Vehicle

Semesterarbeit
an der Fakultät für Maschinenwesen der Technischen Universität München.

Lehrstuhlinhaber	Univ.-Prof. Dr.-Ing. Markus Lienkamp Lehrstuhl für Fahrzeugtechnik
Betreut von	Johannes Betz, M.Sc. Lehrstuhl für Fahrzeugtechnik
Eingereicht von	Pablo Rodríguez Palafox, B.Sc. Matrikelnummer: 03694210
Ausgabe am	01.12.17
Eingereicht am	01.06.18

Lehrstuhl für Fahrzeugtechnik
Fakultät für Maschinenwesen
Technische Universität München

Aufgabenstellung

Deep Learning-based Computer Vision Pipeline to improve Situational Awareness of an Autonomous Vehicle

As part of the Roborace project, the Lehrstuhl für Fahrzeugtechnik is developing the software for a vehicle that will participate in the first racing series for autonomous vehicles. The present thesis is intended to improve the situational awareness of the vehicle by using its frontal RGB cameras.

The goal is to develop a deep learning-based pipeline capable of computing the width of the road at a certain depth in an input image. This will be accomplished by fusing together a 2D semantic segmentation network and a monocular depth estimation network. Both deep learning models will be trained on images from Roborace. To this end, a semantic segmentation dataset featuring Roborace images will have to be manually labelled, and the availble stereo pairs from the competition will also be put together into a stereo dataset on which a depth estimation deep learning architecture can be trained. The resulting pipeline, which will make use of the trained models, will be applied on real images featuring streets of Munich in order to compare the predicted and real values of the road's width at a certain depth. Inference time of the pipeline will be computed in order to check for real-time capability.

The following points are to be addressed:

- Familiarization with the state of the art: Deep Learning in Computer Vision, particularly semantic segmentation and depth estimation
- Development of a pipeline to compute the road's width at a certain depth in an input image
- Creation of both a semantic segmentation dataset and a stereo dataset from the available Roborace images
- Application of the pipeline onto real recorded videos or set of images
- Evaluation of the pipeline, both in accuracy and efficiency

The preparation should document the individual work steps in a clear form. The candidate undertakes to carry out the work independently and to indicate the scientific aids he uses.

The submitted work remains as a document in the ownership of the chair and may be made accessible to third parties only with the consent of the chair holder.

Ausgabe: 01.12.2017 Abgabe: 01.06.2018

Prof. Dr.-Ing. M. Lienkamp

Betreuer: Johannes Betz, M. Sc.

Lehrstuhl für Fahrzeugtechnik
Fakultät für Maschinenwesen
Technische Universität München

Geheimhaltungsverpflichtung

Herr **Rodríguez Palafox, Pablo**

Im Rahmen der Angebotserstellung und der Bearbeitung von Forschungs- und Entwicklungsverträgen erhält der Lehrstuhl für Fahrzeugtechnik der Technischen Universität München regelmäßig Zugang zu vertraulichen oder geheimen Unterlagen oder Sachverhalten industrieller Kunden, wie z. B. Technologien, heutige oder zukünftige Produkte, insbesondere Prototypen, Methoden und Verfahren, technische Spezifikationen oder auch organisatorische Sachverhalte.

Der Unterzeichner verpflichtet sich, alle derartigen Informationen und Unterlagen, die ihm während seiner Tätigkeit am Lehrstuhl für Fahrzeugtechnik zugänglich werden, strikt vertraulich zu behandeln.

Er verpflichtet sich insbesondere

- derartige Informationen betriebsintern zum Zwecke der Diskussion nur dann zu verwenden, wenn ein ihm erteilter Auftrag dies erfordert,
- keine derartigen Informationen ohne die vorherige schriftliche Zustimmung des betreffenden Kunden an Dritte weiterzuleiten,
- keine Fotografien, Zeichnungen oder sonstige Darstellungen von Prototypen oder technischen Unterlagen hierzu anzufertigen,
- auf Anforderung des Lehrstuhls für Fahrzeugtechnik oder unaufgefordert spätestens bei seinem Ausscheiden aus dem Lehrstuhl für Fahrzeugtechnik alle Dokumente und Datenträger, die derartige Informationen enthalten, an Lehrstuhl für Fahrzeugtechnik zurückzugeben.

Eine besondere Sorgfalt gilt im Umgang mit digitalen Daten:

- Kein Dateiaustausch über Dropbox, Skydrive o. ä.
- Keine vertraulichen Informationen unverschlüsselt über Email versenden.
- Wenn geschäftliche Emails mit dem Handy synchronisiert werden, darf dieses nicht in die Cloud (z. B. iCloud) synchronisiert werden, da sonst die Emails auf dem Server des Anbieters liegen.
- Die Kommunikation sollte nach Möglichkeit über die (my)TUM-Mailadresse erfolgen. Diese Emails dürfen nicht an Postfächer anderer Emailprovider (z. B.: gmail.com) weitergeleitet werden.

Die Verpflichtung zur Geheimhaltung endet nicht mit dem Ausscheiden aus dem Lehrstuhl für Fahrzeugtechnik, sondern bleibt 5 Jahre nach dem Zeitpunkt des Ausscheidens in vollem Umfang bestehen.

Der Unterzeichner willigt ein, dass die Inhalte seiner Studienarbeit in darauf aufbauenden Studienarbeiten und Dissertationen mit der nötigen Kennzeichnung verwendet werden dürfen.

Datum: 01.12.2017

Unterschrift:_____

Lehrstuhl für Fahrzeugtechnik
Fakultät für Maschinenwesen
Technische Universität München

Erklärung

Ich versichere hiermit, dass ich die von mir eingereichte Abschlussarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Garching, den 01.06.18

_____

Pablo Rodríguez Palafox

Acknowledgment

I would like to express my sincere gratitude to my advisor Johannes Betz for the continuous support and patience during this work, as well as for the chance to collaborate on such an incredible topic. I wish you all the best in the Roborace project.

Garching, den 01.06.18

Pablo Rodríguez Palafox

Contents

Acronyms

ADAS	Advanced Driver Assistance Systems 6
AI	Artificial Intelligence 6
ASPP	Atrous Spatial Pyramid Pooling 80, 81, 84
CNN	Convolutional Neural Network 6, 9, 11–14, 20, 21, 23, 75, 80, 82
CRF	Conditional Random Field 18, 21, 80, 81
CRP	Chained Residual Pooling 82
DL	Deep Learning 3, 6, 8, 9, 12, 16, 21, 25, 27, 31, 65, 67, 85
DNN	Deep Neural Network 2, 3, 6, 8, 9, 22, 34, 65
FCN	Fully Convolutional Network xii, 13–18, 27–29, 34, 41–47, 60, 65, 67, 69, 70, 73, 75, 76
FPS	frames per second 18, 63
fwt	forward pass time 18, 19, 27
GB	Gigabyte 32
GPU	Graphics Processing Unit 6, 18
IoU	Intersection over Union 13, 17–19, 27, 28, 41–46, 48, 49, 59, 65, 66, 70, 75, 76, 78, 79, 81–84
IPM	Inverse Persperctive Mapping 5
ITS	Intelligent Transportation System 18, 67
LDWS	Lane Departure Warning System 6
LIDAR	Light Detection and Ranging 20

ML Machine Learning 6, 8, 31

NN Neural Network 6–9

PSPNet Pyramid Scene Parsing Network 83

RADAR Radio Detection and Ranging 20

ReLU Rectified Linear Unit 7, 10

ResNet Residual Network 13, 18

RGB Red-Green-Blue 20, 34

Formula Directory

Symbol	Unit	Description
net_p		Sum of all the inputs of a neuron in a Neural Network for data point p
out_p		Activation of a neuron in a Neural Network for data point p
E		Loss function in a Neural Network
δ_i		Error signal of neuron i in a Neural Network
$w_j i$		Weight from neuron j to neuron i in a Neural Network
$\hat{y_i}$		Predicted score for class i in a given pixel
y		True class of a given pixel
S		Softmax function
D_p		Cross-entropy Loss of pixel p
L_{image}		Loss function of a given image
L		Loss function of a given batch of images
$R(W)$		Regularization function of weights W
λ		Regularization strength
u	pixels	Horizonatal coordinate in image plane
v	pixels	Vertical coordinate in image plane
c_x	pixels	Horizonatal coordinate of the principal point
c_y	pixels	Vertical coordinate of the principal point
X	pixels	X coordinate in camera space
Y	pixels	Y coordinate in camera space
Z	pixels	Z coordinate in camera space
d		Disparity of a given pixel

Symbol	Unit	Description
$\hat{d}$	m	Depth of a given pixel
f	pixels	Focal length
F	m	Focal length
b	m	Baseline of a stereo rig
Q		Back-projection transformation matrix

1. Introduction

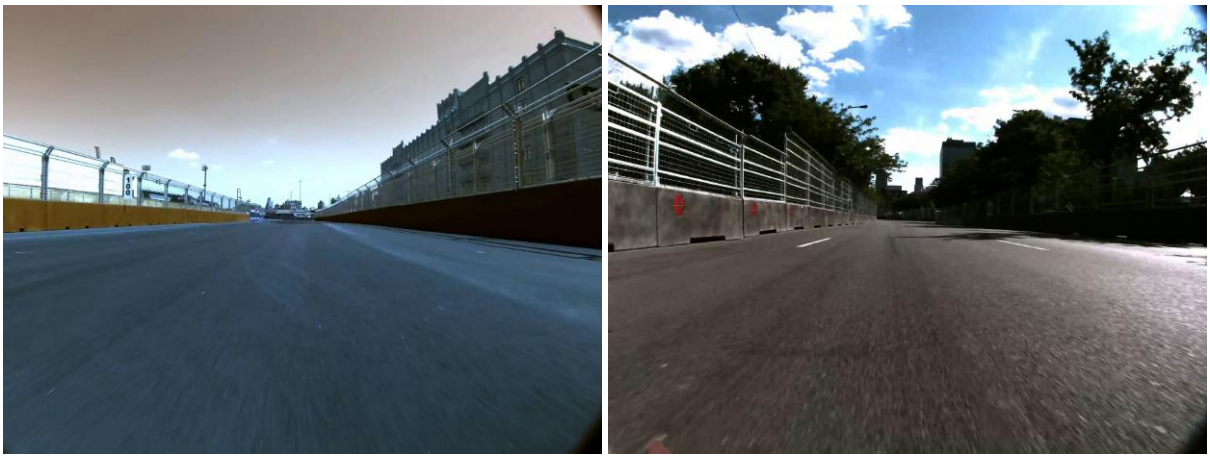
1.1. Motivation

According to the World Health Organization and their Global Status Report on Road Safety 2015 [1], the total number of road traffic deaths has plateaued at 1.25 million per year [1] worldwide. And even though every year more countries pass laws related to best practices on seat-belts, speed, motorcycle helmets or drink-driving, this number is not expected to decrease notably any time soon, unless we erradicate the biggest source of the problem: us. The sooner we manage to build the technology necessary to autonomously drive vechicles, the smaller the number of road-related accidents will probably take place.

Researchers, like Chris Urmson, now CEO of the start-up company Aurora Innovation [2] and former head of Google's self-driving-car project [3], have been working on the driverless car idea for a couple of decades now [4]. Even though it will still take some time until we can reach level 5 of autonomy, according to some sources [5] we are slowly getting there.

Academia has a lot to say in the development of self-driving cars, being the starting point of many tools and techniques used in the autonomous driving industry today. Driverless car races like the Darpa Grand Challenge [4] back in 2004 and Roborace [6] nowadays represent the public window universities have to show their research on the topic of autonomous vehicles.

Precisely the second driverless car race challenge mentioned above (Roborace) is the motivation of this work. The particularity of Roborace, which takes place in Formula E race tracks (Figure 1), is that the teams participating do not build the car themselves. On the contrary, the vehicle is provided by the organization of the competition and is exactly the same for all teams. The only difference among participants is the software they upload into the race car.



(a) (b)

Figure 1 Example views of Roborace race tracks.

As one of the participants of the Roborace challenge, the team from TUM is currently building the different software pieces to successfully autonomously drive the Robocar (Figure 2). In this work, we address the issue of scene understanding by using the front cameras (stereo rig) the vehicle incorporates.

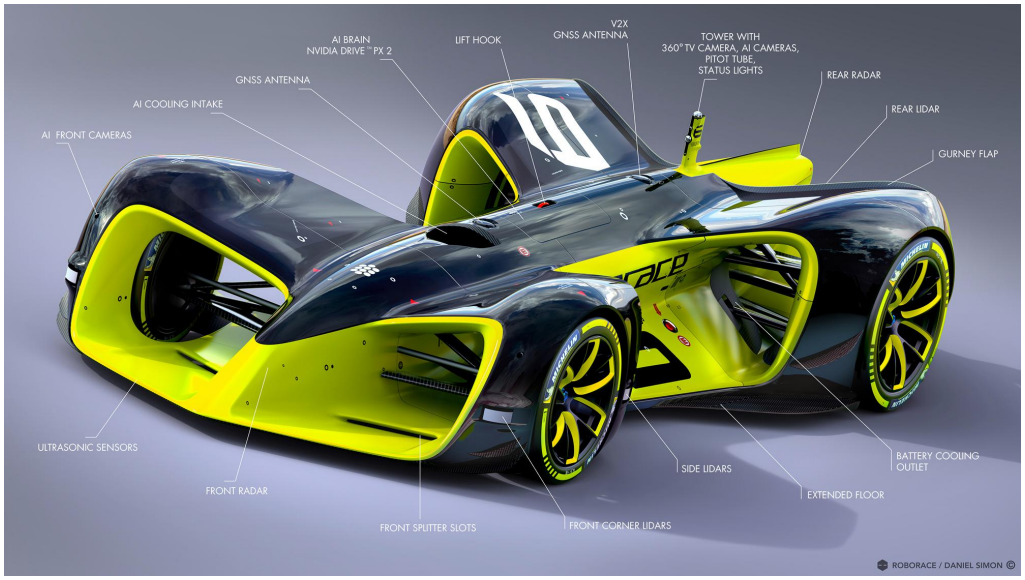


Figure 2 Robocar, the vehicle from the Roborace competition. [6]

1.2. Purpose of the research

The main goal in this research is to develop and implement a computer vision pipeline that allows a self-driving car (more specifically, the Robocar) extract information from its environment by using a camera situated in its front part. Since this work is developed within the Roborace competition, the information we ultimately want to compute is where the vehicle is located with respect to the sides of the road in the Roborace race tracks.

To do so, we propose the following objectives:

1. Train a DNN (Deep Neural Network) to differentiate between road and fence in images similar to the ones shown in Figure 1.
2. Train a DNN to do depth estimation in order to generate a 3D Point Cloud of the scene.
3. Put both networks together into a pipeline that, given a single input image of the scene, outputs relevant information about the position of the vehicle within the road.

1.3. Structure of the thesis

This work has been organized into five chapters. To start with, in Chapter 2 we will present the state of the art in the field of visual perception in self-driving cars. We begin by analyzing the traditional approach to the problem of lane line detection, to then shift towards the more recent, exciting field of DL (Deep Learning), studying two very relevant applications: semantic segmentation and depth estimation.

After having settled the theoretical basics upon which our research is based, in Chapter 3 we give an overview of the computer vision pipeline we have developed in this work, which we will refer to as "SemanticDepth", capable of locating a vehicle within the road by receiving a single image of the scene as input.

In Chapter 4 we dive into the core of this pipeline, explaining all the different parts that enable it extract relevant measures about the scene viewed in the input image. To accomplish this, it fuses two DNNs: one in charge of doing semantic segmentation and a second one capable of obtaining a disparity map from the input image. We will first present how we implemented the semantic segmentation network (Section 4.1) and how we developed our own dataset for training this segmentation network (Section 4.2). Subsequently, in Section 4.3 we will comment on how the depth estimation network we make use of in SemanticDepth works, as well as why we could not train this architecture on our own Roborace dataset of stereo images (Section 4.4).

To close up Chapter 4, in Section 4.5 we will show how we have put both semantic segmentation and depth estimation together to produce a 3D reconstruction of the scene viewed in the input image, as well as how we post-process the obtained 3D Point Cloud to locate the vehicle within the road.

In Chapter 5 we present the different results obtained in this work. First, we focus on the results achieved by our semantic segmentation implementation (Section 5.1) on various datasets. Then, we present what the depth estimation network we use in this work is capable of (Section 5.2). Finally, we present the results achieved by our pipeline, SemanticDepth, on a set of images taken in the streets of Munich (Section 5.3).

Subsequently, in Chapter 6 we discuss the results presented in Chapter 5, following the same structure: firstly, discussion of the results obtained by our semantic segmentation implementation; secondly, comments on the results outputted by the depth estimation network we use; thirdly, comments on the performance of our SemanticDepth pipeline on images featuring streets of Munich.

Finally, in Chapter 7 we will give a short summary of our work and comment on what we consider the next steps should be in order to improve what has been already achieved.

Additionally, in Appendix A we present a list of some of the most relevant semantic segmentation architectures that have emerged since 2014. In Appendix B we describe the process of freezing and optimizing a segmentation model, so that it becomes both ligther and faster, two relevant characteristics during inference. Concerning the depth estimation task, Appendix C offers a general view on stereo vision, from the pinhole model to the generation of depth maps.

2. State of the Art

2.1. Lane Detection Systems

For the past years, one of the most common scene understanding tasks in the realm of intelligent vehicles has been lane line detection. In many proposed systems [7], lane detection consists in locating the lane line marks on the road by applying the following operations to the input image:

1. Conversion of the input image to grayscale.
2. Conversion of the grayscale image to binary.
3. Application of different filters on the binary image.
4. Obtaining the edges by applying the Canny edge detector algorithm [7].
5. Extraction of lines by applying the Hough transform [7].

Other lines of research [8] have taken different approaches, employing the flood-fill algorithm to label the connected components of the grayscale images and then extracting the largest connected component (which is the road region) from the labelled image. Finally, the lane marks are equally extracted from the connected components. In all previously mentioned systems, various challenges are encountered: low quality lane marks, shadows, sharp curves, merging lanes, writings and other marks on the road or unusual pavement materials.

To mitigate the effect of these and many other challenging situations, different approaches were already proposed back in 2008 [9, 10]. The novel idea proposed was taking a top-view of the image, called the IPM (Inverse Persperctive Mapping). This image was then filtered using selective Gaussian spatial filters and then thresholded by keeping only the highest values. Straight lines were detected using Hough transform and a novel RANSAC spline fitting step was performed to refine the detected straight lines and correctly detect curved lanes (Figure 3).

(a) Splines before post-processing in blue

(b) Splines after post-processing in green. They appear longer and localized on the lanes

Figure 3 Post-processing splines. [10]

All previously mentioned approaches are of great importance, and many vehicles being produced nowadays are implementing a LDWS (Lane Departure Warning System) as part of the vehicle's ADAS (Advanced Driver Assistance Systems). However, the limitations of a LDWS in terms of enlarging the vehicle's scene understanding are evident. Take for example a city, where lanes may not even be marked with lines as in highways. In such an environment, a different approach is necessary, and semantic segmentation has arisen as one of the most adequate tool for solving part of the problem of scene understanding in autonomous driving.

In Section 2.3 we will dive into understanding how this recently developed technique works. But before that, in Section 2.2 we will set the basics on which semantic segmentation is built: DL.

2.2. Deep Learning

The recent advances in computing power and especially in GPUs (Graphics Processing Units) has lead to astounding improvements in the fields of AI (Artificial Intelligence), ML (Machine Learning) and DL. The general public often struggles differentiating among these three terms. The main doubt is usually whether they refer to they same thing or rather depict different areas of research. The answer resides in Figure 4. In it we observe that DL is basically an approach to ML (a subset of AI) that uses DNNs. DL uses this one tool (DNNs) to accomplish an incredible array of objectives, from speech recognition to driving a car.

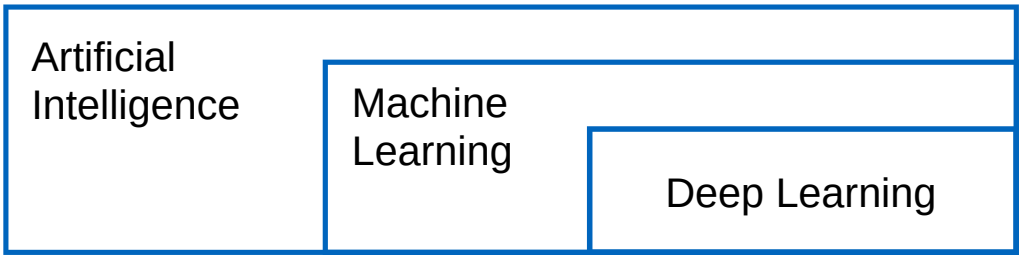


Figure 4 Relationship between Artificial Intelligence, Machine Learning and Deep Learning.

Before diving into how DL is helping cars driving themselves, we must begin describing the perceptron, the fundamental unit of NNs (Neural Networks), and how to combine these units into a simple NN (Section 2.2.1). Then, we will be capable of understanding what is a CNN (Convolutional Neural Network) (Section 2.2.3) and how they can be used for two different tasks from which the vision system of a car can benefit to a great extent, namely semantic segmentation (Section 2.3) and depth estimation 2.4.

2.2.1. Neural Networks

NNs are an incredible tool to extract meaning from complex, imprecise and noisy data, being able to learn patterns usually not visible to humans. To do so, they try to emulate the human brain, in the sense that their structural constituents (the so called artificial neurons) behave like the human neurons, emulating the process of collecting input signals and generating an output when reaching a certain threshold. An overview of a single artificial neuron can be seen in Figure 5.

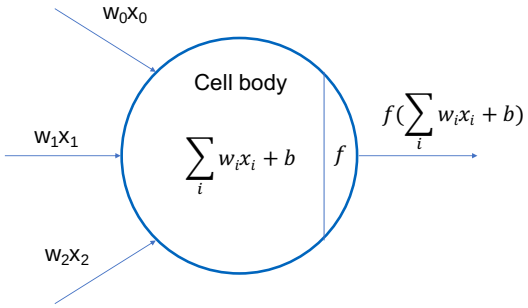


Figure 5 Model of an Artificial Neuron [11].

A single neuron is very efficient at building decision boundaries in order to organize data into different classes. To accomplish this task, we first need to train the neuron, *i.e.*, find the best set of parameters (weights and bias) that minimizes an error function, *e.g.*, the L2 or euclidean norm, of the mismatch between the target and the actual output of the neuron for every data point p of the existing data (Expression 2.1). This technique is known as the Gradient Descent Learning Rule.

$$E(t) = \sum_{p=1}^{P_t} (t_p(t) - out_p(t))^2 \tag{2.1}$$

Note that with a single neuron we can only separate data whose distribution matches the activation function of the neuron (linear, sigmoidal, tanh, ReLU (Rectified Linear Unit)). In order to learn arbitrary data and not just such that happens to fit the given neuron's transfer function, we can combine multiple neurons in a NN (Figure 6), using feed forward connectivity between the so-called hidden layers and the input and output layers, resulting in a fully-connected network.

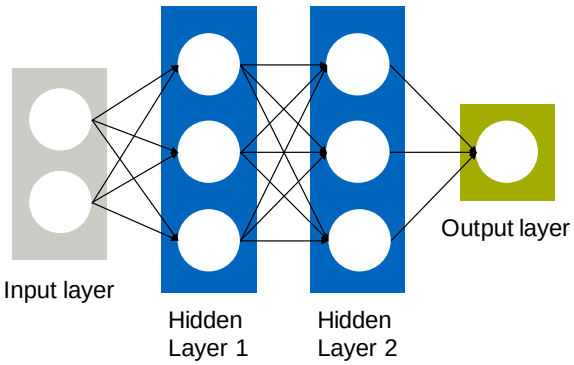


Figure 6 Neural Network model with two input neurons, two hidden layers with three neurons each and an output layer with one neuron [11].

To train a NN we again use the Gradient Descent Learning Rule. As mentioned before, this technique looks for the minimum of the error function (in this case, of the whole network) in weight space. In essence, the error produced in the output neurons will be reintroduced into the network

by an algorithm called error backpropagation. More specifically, we will have to compute what is known as the error signal δ_i of every neuron i in the network. With this error signal, we could then compute the weight update Δw_i as in Expression 2.2, where the subindices i and k correspond to neurons in the current and anterior layers, respectively.

$$\Delta w_{ik}(t) = \delta_i(t) \cdot out_k(t) \tag{2.2}$$

Note as well that the error signal $\delta_i(t)$ depends on the error signals from neurons j of the so-called posterior layers (Equation 2.3). This relationship encodes the backpropagation of the error.

$$\delta_i(t) = f'(net_i(t)) \cdot \sum_{j \in P} \delta_j(t) \cdot w_{ji}(t) \tag{2.3}$$

Without going into greater detail regarding this Equation, we can get an idea of how backpropagation works for a simple NN. And if we were to keep adding layers to the NNs in Figure 6, error backpropagation algorithm would still apply and we would be enlarging the space of functions the NN would be capable of producing. This concept is precisely what DL exploits: stacking multiple hidden layers between the input and output layers of a NN.

Before diving into DNN, we must address a very important concept concerning ML systems in general and NNs in particular: hyperparameters and data splitting.

2.2.2. Setting hyperparameters. Data splitting

In ML, and therefore also in NNs, hyperparameters are parameters which must be set before learning starts, *i.e.*, they are not learned from the training data. Besides, hyperparameters are very problem-dependent. For each problem, one must try them all out and see what works best for that specific case.

Setting hyperparameters that work best on the data we train on is not a good choice. The most common practice is dividing all our data into three subsets, namely train set (70%), validation set (15%) and test set (15%) (Figure 7). We then train our algorithm with many different choices of hyperparameters on the training set, evaluate on the validation test and pick the set of hyperparameters which perform best on the validation set. Finally, we can run our trained model once on the test set and obtain an accuracy value that defines our model's performance.

Figure 7 Data splitting into train, validation and test sets [11].

2.2.3. Deep Neural Networks. Convolutional Neural Networks.

DL is just the next logical step in the field of NNs. The deeper the architecture is, *i.e.*, the more layers it has, the greater the number of features it will be able to extract from the input.

A state-of-the-art DNN that is nowadays broadly used in many areas, *e.g.*, object detection or text recognition, is the CNN or ConvNet, which was first proposed in 2012 by Alex Krizhevsky [12]. The only difference with an ordinary DNN is that a CNN makes a explicit assumption about the input, which allows it to encode certain properties into its architecture. These properties then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network [11].

In essence, CNNs make use of the concept of statistical invariance. Take for example an image with a car in it. Ideally, the CNN should be able to say that the image contains a car, independently of where in the image the car is. In practice, ConvNets accomplish this by using the so-called convolutional layers.

The main difference between a convolution layer and a fully-connected layer is that the former tries to preserve spatial structure by spatially sliding a small filter, *i.e.*, the weights, over the image and computing dot products at every location (Figure 8). This operation outputs an activation map, which encodes certain features of the input volume. Note that when generating the activation map of Figure 8 we have used a stride of 1 and zero-padding of 2, in order to maintain size spatially.

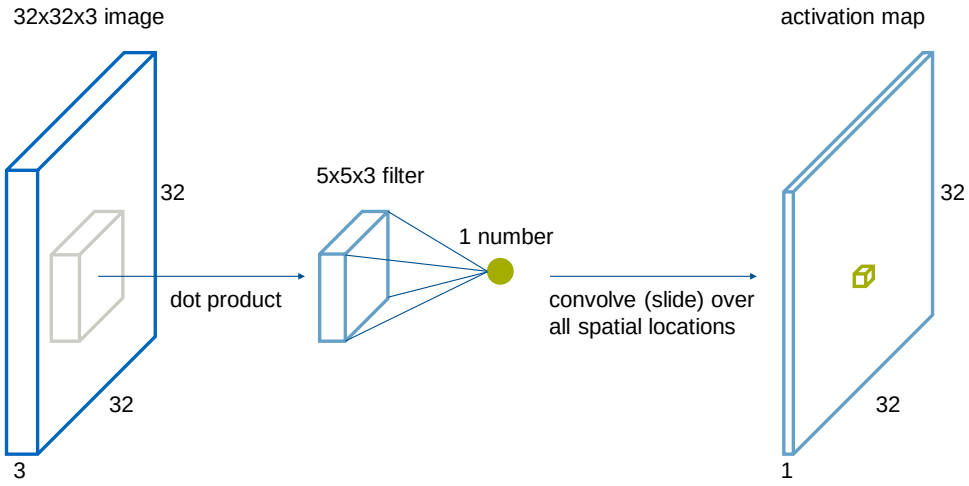


Figure 8 Generation of an activation map by spatially sliding a 5x5 filter over the image with stride 1 and zer-padding with 2. [12]

It is important to understand that the filter always extends the full depth of the input volume, such that the dot product between the filter and the corresponding chunk of the image is mathematically valid. Note as well that every filter generates only one activation map. If we were to use another filter of the same dimensions as the first one, we would obtain a second activation map on top of the one obtained previously (Figure 9). This second activation map would encode different features to the ones encoded by the first activation map.

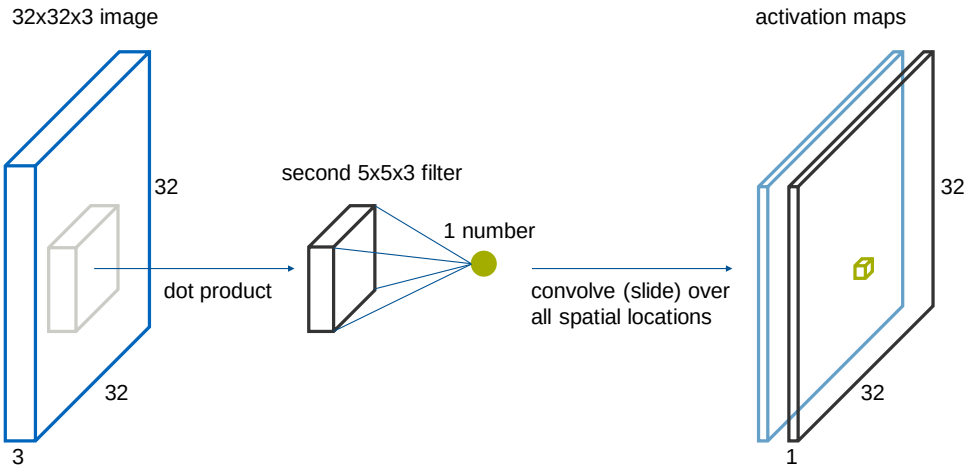


Figure 9 Generation of a second activation map by spatially sliding a 5x5 filter over the image with stride 1 and zer-padding with 2 [12].

Usually, a series of filters are used when doing a convolution operation, thus obtaining a new volume comprised of as many activations maps as filters were used (Figure 10). As a final remark about convolutional layers, it is worth mentioning that they are always followed by an activation function, commonly a ReLU.

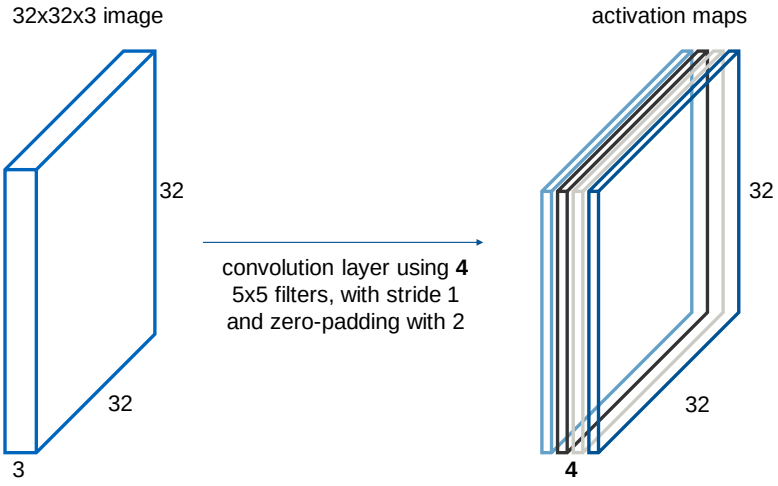


Figure 10 Convolution layer with 4 5x5 filters with stride 1 and zero-padding with 2. [12]

Another basic component of ConvNets is the pooling layer. The intuition behind pooling layers is to make the representations smaller and more manageable, such that we keep the number of parameters in our architecture as small as possible. In practice, a pooling layer operates over each activation map independently, essentially downsampling the input volume in width and height (Figure 11). Note, however, that the the depth, *i.e.*, the number of activation maps, is not modified by the pooling operation.

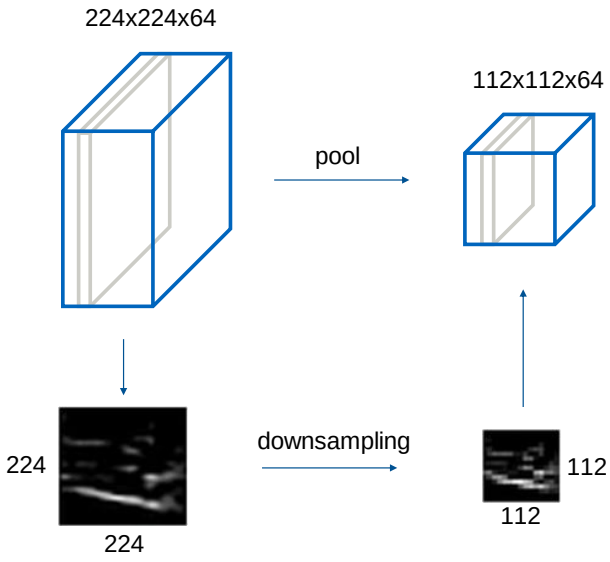


Figure 11 Pooling layer. In essence, a pooling layer downsamples each activation map of the input volume. [12]

A common pooling approach is max pooling. As in convolutional layers, in pooling layers we also have a filter, which represents the region at which we pool over. As an example, if we were to carry out a max pooling operation by sliding a 2x2 filter, like the one shown in Figure 12, over each activation map of the input volume, we would obtain downsampled activation maps, each containing the maximum values for each of the pooled regions.

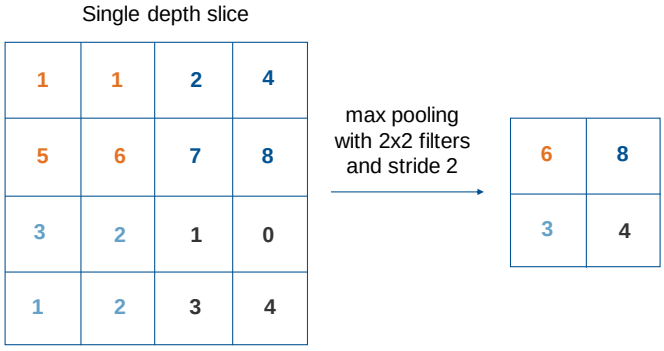


Figure 12 Max pooling operation using 2x2 filters and a stride of 2. [12]

A typical CNN architecture contains all elements we have presented so far: convolution layers (followed by a ReLU activation function), pooling layers and fully-connected layers, organized, for example, as [INPUT - CONV (+ ReLU) - POOL - FC]. More complicated CNNs can be built simply by adding more "CONV (+ ReLU) + POOL" layer pairs (Figure 13).

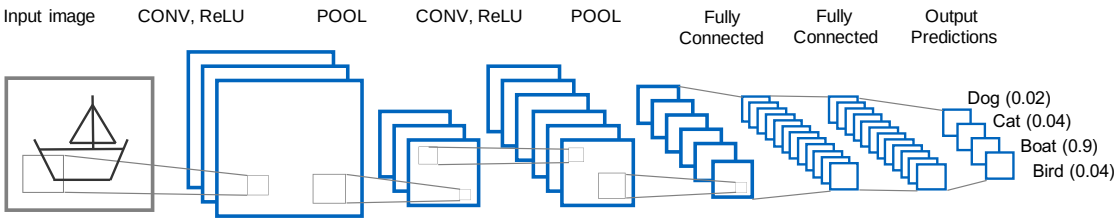


Figure 13 Convolutional Neural Network, based on the architecture proposed by Alex Krizhevsky in 2012. [12]

As we already mentioned above, CNNs are great architectures for classification tasks, such as telling whether there is a pedestrian in an image or not, and with what level of probability. However, they cannot tell where in the picture is the pedestrian, since fully-connected layers, *i.e.*, the final layer of CNNs, do not preserve spatial information.

This task, *i.e.*, telling where in an image is an object, is tackled by semantic segmentation, an essential part of scene understanding in autonomous driving. In Section 2.3 we will cover in detail what is semantic segmentation and what a semantic segmentation architecture looks like. Moreover, we will compare the performance of some of the most important segmentation architectures on a renowned dataset.

2.3. Deep Learning in Semantic Segmentation. Fully Convolutional Networks

Semantic segmentation is in essence understanding an image at pixel level, *i.e.* each pixel in the image is assigned to an object class. Take as an example Figure 14. A semantic segmentation model must not only determine the existance of an object car or of an object road in it, but also the boundaries of each object and their location on the image plane. It is therefore necessary, unlike in classification tasks [12, 13], to obtain dense pixel-wise predictions.

(a) Original image (b) Ground truth

Figure 14 Example of semantic segmentation applied on an image of the Cityscapes dataset [14].

Before DL became mainstream in computer vision, different approaches were taken for semantic segmentation. Ensembles of decision trees that acted directly on image pixels, such as TextonForest [15], or methods to predict human pose from a single depth image, such as Ramdom Forest based classifiers [16], were the state of the art approaches in semantic segmentation.

Then, as with the task of image classifcation, CNNs [12] started gaining huge success on semantic segmentation tasks, although they had to be modified in order to not output a class score but an image of the same resolution as the input.

Initially, a path classification approach [17] was taken, where the label of each pixel was predicted from raw pixel values of a square window centered on it. The reason for using patches was that classification networks have fully connected layers and, therefore, require fixed-size images.

Later on, in 2014, FCNs [18] popularized CNN architectures for dense predictions without any fully connected layers. This approach, considered to be a major breaktrough in computer vision, allowed segmentation maps to be generated for images of any size and was significantly faster compared to the previous patch-classification approach [17]. More precisely, the FCN-8s implementation [18] achieved a 20% relative improvement to 62.2% mean IoU (Intersection over Union) on Pascal VOC2012 dataset [19].

Structuraly, an FCN architecture is comprised of an encoder network followed by a decoder network (Figure 15). The encoder is usually a pre-trained classification network (like VGG [20] or ResNet (Residual Network) [21]), whose goal is to extract features from the image.

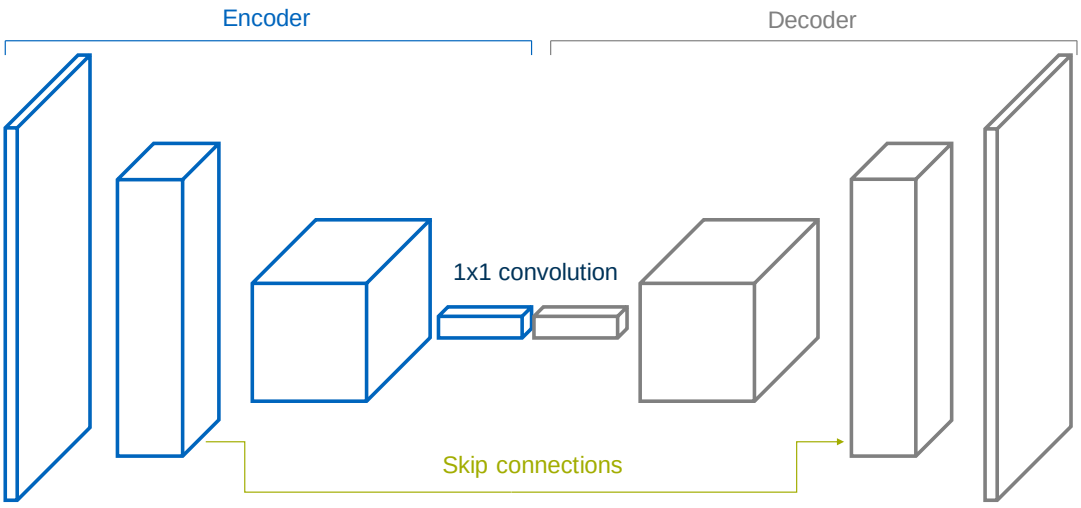


Figure 15 FCN architecture. [18]

On the other hand, the decoder's task is to semantically project this discriminative features (lower resolution) learnt by the encoder onto the pixel space (higher resolution) to get a dense classification, *i.e.*, it upscales the output of the encoder such that it is the same size as the original image. Thus, it results on segmentation or prediction of each individual pixel in the original image.

The great majority of the state-of-the-art semantic segmentation architectures are based on this encoder-decoder schema. Their differences usually reside in the decoder part, each architecture incorporating their own new approaches to it. However, three important techniques are commonly employed in almost all segmentation architectures:

1. Replacement of the final fully-connected layers used in CNNs with 1-by-1 convolutional layers.
2. Upsampling through the use of transposed convolutional layers (also known as deconvolutions).
3. Skip connections.

2.3.1. 1-by-1 Convolutional Layers

FCNs replace the final fully-connected layer of the encoder, *i.e.*, a pre-trained classification network, with 1-by-1 convolutional layers (Figure 16). This causes the output value of the encoder to remain 3-dimensional, instead of being flattened to a 2D output. Thus, spatial information is preserved.

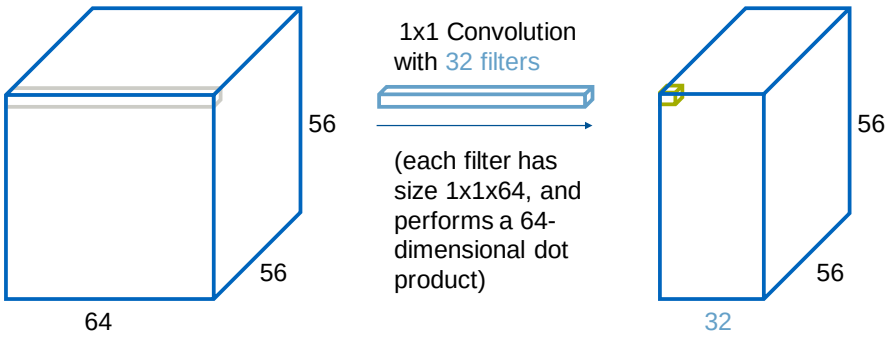


Figure 16 1-by-1 Convolution. [11]

This type of operation is inexpensive, since rather than a convolution, 1-by-1 convolutions are simply a matrix multiplication, and they only add a few parameters to the model.

2.3.2. Upsampling through the use of transposed convolutional layers

After computing the 1-by-1 convolution of the encoder's output, the next step is upsampling the activation maps. This is accomplished through the use of transposed convolutions. Essentially, they are a reverse convolution in which the forward and the backward passes are swapped. Transposed convolutions are often known as deconvolutions, as they undo the convolution operations previously carried out in the encoder.

2.3.3. Skip connections

The third technique that FCNs use is skip connections. One effect of convolutions and encoding in general is that they narrow the scope by looking closely at some features, but lose the bigger picture as a result. If we were to decode the output of the encoder back to the original image size, some information would have been lost in the process.

Skip connections (green line in Figure 15) address this problem. They allow the network to easily retain information from multiple resolution scales by connecting, *i.e.*, doing an element-wise addition operation, the output of one layer to a non-adjacent layer. As a result, the network is able to make more precise segmentation decisions.

Up to this point, we have almost entirely defined the structure of FCNs. The only piece missing is the loss function that the arquitecture uses to quantify its learning. We discuss this next.

2.3.4. Cross Entropy Loss

The final layer of an FCN outputs a volume with the same width and height as the original input image and a depth equal to the number of possible classes, c, such as, road, car or pedestrian. In other words, for every pixel in the original image we have computed c values, known as logits, each of which is giving an idea of how confident the network is about that particular pixel belonging to one of the c classes.

Before comparing logits with labels and computing some kind of loss function that we can use for backpropagation, we must turn logits into probabilities, so that for every pixel the c computed probabilities sum up to 1. We can accomplish this by using the so-called softmax function, $S(\hat{y}_i)$, presented in Equation 2.4, where $\hat{y}_i$ represents the logit or score predicted for class i in a given pixel. The $\sum_{j=1}^{c} e^{\hat{y}_j}$ represents the sum of the logits over all the c classes for a given pixel.

$$S(\hat{y}_i) = \frac{e^{\hat{y}_i}}{\sum_{j=1}^{c} e^{\hat{y}_j}} \tag{2.4}$$

Once every pixel has a probability for each one of the c classes, we can make use of the cross-entropy loss (Equation 2.5). This loss function, $D_p(\hat{y}, y)$, which is defined for every pixel p, gives an idea of the distance between the array of predicted classes, $\hat{y}$, for this particular pixel p (which is an array of length c) and the actual class for that pixel, y, which is usually formatted as one-hot encoding, *i.e.*, as an array of length c where the element corresponding to the true class is 1 and the rest is filled with zeros.

$$D_p(\hat{y}, y) = -\sum_{i=1}^{c} y_i ln \hat{y}_i \tag{2.5}$$

The cross-entropy loss outputs a value for each pixel, which represents how badly the class of the pixel has been predicted. To be able to apply backpropagation and train the network, we need to have a loss function for the whole input image. Equation 2.6 is used to compute this loss, adding up all cross-entropy losses of every pixel, p, over the total number of pixels, P, in the image.

$$L_{image} = \sum_{p=1} PD_p \tag{2.6}$$

Finally, we can compute a loss L for all images in a certain training batch of N images (Equation 2.7). Note that we must include a regularization term to the computation of the final loss, $\lambda R(W)$, where λ is the regularization strength (hyperparameter), R is the regularization function and W represents the weights of our network. In the next section (2.3.5) we will cover regularization in greater detail.

$$L = \frac{1}{N}(\sum_{image=1}^{N} L_{image}) + \lambda R(W) \tag{2.7}$$

2.3.5. Regularization

Regularizing means applying artificial constraints to your network that implicitly penalize the complexity of the model. We can also view regularization as a way of telling the network to find the simplest set of weights that fits the training data, so that it can then generalize to test data, *i.e.*, to not do overfitting of the training data. Note that regularization is not specific of FCNs or semantic segmentation, but a general technique employed in DL.

One commonly used regularization technique is the L_2 regularization (Equation 2.8), which is precisely the $R(W)$ function we presented above (Section 2.3.4, Equation 2.7). By adding the L_2 euclidean norm of all weights multiplied by λ to the loss, we penalize large weights.

$$R(W)_{L_2} = \frac{1}{2} \|W\|_2^2 \tag{2.8}$$

Another regularization technique that emerged recently is dropout. To explain it, we will present a simple example. Suppose we have a layer that connects to another layer. The values that go from the first layer to the next are often called activations, as we already mentioned in Section 2.2.1. Essentially, for every example we train the network on, the dropout technique tells us to set a certain percentage of these activations to zero, usually half of them. Selecting which activations will be destroy is done completely randomly for every layer in the network and every input image.

The intuition behind dropout is that the network can never rely on any given activation to be present, because it might be set to zero at any given moment. Therefore, the network is forced to learn a redundant representation for any given input, in order to make sure that at least some of the information about the input remains in the network. Forcing the network to learn redundant representations might seem inefficient, but is key in order to make the model more robust and to prevent overfitting.

2.3.6. Intersection over Union Metric

In order to evaluate the performance of a model in the semantic segmentation task, the IoU metric is usually employed. It is literally the intersection set divided by the union set (Equation 2.9).

$$IoU = \frac{Intersection \quad set}{Union \quad set} \tag{2.9}$$

The intersection set is the result of applying an AND operation to two sets, namely the set of pixels that truly belong to a certain class and the set of pixels that our model has predicted to belong to that class. If a pixel exists in both sets, then we put it into the intersection set. In other words, for each class, *e.g.*, car, pedestrian or fence, the intersection is defined as the number of pixels that are both truly part of that class and are classified as part of that class by the network.

The union set is the result of applying an OR operation to the two sets we mentioned before. If a pixel belongs to at least one of the two sets, then we put it into the union set. Therefore, the union set is defined as the number of pixels that are truly part of a class plus the number of pixels that are classified as part of that class by the network.

One can rapidly deduce that the intersection set is smaller or equal to the union set (Equation 2.10).

$$Intersection \quad set <= Union \quad set \tag{2.10}$$

The ratio then tells us the overall performance per pixel per class. Of course, we can also infer that the ratio must always be smaller or equal to 1 (Equation 2.11).

$$\frac{Intersection \quad set}{Union \quad set} <= 1 \tag{2.11}$$

Usually, we calculate the mean IoU of the network, which is just the average of all the IoU values for all the classes. This gives an idea of how well the model handles all the different classifications for every single pixel.

2.3.7. Work derived from the original FCN architecture. Comparison of some state-of-the-art semantic segmentation architectures

Up to this point, we have thoroughly defined what Long *et. al* [18] FCN architecture looks like. Many of the semantic segmentation models that followed were deeply fundamented on it, to the point that nowadays the concept of FCN does not only refer to the original architecture [18], but

also to the broader concept presented in that work. In this section we will present a short summary of the most relevant work that followed the FCN-8s architecture, as well as a comparison of their performance in a state-of-the-art dataset, namely Cityscapes [14].

Kendal et al. [22] (SegNet) proposed to upsample the features outputted by the encoder with a large decoder segment that performs finer unpooling by using indices of the encoder's max-pooling blocks [23]. Other works like [24] (DeepLab) proposed to refine the coarse output by using CRFs (Conditional Random Fields), and works like [25] (CRFasRNN) proposed to integrate them inside the convolutional architecture. But relying on algorithms like CRF to refine segmentation highly increases the network's computational overload. [23]

Very recent work has achieved top performance by adapting ResNets [21] into the segmentation task, and combining them with dilated convolutions [26], which allows for exponential expansion of the receptive field without any loss of resolution [23]. The work in [27] (DeepLab2) combines a ResNet-101 with spatial pyramid pooling and CRFs to improve performance. Lin et al. [28] (RefineNet) proposed a multi-path refinement network that exploits all the information available along the downsampling process, in order to enable high resolution predictions using long-range residual connections [23]. Pohlen *et al.* [29] (FRRN) proposed a ResNet-like architecture that combines multi-scale context with pixel level accuracy by using two processing streams, one at full resolution and another that performs downsampling operations. Ghiasi *et al.* [30] (LRR) proposed a complex architecture that constructs a Laplacian pyramid to process and combine features at multiple scales.

All these approaches achieve top performance but the required resources make them extremely expensive in terms of resources in modern GPUs, becoming unfeasible for ITS (Intelligent Transportation System) applications [23]. In the recent ENet [31], on the contrary, efficiency is the ultimate goal, but many important sacrifices are made in the network layers to gain efficiency at the expense of a lower classification performance, compared to other approaches.

An architecture that sits in the middel, aiming at the best possible trade-off between accuracy and efficiency, without neglecting any of them independently, is the recent [23] (ERFNet). This architecture, based on a novel residual block that uses factorized convolutions, can run at several FPS (frames per second) on a modern GPU, even on embedded devices, which can be mounted on a vehicle. [23]

In Appendix A we explain in greater detail some of the semantic segmentation architectures cited above, listing down their contributions as well as showing their benchmark scores (mean IoU) on VOC2012 test set [19] and on Cityscapes test set [14].

As an overview of what is shown in Appendix A, Table 4 presents the class-IoU results of each one of them on the test set of Cityscapes [14] and their fwt (forward pass time) (or inference time) on a modern GPU. [23]

Table 4 List of various state-of-the-art semantic segmentation architectures and their results (Class-IoU and fwt) on the Cityscapes test set, as reported in the online benchmark of the dataset. [23]

Network	**Pretrain**	Class-IoU	fwt
DeepLabv3 [32]	ImageNet	85.7	n/a
PSPNet [33]	ImageNet	85.4	n/a
RefineNet [28]	ImageNet	73.6	n/a
FRRN [29]	-	71.8	n/a
Adelaide-cntxt [34]	ImageNet	71.6	35+
DeepLabv2-CRF [27]	ImageNet	70.4	n/a
LRR-4x [30]	ImageNet	69.7	n/a
ERFNet [23]	ImageNet	69.7	0.024
Dilation10 [26]	ImageNet	67.1	4.0
DPN [35]	ImageNet	66.8	n/a
FCN-8s [18]	ImageNet+Pasc	65.3	0.500
DeepLab [24]	ImageNet	63.1	4.0
CRFasRNN [25]	ImageNet	62.5	0.700
SQ [36]	ImageNet	59.8	0.060
ENet [31]	-	58.3	0.013
SegNet basic [22]	ImageNet	56.1	0.060

2.4. Deep Learning in Depth Estimation

Extracting depth information is key in autonomous driving. This task is usually accomplished through the use of LIDAR (Light Detection and Ranging) and RADAR (Radio Detection and Ranging) systems, which have the advantage that they do not require light to perceive the environment. However, there are situations in which these systems may fail. Such is the case of a LIDAR trying to detect a wall made out of glass, where the rays emited by the LIDAR would simply go through the glass. But above all, there is the matter of cost. LIDAR and RADAR are notably expensive, at least much more expensive that another system with which we can also obtain depth information: RGB (Red-Green-Blue) cameras.

Depth estimation from images is indeed an interesting topic in self-driving cars. Both because using RGB cameras, as mentioned above, is much cheaper than using LIDAR and RADAR and also because cameras produce a dense representation of the world they see, meaning that the level of information encoded in an image is higher than that provided by a LIDAR. For these reasons, there has been a broad range of different approaches tackling the problem of depth estimation, either using pairs [37], several overlapping images taken from various viewpoints [38], temporal sequences [39], or assuming a fixed camera, static scene, and changing lighting [40].

Whereas these approaches need to have more than one input image of the view of interest, monocular depth estimation only requires a single input image, and does not need to make any assumption about the scene geometry either. Monocular depth estimation is highly interesting because perfectly synchronized stereo images are not easy to obtain during inference, *i.e.* in real time systems. In the case of Roborace, stereo pairs taken by the vehicle's stereo rig are indeed insufficiently synchronized, a problem on which we will comment in Section 4.4.1. To overcome this deficiency, in this work we use an algorithm developed by Godard [41] (monodepth) in order to estimate depth from just a single source image. Therefore, what follows in this Section 2.4 will be a short summary of Godard's revision of related work within the field of depth estimation, as well as a short explanation of Godard's algorithm.

2.4.1. Learning-Based Stereo

Almost all stereo estimation algorithms are based on computing the similarity between every pixel in the first image and every other pixel in the second one. The stereo pair is usually rectified. Therefore, disparity estimation (refer to Appendix C for an explanation on stereo vision concepts, such as disparity) can be tackled as a 1D search problem for every pixel. Approaching this 1D matching problem as a supervised learning problem and training a function to predict the correspondences has achieved remarkably superior results [42, 43]. Moreover, by tackling this binocular correspondence search as a multi-class classification problem one can obtain improvements both with regards to quality of results and speed [44]. [41]

But instead of simply learning the matching function, Mayer *et al.* [45] made use of a fully CNN (DispNet) which computes the correspondence field between two images. During training, they attempt to directly predict the disparity for each pixel by looking for the minimum of a regression training loss. It is worth mentioning that DispNet's architecture is somehow the evolution of their previously developed end-to-end deep optical flow network [46]. [41]

All approaches above require large quantities of accurate ground truth disparity data and stereo pairs at training time. This type of data is not easy to acquire for real world scenes, so it is very common that these methods make use of synthetic data for training. The problem here is that, even though we are gradually becoming better at producing more realistic synthetic data, *e.g.* [47], for every new application scenario we still need to manually create new content. [41]

2.4.2. Supervised Single Image Depth Estimation

In contrast to the stereo estimation approach (Section 2.4.1), in single-view, or monocular, depth estimation we only have single images available at test time. Saxena *et al.* [48] proposed a patch-based model (Make3D) which first over-segments the input image into patches and then estimates the 3D location and orientation of local planes to explain each patch. A linear model trained offline on a dataset of laser scans is used to make the predictions of the plane parameters. This and other planar-based approaches, *e.g.* [49], have a big disadvantage: they can have difficulty modeling thin structures and, as predictions are made locally, they lack the global context required to generate realistic outputs. Instead of hand-tuning the unary and pairwise terms, Liu *et al.* [50] introduce a CNN to learn them. [41]

Eigen *et al.* [51] [52] managed to produce dense pixel depth estimates using a two scale deep network trained on images and their corresponding depth values. Instead of relying on hand crafted features or initial over-segmentation, like most other previous approaches in monocular depth estimation had done, they learn a representation directly from the raw pixel values. This successful work was the starting point for many others who added little improvements, such as: CRFs to improve accuracy [53], changing the loss from refression to classification [54] or simply using different, more robust loss functions [55]. Like the previous stereo-based methods, these works all require high quality, pixel aligned, ground truth depth at training time. [41]

2.4.3. Unsupervised Depth Estimation

With the introduction of DL, several approaches in depth estimation have been recently proposed. The Deep3D network of Xie *et al.* [56] tries to generate the corresponding right view from an input left image, *i.e.*, the source image, by using an image reconstruction loss. This method produces a distribution over all the possible disparities for each pixel, where the resulting synthesized right image pixel values are a combination of the pixels on the same scan line from the left image, weighted by the probability of each disparity. The disadvantage of this image formation model is that increasing the number of candidate disparity values too much also increases the memory consumption of the algorithm, thus making it very difficult to scale this approach to bigger output resolutions. [41]

Another similar work is that of Garg *et al.* [57], which also trains a network for monocular depth estimation by making use of an image reconstruction loss. However, the image formation model they use is not fully differentiable, meaning that some of the units' transfer functions, *e.g.*, *tanh*, *sigmoid* or *ReLU*, are not differentiable. To overcome this problem, a Taylor approximation is performed in order to linearize their loss, but this results in an loss function that is more difficult to optimize. [41]

2.4.4. Monodepth - Unsupervised Monocular Depth Estimation with Left-Right Consistency

Godard proposes a fully convolutional DNN loosely inspired by the supervised DispNet architecture of Mayer *et al.* [45]. By posing monocular depth estimation as an image reconstruction problem, he can solve for the disparity field without requiring ground truth depth. However, only minimizing a photometric loss (Godard's first approach was to only use the left image to reconstruct the right image) can result in good quality image reconstructions but poor quality depth. They overcome this problem by including a left-right consistency check in their fully differentiable training loss. By doing so they improve the quality of their synthesized depth images. Note that this type of consistency check is commonly used as a post-processing step in many stereo methods, *e.g.*, [42], but Godard incorporates it directly into the network. [41]

2.4.4.1 Depth estimation as Image Reconstruction

Given a single image I at test time, the goal is to learn a function f that can predict the per-pixel scene depth, $\hat{d} = f(I)$. As Godard notes, most existing learning based approaches treat this as a supervised learning problem, where they have images and their corresponding target depth values at training. However, one can rapidly deduce that this it is not yet practical to acquire such ground truth depth data for a large variety of scenes. Even laser scanners can fail to produce precise data in scenes featuring movement or reflections. Alternatively, Godard poses depth estimation as an image reconstruction problem during trainig, *i.e.* given a calibrated pair of binocular cameras, by learning a function that is able to reconstruct one image from the other, the algorithm will have learned during the process something about the 3D shape of the scene. [41]

Specifically, at training time, the algorithm has access to two images I^l and I^r, corresponding to the left and right color images from a calibrated stereo pair, captured at the same moment in time (*i.e.* synchronized). Instead of trying to directly predict the depth, Godard's network attempts to find the dense correspondence field d^r that, when applied to the left image, would allow to reconstruct the right image. The reconstructed image $I^l(d^r)$ is referred to as $\hat{I}^r$. Similarly, the left image can also be estimated given the right one, $\hat{I}^r = I^r(d^l)$. Assuming that the images are rectified (Appendix C), d corresponds to the image disparity, which the model will learn to predict. Given the baseline distance b between the cameras and the camera focal length f, the depth $\hat{d}$ can finally be computed from the predicted disparity as $\hat{d} = bf/d$. [41]

2.4.4.2 Depth estimation Network

Godard's network estimates depth by inferring the disparities that wrap the left image to match the right one. The key aspect of their method is that they can simultaneously infer both disparities (left-to-right and right-to-left) using only the left input image, obtaining better depths by enforcing them to be consistent with each other. [41]

Monodepth generates the predicted image with backward mapping using a bilinear sampler, which results in a fully differentiable image formation model. Figure 17 shows the three different strategies used for backward mapping: *naive*, *no LR* and *final* strategy. Naively learning to generate the right image by sampling from the left one will produce disparities aligned with the right image (target).

However, the objective is to obtain a disparity map that aligns with the input left image, meaning the network has to sample from the right image. Godard proposes another approach, which is training the network to generate the left view by sampling from the right image, thus creating a left-view-aligned disparity map (referred to as NoLR in Figure 17).

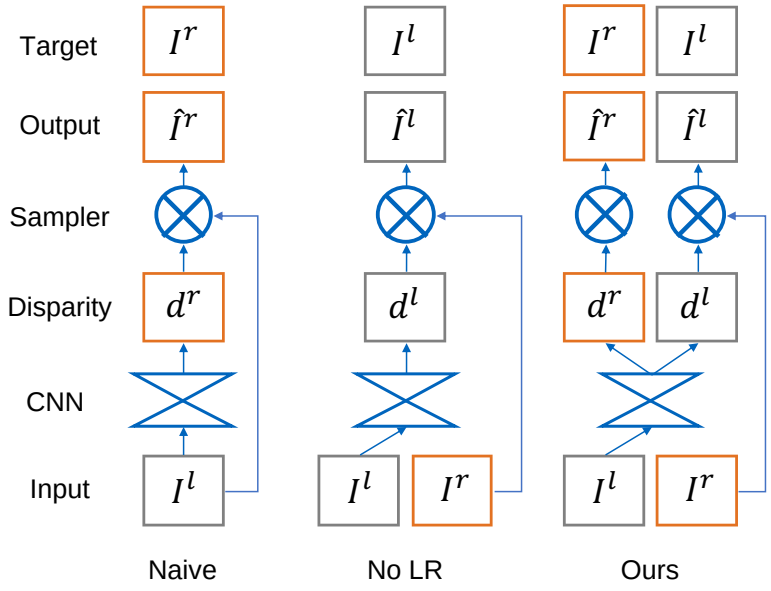


Figure 17 Sampling strategies for backward mapping. With naive sampling the CNN produces a disparity map aligned with the target instead of the input. No LR corrects for this, but suffers from artifacts. Godard's final approach uses the left image to produce disparities for both images, improving quality by enforcing mutual consistency. [41]

Even though this solves for the initial problem, the inferred disparities show 'texture-copy' artifacts and errors at depth discontinuities as seen in Figure 18. This is solved by training the network to predict the disparity maps for both views by sampling from the opposite input image. This approach still only requires a single left image as input to the CNN and the right image is only used during training (see 'Ours' (referring to Godard's final method) in Figure 17).

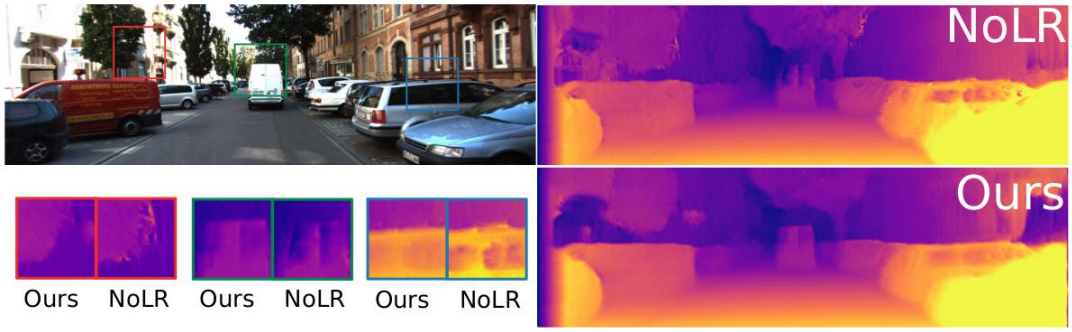


Figure 18 Comparison between Godard's method with and without the left-right consistency. Their consistency term produces superior results on the object boundaries. Note that 'Ours' referrs to Godard's final method. [41]

In summary, the monodepth architecture, while inspired by DispNet [45], features various important modifications that allow to train a monocular depth estimation network without requiring ground truth depth, only rectified stereo pairs.

2.5. From Monodepth Disparity Maps to 3D Point Clouds

The disparity values that the monodepth [41] architecture predicts for a given image are relative to the image's width. Therefore, we need to scale the disparity map with the image's width resolution (Equation 2.12) before converting the former to depth.

$$real\_disparity = input\_image\_width \cdot monodepth\_disparity \tag{2.12}$$

With the correct disparity map we can now apply standard stereo vision theory (Appendix C.2). In particular, we can apply Equation 2.13, where (u, v) are the coordinates of each pixel in the disparity map, d is the disparity value of each pixel, (X, Y, Z, W) are the homogeneous coordinates of the 3D points in the camera space, (c_x, c_y) are the pixel coordinates of the principal point, f is the focal length of the camera in pixels and b is the baseline in meters of the stereo pair which took the images on which the monodepth [41] model was trained (Cityscapes, in our case).

$$Q \begin{bmatrix} u \\ v \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix}, \quad where \quad Q = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 0 & f \cdot b \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.13}$$

In its current form, monodepth supposedly can only be applied to images with a focal length and aspect ratio identical to the dataset it was trained on. According to the author of the monodepth architecture, good results are not to be expected when testing on images with different characteristics from those of the training set, since it is not completely clear how to convert the disparities into depths. Godard points out, however, that a reasonable but not proven way to do this could be to measure an object of known dimensions in the test images and use this measure to find a reasonable scaling factor for the disparity.

3. Concept of the work. SemanticDepth - A Pipeline to merge Semantic Segmentation with 3D Vision

SemanticDepth (Figure 19) is the computer vision pipeline developed in this work. It aims to improve the situational awareness of the Robocar in the Roborace competition by using its frontal left camera, but it has been tested to also work in other environments, such as the streets of a city like Munich. In this brief chapter, we intend to provide the reader with a broad picture of the pipeline. We will do so by shortly summarizing its key stages.

SemanticDepth has been designed to process Roborace-like images featuring a road and, optionally, two fences to each side of the road. By fusing together two DL models, namely a semantic segmentation model and a monocular depth estimation model, it is able to compute relevant distances in the scene.

The intuition behind why we are using a semantic segmentation model together with a monocular depth estimation model is explained next. On the one hand, a segmentation model, when presented with an image, ultimately outputs 2D masks of the objects it has been programmed to classify. In our case, we want to differentiate between the classes road, fence and background.

On the other hand, a monocular depth estimation model is able to generate the disparity map corresponding to the image it has been given. By overlaying the masks obtained in the segmentation onto the disparity map, we can selectively convert certain regions of the disparity map into 3D Point Clouds. In our case, by using the road and fences 2D masks, we can obtain their corresponding 3D Point Clouds separately, thus accomplishing 3D segmentation of the scene.

We can then separate the Point Cloud corresponding to the fences, which at this point contains both the left and right fences, into separate Point Clouds for each fence, and apply denoising functions to the three Point Clouds we have, namely road, left fence and right fence.

The pipeline's last step is computing any relevant distance about the scene. In our case, we have opted for the following measures. On the one hand, a "naive distance" featuring the width of the road at a certain depth. On the other hand, an "advance distance", which is the result of fitting planes to the 3D Point Clouds of the road and fences first, looking then for the intersection of these planes with each other, *i.e.*, road with left fence and road with right fence, and finally, computing the distance between the two itersected lines at a certain depth.

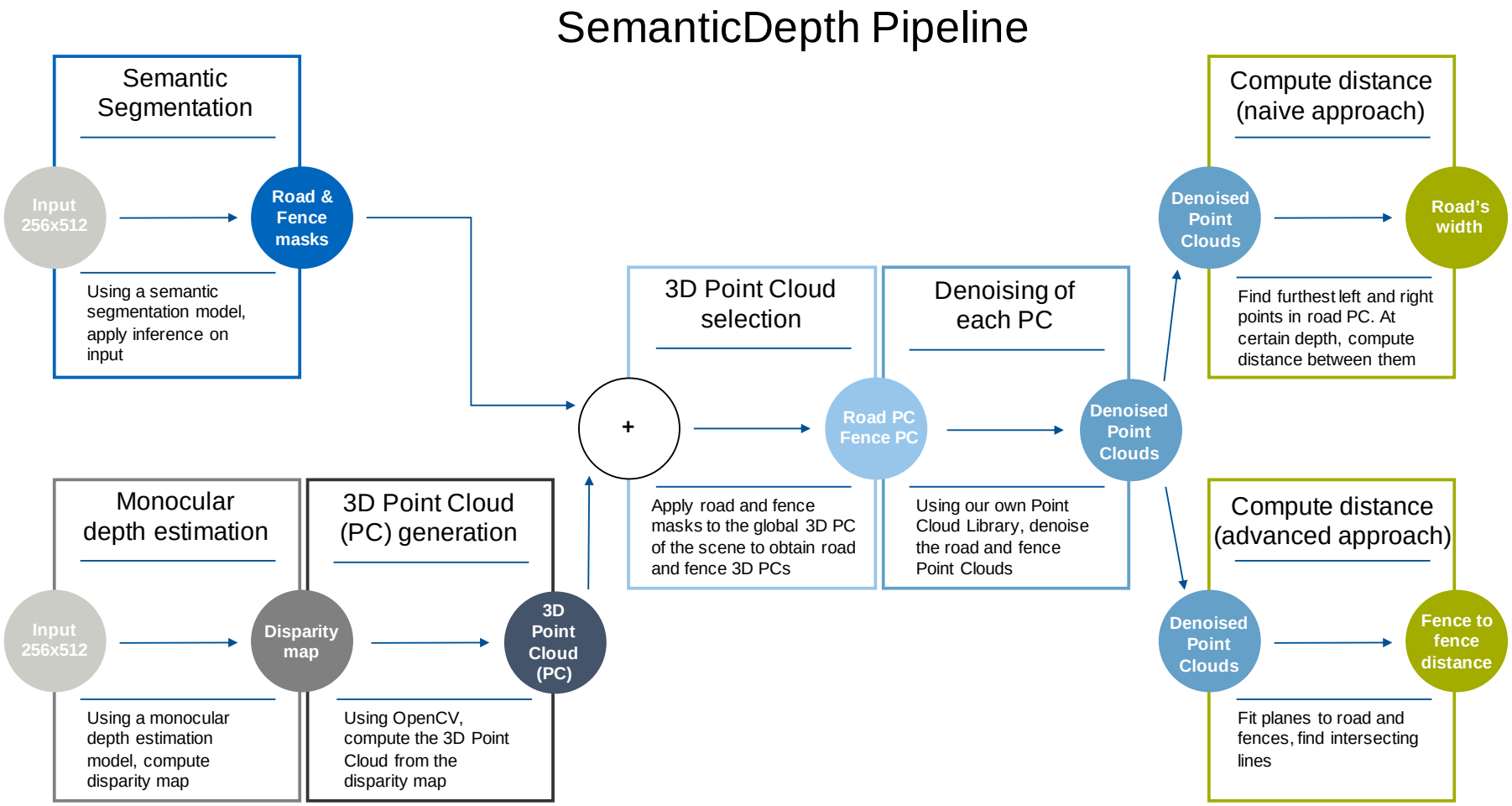


Figure 19 SemanticDepth pipeline.

4. Implementation of the SemanticDepth pipeline

In this chapter we will go over the SemanticDepth pipeline step by step. We will go over each of the DL models separately (sections 4.1 and 4.3). We will also explain how we have put together Roborace datasets for both the segmentation task and the depth estimation task (sections 4.2 and 4.4, respectively). Finally, we will present in greater detail how we have merged both models together into a pipeline (Section 4.5) that extracts relevant distances about the scene.

4.1. Semantic Segmentation Model

In this section we will first explain the different segmentation architectures we considered and why we settled on the FCN-8s (Section 4.1.1). Then, we will cover some details about the implementation of the FCN-8s (Section 4.1.2).

4.1.1. Selection of a Semantic Segmentation Architecture

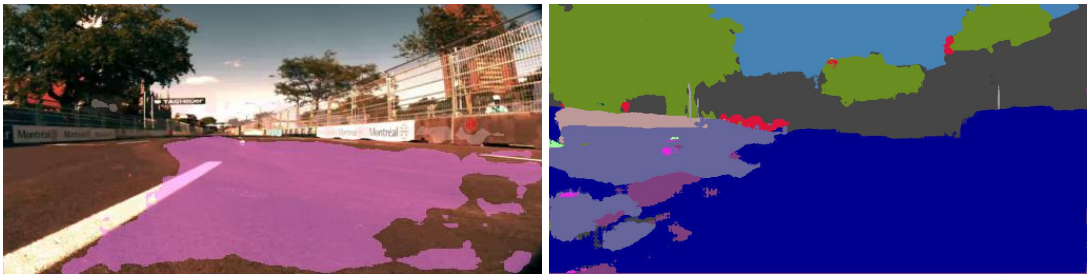
For the task of semantic segmentation, the range of architectures to choose from is fairly large, as we have already shown in Section 2.3, and more specifically in Section 2.3.7. For our particular case, we needed an architecture which was both efficient and accurate.

From the various architectures studied in this work, we gathered information corresponding to their mean class-IoU on the Cityscapes test set and their forward pass time (fwt), as we already showed in Table 4 (Section 2.3.7).

We detected that the ERFNet [23] brings both efficiency and accuracy together, with a mean class-IoU of 69.7% and a forward pass time (fwt) of 0.024 seconds on the Cityscapes test set. In order to qualitatively understand how good this architecture could generalize to environments and conditions it had not been trained on, we tested a pre-computed model of the ERFNet (trained for 150 epochs on the 20 training classes of Cityscapes) on several Roborace images (subfigures 20b and 20d).

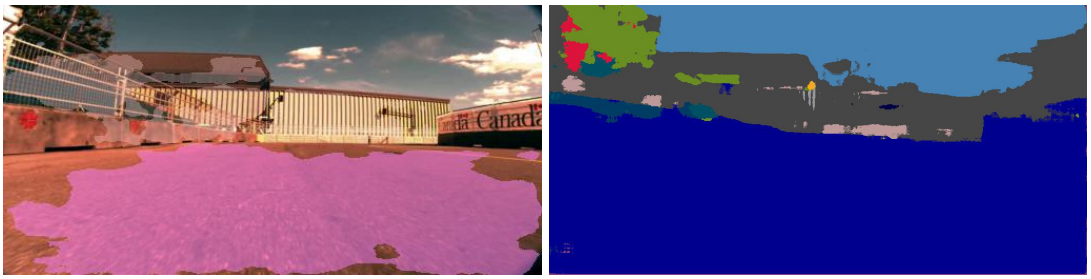
In order to have a baseline with which to compare the performance of the ERFNet, we implemented the FCN-8s architecture, also one of the best options out of the architectures we were considering, given its good trade-off between accuracy and complexity. We trained it on Cityscapes for 50 epochs on 3 classes, namely road, constructions, *i.e.*, building, wall, fence and guard rail, and background (subfigures 20a and 20c.

Even though only qualitatively, one could easily tell that the FCN-8s implementation outperforms the ERFNet model when presenting them with images they had never seen before, like Roborace images. In a sense, this was not unexpected, since the FCN-8s is known to be a fairly reliable architecture [58], in that it deals better with generalization than other networks.



(a) Example 1: using FCN-8s architecture trained on Cityscapes for 50 epochs

(b) Example 1: using ERFNet architecture trained on Cityscapes for 150 epochs



(c) Example 2: using FCN-8s architecture trained on Cityscapes for 50 epochs

(d) Example 2: using ERFNet architecture trained on Cityscapes for 150 epochs

Figure 20 Qualitative comparison between an FCN-8s architecture [18] trained on Cityscapes for 50 epochs and pre-computed ERFNet [23] model trained on Cityscapes for 150 epochs. The class road is represented by purple. Blue represents the class car.

Therefore, taking into account that the FCN-8s is one of the fastest architectures we were considering, *i.e.*, on average 0.2 seconds on the Cityscapes test set, (Table 4), that its class-IoU performance on the Cityscapes test set was fairly high (65.3%) and that it is a flexible architecture, easy to tweak and modify, we finally decided on it.

4.1.2. Implementation of an FCN-8s architecture. Hyperparameters.

As mentioned in the previous section, we opted to use the FCN-8s architecture for the semantic segmentation task. We implemented the network in Python, using the TensorFlow [59] framework. As encoder (refer to Section 2.3), we use a VGG-16 [20] pre-trained model, reusing layers 3, 4 and 7 of the VGG model as skip connections. Of course, we previously obtain the 1-by-1 convolutions of the layers mentioned.

The size to which input images are resized before entering the network was set to 512 x 256, and the set of hyperparameters that we found to work best are the ones displayed in Table 5. With respect to the kernel initializer, we finally decided on using a truncated normal initializer with a standard deviation of 0.01, since it produced far better results than using the Xavier initialization [60].

We will not go into greater detail as to how we obtained the hyperparameters shown in Table 5. On the contrary, we will thoroughly explain how we put together our own dataset of labelled Roborace [6] images for semantic segmentation (Section 4.2).

Table 5 Hyperparameters used for training.

Learning rate	Batch size	Dropout	Kernel initializer	Kernel regularizer
1e-5	1	0.5	Truncated normal initializer (standard deviation of 1e-2)	1e-3

4.2. Creation of our own Datasets for Semantic Segmentation - roborace 425 and roborace750

In semantic segmentation, creating a new dataset from scratch is not common. However, after training our FCN-8s implementation on the fine annotated Cityscapes dataset [14] for 50 Epochs and making inference on Roborace images (Figure 21), we noticed there was huge margin for improvement. For this reason, we decided to build our own labelled dataset of Roborace images.

(a) Example 1: using FCN-8s architecture trained on Cityscapes for 50 epochs

(b) Example 2: using FCN-8s architecture trained on Cityscapes for 50 epochs

Figure 21 Two Roborace images on which we applied the FCN-8s model we trained on the fine annotated Cityscapes dataset for 50 epochs.

In this section we explain the process we followed to build our own labelled dataset, which we will refer to as "roborace750", since it is comprised of 750 labelled images from 3 different Roborace race tracks. A second and smaller dataset, "roborace425", was also put together in this work in order to compare results between datasets and verify that the larger the dataset is, the better the results are.

4.2.1. Selection of Images and Annotation Process

As of this writing, the available number of Roborace race tracks was four, namely Berlin, Montreal, New York and Hong Kong. For each city we had frames both from the left and the right cameras of the Devbot, the prototype of the Robocar (Figure 2). For the task of semantic segmentation, we opted for working with the left images from the first three cities mentioned above.

After getting rid of the frames in which the vehicle had not yet begun moving, we ended up with the distribution of images shown in Table 6. For each city, we then randomly selected 250 images, which we labelled manually. The annotation tool we used for labelling our images was that developed by the Cityscapes project [61], with certain updates we had to carry out on the tool's source code to make it executable with Python 3.

Table 6 Available left frames from Berlin, Montreal and New York race tracks.

Berlin	Montreal	New York	Total
3633	3751	2381	9765

Given that our goal was to locate the car with respect to the sides of the road, *i.e.*, the fences, we decided to annotate 3 different classes on our pictures, namely road, fence and background. Figure 22 shows an example of the original and the annotated image.

(a) Example 1: Original image (b) Example 1: Annotated image

Figure 22 Annotation of the Roborace images: (left) original image; (right) annotated image.

The process of annotating an image is essentially drawing polygons that surround the desired target classes. In order to automate the process as much as possible, we always started drawing a coarse polygon containing the road, to then fine annotate the fence, paying special attention to the road-fence border (Figure 23). Note as well how we avoided labelling the lower-right corner as road, since this little triangle, as we can observe in some frames with enough brightness, is most probably part of the camera lens, or maybe part of the vehicle's chasis.

(a) First, coarse annotation of the road (b) Second, fine annotation of the fence, overlapping the part of the road polygon that does not represent the actual road.

Figure 23 Annotation process: (left) coarse annotation of the road; (right) fine annotation of the fence.

4.2.2. The datasets: roborace425 and roborace750

As we already mentioned, we have built two datasets, namely roborace425 and roborace750. Both datasets follow the structure explained in Section 2.2.2, *i.e.*, train set, validation set and test set. Recall that the train set is that on which the network should actually train. The validation set is used to detect a very common problem in ML: overfitting. The network does not actually use the images of the validation set to update its weights. Finally, the test set is the one on which we make inference, *i.e.*, test how good the model we have trained performs on images it has never seen before.

In our case, both datasets' test sets are comprised of exactly the same images, since we wanted to compare the performance of a model when trained on each dataset. Table 8 shows how we have structured the datasets.

Table 7 Distribution of images in roborace425 and roborace750.

	roborace425	**roborace750**
train set	**250** [Montreal]	**500** [250 [Montreal] + 250 [New York]]
val set	**50** [New York]	**125** [Berlin]
test set	**125** [Berlin]	**125** [Berlin]

Note that roborace750 follows approximately the common distribution used in DL, *i.e.*, train set roughly 70%, validation set 15% and test set 15% of all 750 available images. On the contrary, roborace425 does not follow this structure, the reason being that roborace425 was only needed for comparison purposes, not as a fully standarized dataset. Note as well that roborace750's validation and test sets, though comprised of the same numer of images from the Berlin race track, *i.e.*, 125 images, contain different frames featuring different scenes of the Berlin race track.

The subjects covered thus far (sections 4.1 and 4.2) include all aspects related to one of the two entry points of the SemanticDepth pipeline, *i.e.*, semantic segmentation. In the flow diagram featured in Figure 19, it corresponds to the top-left box. Next, we will deal with depth estimation, the other entry point to the pipeline (bottom-left box in the flow diagram).

4.3. Depth Estimation Model - Monodepth

In Section 2.4.4 we already went over the monodepth network, developed by Godard [41]. In essence, the novelty of this monocular depth estimation architecture lies in that it poses monocular depth estimation as an image reconstruction problem, solving for the disparity map without requiring ground truth depth.

Godard already trained his architecture on different datasets, such as Cityscapes, both on the fine annotated and coarse annotated datasets. A total of 22973 training stereo images, which add

up to 110 GBs (Gigabytes). The precomputed Cityscapes model gives exceptional results when presented with Cityscapes images, producing disparity maps which differ only slightly from the ground truth velodyne depth values provided by the Cityscapes dataset [41].

However, using this same Cityscapes-trained model on Roborace images produces less optimal results. For this reason, we decided to put together our own dataset of stereo-pair Roborace images. The final goal was to train Godard's architecture on it.

4.4. Creation of our own Dataset for Depth Estimation - RoboStereo

Firstly, we must bear in mind that a stereo dataset must be comprised of perfectly rectified and synchronized stereo pairs (Appendix C.2.5). With respect to rectification, this can be achieved offline as a post-processing step. The only requirement is having either all stereo rig parameters, *i.e.*, both cameras' intrinsic and extrinsic parameters plus translation and rotation matrices between cameras (Appendix C.2.4), or various image pairs featuring a chessboard of known dimensions. These chessboard images must of course have been taken with the same stereo rig as the rest of the image pairs on which we wish to train our depth estimation model.

This first requirement, *i.e.*, rectification of the stereo pairs, is not hard to meet. Once you know the geomtetric disposition of the cameras and their intrinsic and extrinsic parameters, rectifying a stereo pair is straight forward.

However, the second requirement, synchronization, is determined at the time of taking the frames, and completely depends on how the stereo rig was programmed to capture the stereo pairs. Too big of a time lag between left and right frames can be disastrous for the correctness of the dataset. In our case, from a total of 13511 image pairs available from the 4 different race tracks (Berlin, Montreal, New York and Hong Kong), only a few meet this synchronization requirement.

Next, we will discuss both synchronization and rectification in the case of the stereo pairs provided by the Roborace competition.

4.4.1. Synchronization of Roborace stereo image pairs

By studying the timestamp files provided by Roborace for each race track, we generated Table 8, which shows the number of stereo pairs whose time lag is equal or inferior to a certain threshold (0 ms, 5 ms, 10 ms, 15 ms and 20 ms). We did this for each race track individually and for the 4 race tracks in total. It is worth noting, for example, the row corresponding to 0 ms time lag. Only a total of 97 out of 13511 (0.71%) stereo pairs meet the requirement of having a time lag of 0 ms.

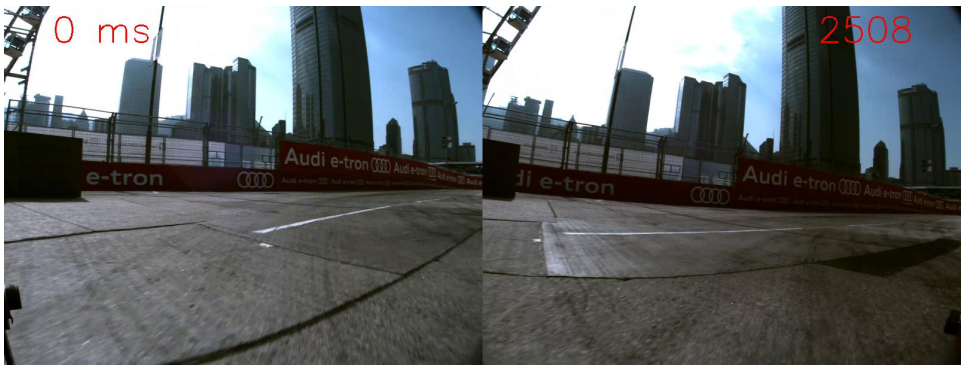
If we relax the synchronization condition to 5 ms, still only a 9.53% of all available image pairs meet the requirement. Of course, the higher the allowed time lag between left and right images, the higher the number of stereo pairs that meet the condition, but also the lesser the quality of the resulting dataset.

Table 8 Distribution of images in roborace425 and roborace750.

Time lag	**Berlin**	**Montreal**	**New York**	**Hong Kong**	Total
0 ms	28 / 3633	5 / 3751	47 / 2381	17 / 3746	**97** / 13511 (0.71 %)
5 ms	311 / 3633	44 / 3751	677 / 2381	256 / 3746	**1288** / 13511 (9.53 %)
10 ms	594 / 3633	269 / 3751	1263 / 2381	709 / 3746	**2835** / 13511 (20.98 %)
15 ms	669 / 3633	1615 / 3751	1416 / 2381	1737 / 3746	**5437** / 13511 (40.24 %)
20 ms	718 / 3633	2757 / 3751	1444 / 2381	2567 / 3746	**7486** / 13511 (55.4 %)

If we were to accept a time lag of 5 ms between left and right images, we would have a total of 1288 Roborace stereo pairs on which to train Godard's architecture [41], or any other depth estimation network. Compared to the 22973 Cityscapes stereo image pairs on which Godard trained his network, is not much, but there is always the possibility to fine tune on top of the pre-computed Cityscapes monodepth model.

However, checking the resulting 5-ms-time-lag stereo dataset, we noticed that some of the pairs, which supposedly had a time lag smaller or equal than 5 ms, visually did not match (Figure 24).



(a) Hong Kong - Example 1. Supposedly, time lag of 0 ms between left and right frames.

(b) Hong Kong - Example 2. Supposedly, time lag of 0 ms between left and right frames.

Figure 24 Examples of poorly synchronized Roborace stereo image pairs.

Since the time resolution of the timestamps provided by Roborace is 1 ms, this unsynchronization problem could have two possible sources. On the one hand, even though the timestamp file reflects a time lag of 0 ms for a given stereo pair, in reality this value could be smaller, *e.g.*, 0.9 ms. At a speed of 225 km/h (the common maximum speed for a Formula E vehicle), such a time lag could mean a distance lag of 5.6 cm between left and right shots, by no means negligible in stereo vision. Another possible explanation could simply be that the timestamp files provided by the Roborace organization has been written wrongly.

4.4.2. Rectification of Roborace stereo image pairs

The rectification issue is less substantial than that of synchronization, the reason being that given the stereo rig parameters, one can easily rectify the image pairs. However, as of this writing, we could not obtain neither the stereo rig parameters nor a set of stereo images featuring a chessboard to compute the parameters ourselves. Therefore, rectification of the stereo image pairs of our Roborace stereo dataset has not been addressed. Note, of course, that before rectification we should have had to undistort the images, given the high level of radial distortion (Appendix C.1.1) present in the images.

Given the impossibilty to work on Roborace data in the depth estimation task, the explanation of the SemanticDepth pipeline (Section 4.5) features an image from the Cityscapes dataset.

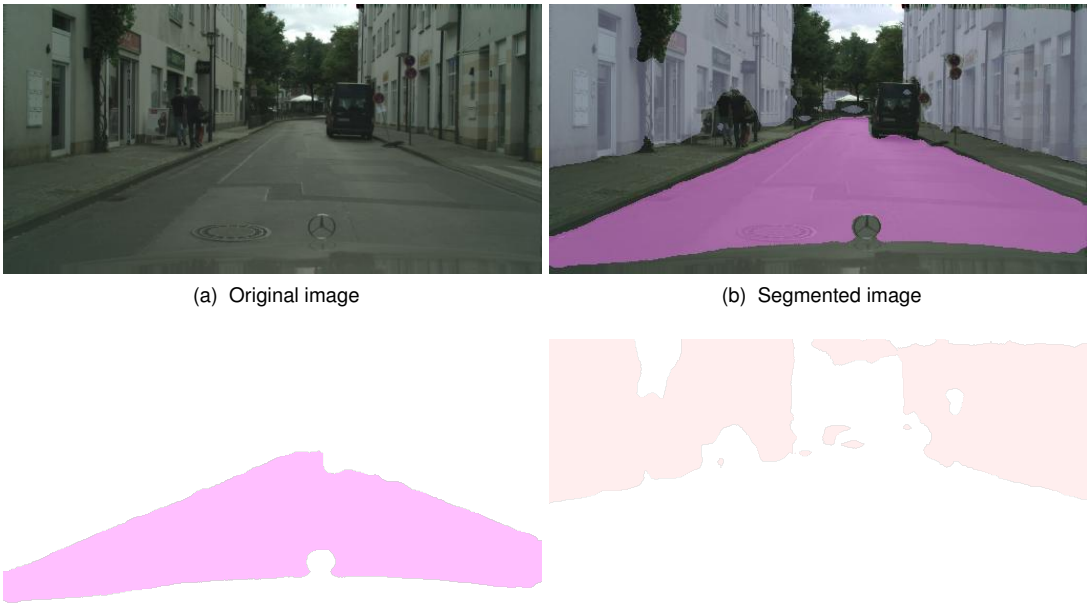
4.5. SemanticDepth pipeline

As mentioned already in Chapter 3, in order to compute the distance from the vehicle to the sides of the road, *i.e.*, the fences, in the Roborace race tracks, we have developed a pipeline which integrates a semantic segmentation architecture (Section 4.1) with a depth estimation DNN (Section 4.3). In Figure 19 we already presented the flow diagram of the pipeline, but for the convenience of the reader we will enumerate its fundamental steps again.

1. Read input RGB image and resize it to 512 x 256.
2. Use a semantic segmentation model (FCN-8s in our case) to make inference on frame and generate a road mask and a fence mask.
3. Produce disparity map using a monocular depth estimation network (monodepth [41]).
4. Generate 3D Point Cloud from the disparity map using stereo vision theory (Section **??**).
5. Apply previously computed masks to the 3D Point Cloud and obtain a road3D Point Cloud and a fence3D Point Cloud. Separate the fence3D Point Cloud into left fence and right fence.
6. Remove noise from 3D Point Clouds.
7. Compute road's width (naive approach) and distance between fences (advanced approach) at a certain depth.

4.5.1. Obtaining road and fence 2D masks by using a semantic segmentation model

After resizing the input image to 512 x 256, we apply our semantic segmentation model on it, in order to classify all its pixels into 3 different classes, namely road, fence and background. This step outputs both a road and a fence 2D mask, which will be used later on (Figure 25). Note that the segmented image will be simply used for representation purposes of the final result, whilst only the road and fence masks are relevant for the next steps of the pipeline.



(a) Original image (b) Segmented image

(c) Road mask (d) Fence mask

Figure 25 Semantic segmentation step in SemanticDepth pipeline.

4.5.2. Obtaining the disparity map by using a monocular depth estimation model

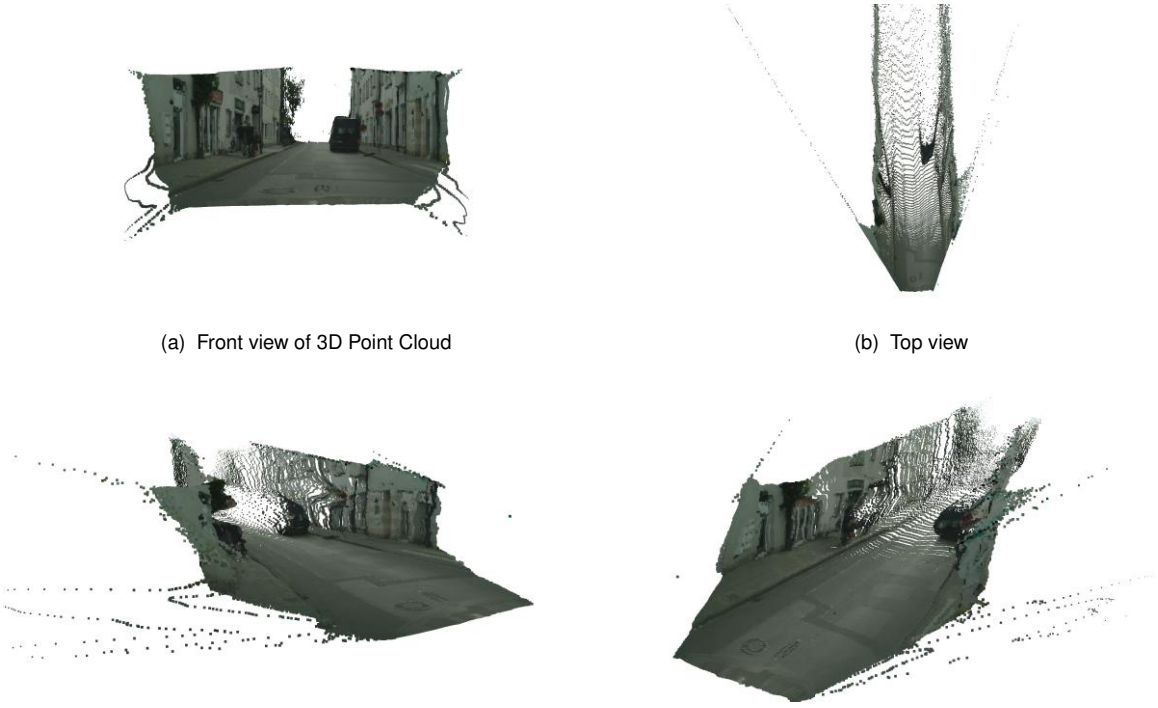
Next, we compute the disparity map of the resized input image by using a monodepth [41] model. Figure 26 shows an example of a disparity map obtained using a pre-computed monodepth model trained on Cityscapes by Godard [41]. Had we been able to train the monodepth architecture on the Roborace stereo dataset, we would have used that model to make inference on Roborace images, which indeed was the initial goal.

(a) Original image (b) Disparity map

Figure 26 Disparity map (right) obtained from a single image (left) using monodepth [41].

4.5.3. Generation of a 3D Point Cloud from the disparity map

In this step we convert the disparity map obtained in the previous step into a 3D Point Cloud. In Section 2.5 we presented the procedure to accomplish this taks, *i.e.*, first scaling the disparity outputted by monodepth with the full width resolution of the input image and secondly using computer vision theory (Equation 2.13) to convert the corrected disparity map into 3D points. Figure 27 shows different views of a 3D Point Cloud generated from the disparity map shown in Figure 26.



(a) Front view of 3D Point Cloud

(b) Top view

(c) Left view

(d) Right view

Figure 27 Different views of a 3D Point Cloud obtained from a disparity map. The disparity map has been produced using a pre-computed monodepth [41] model trained on Cityscapes.

Recall from Section 2.5 that in its current form monodepth should only be applied to images with a focal length and aspect ratio identical to the dataset it was trained on. Moreover, Equation 2.13 showed that, in order to convert disparities into depth values, we need the baseline of the stereo camera that took the training images on which the monodepth model was trained on.

Therefore, in order to apply the monodepth architecture onto images taken with a single camera (instead of a stereo camera) and with different focal lengths and aspect ratios than those from the stereo dataset the monodepth model was trained on, we had to slightly modify Equation 2.13 and design a simple procedure, which we describe next.

First, we scale the diparity map outputted by monodepth with the original width of the input image, regardless of whether the resolution of the input image differs from that of the Cityscapes [14] images (Equation 2.12). Secondly, we set the baseline in Equation 2.13 to 1 m. Finally, we use the intrinsic parameters, *i.e.*, focal length and principal point, of the camera with which the image we want to process was taken and use those in Equation 2.13.

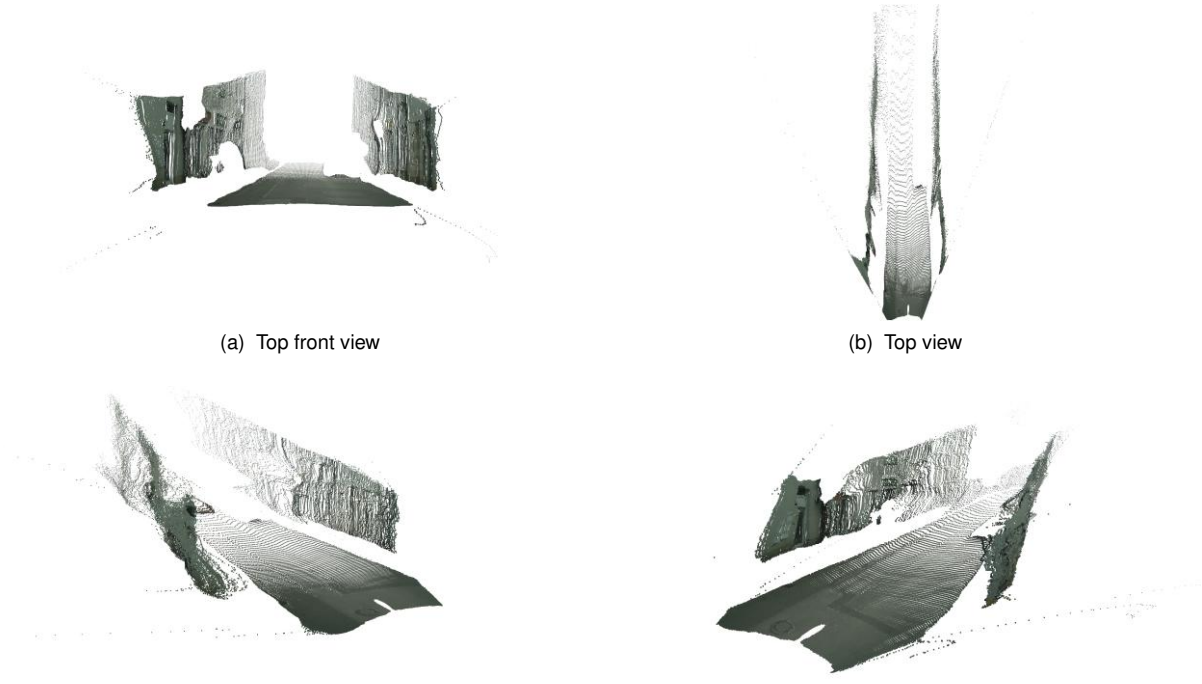
Equation 4.1 shows the updated version of Equation 2.13. Note that we also decided to apply a rotation of 180 º around the x axis to matrix Q, so that the y axis would look up, the x axis to the right and the z axis into the screen.

$$Q \begin{bmatrix} u \\ v \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix}, \quad where \quad Q = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & -1 & 0 & c_y \\ 0 & 0 & 0 & -f \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{4.1}$$

4.5.4. Overlaying the 2D road and fence masks onto the 3D Point Cloud of the scene

The next step is to apply the road and fence 2D masks, which we previously obtained in the semantic segmentation step (Section 4.5.1), onto the 3D Point Cloud. By doing so, we obtain a 3D Point Cloud featuring only the road and the fences, or simply any kind of building wall in the case of Cityscapes. Figure 28 shows the resulting 3D Point.

The level of noise in both 3D Point Clouds, *i.e.*, road and fences, is fairly high. Given that our final goal is computing relevant distances within the 3D Point Cloud, this poses a big hurdle. Therefore, the next step must consist in removing as much noise as possible from them.



(a) Top front view

(b) Top view

(c) Top left view

(d) Top right view

Figure 28 3D Point Cloud featuring only points belonging to classes road and fence.

4.5.5. Denoising of the 3D Point Clouds of the road and fences. Fitting planes to them.

For the denoising task we have created a small Point Cloud library. Amongst some other tools, we have written a function which first computes the Median Absolute Deviation of a given 3D Point Cloud along one of the three dimensions (set by the user as a parameter), secondly computes a penalty for each point of the given Point Cloud and finally removes all points whose penalty value is higher than a given threshold.

In SemanticDepth, we first apply the function described above to the 3D Point Cloud of the road, both in the x and y axes. By doing so, we manage to remove most of the existing noise in it. In the z axis, we apply a function to simply remove all points that are too close to the camera, since in the first meters there is always a certain level of distorsion in the Point Cloud of the road.

At this point, the road's Point Cloud is fairly free of noise. However, eventually we will want to compute the furthest 3D point to the left and to the right of the Point Cloud at a certain depth. Therefore, it is necessary to remove all possible outliers. To do so, we first try to fit a plane to the road and then remove all points whose distance to the computed plane is greater than a given threshold.

The way we approach plane fitting is by computing the least-squares solution to the plane equation $Ac = b$, where matrix A, of shape (<numberOfPointsInPointCloud>, 3), is of the form $[[x][y][1]]$. Column $[1]$ denotes a one dimensional vector filled with 'ones' and length equal to the number of points in the road Point Cloud. Vector b, of shape (<numberOfPointsInRoadPointCloud>, 1) is equal to $[z]$. By solving for c we obtain the coefficients of the plane which best fits our set of road points.

The process described in the preceding paragraph is designed to fit a plane to a 3D Point Cloud whose two Principal Components are x and y. In other words, it would optimize the fitting of a plane perpendicular to the z axis. Fitting planes which are penpendicular to the x and y axis follows the same procedure described above, with the only difference that vector b would contain the x or y components, respectively, of the points to which we want to fit a plane, and the first two columns of matrix A would contain the other two components of the points, (y, z) or (x, z), respectively.

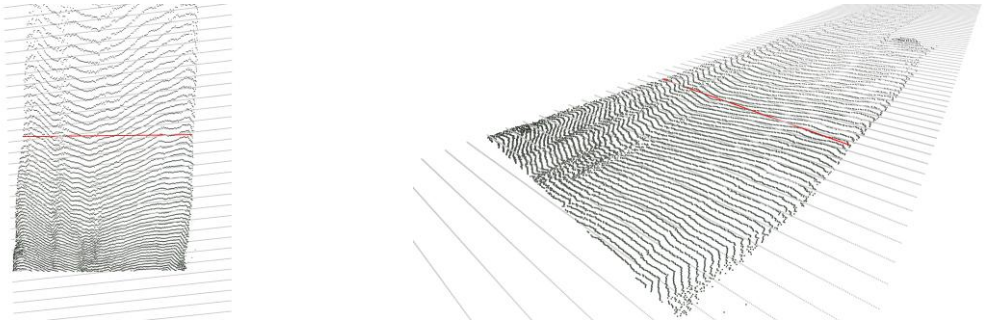
With regards to the fences, the denoising process is similar. Before that, however, we need to separate the original Point Cloud containing both fences into two different Point Clouds, containing of course the left and right fences. To accomplish this task, we compute the mean of all the points belonging to the original fence Point Cloud, which will usually fall somewhere around the center of the road, *i.e.*, x coordinate close to 0, and then save all points to the left of the mean value into a "left fence" Point Cloud and viceversa.

What remains now is extracting any kind of valuable information from the 3D Point Clouds we have obtained. One possible measure that may help increase the vehicle's situational awareness is the width of the road. We have approached this task in two different ways. A naive approach is explained in Section 4.5.6, and a more advanced approach in Section 4.5.7.

4.5.6. Computing the Road's Width. Naive Approach

In this first approach, we only use the Point Cloud corresponding to the road. In order to compute the road's width, we will try to find the distance between the furthest left and furthest right points of the road at a given depth.

Essentially, this consists in iterating over all points situated at a certain depth value (z axis), looking then for the two points with the highest positive and negative x values (the x dimension goes along the width of the road) and finally computing the euclidean 3D distance between these two points (Figure 29).



(a) Top front view (b) Top view

Figure 29 Computing the width of the road using the navie approach.

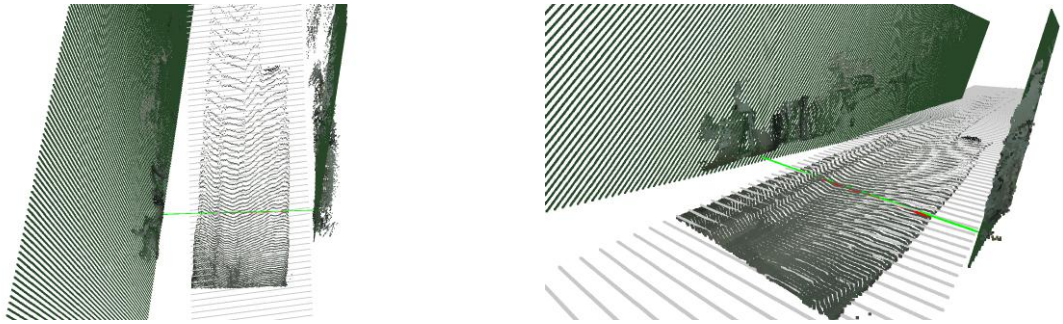
Note that repeating this same procedure for several distances may also be interesting, though it simply depends on what is needed from the visual perception algorithm.

This naive approach could suffer from poor road segmentations. In this case, we need a more robust approach, where we do not only rely on the segmentation of the road but also use the information given by the fences.

4.5.7. Computing the Road's Width. Advanced approach

A more robust approach we have designed consists not only in fitting a plane to the road's Point Cloud, but also to that of the left and right fences. Then, we can easily compute the intersection between the road's plane and the two fences' planes, thus obtaining the equations for both intersection lines. Finally, we can compute the distance between these two lines at a given depth (z) value (Figure 30).

Let us look at this process in greater detail. Once we have two fence objects (left and right), we can use the same procedure we used before for the road in order to fit planes to the fences. At this point, the two lines that delimit the path are completely defined by a system of two equations each, *i.e.*, left line: road's plane equation intersected with left fence's plane equation; right line: road's plane equation intersected with right fence's plane equation.



(a) Top view (b) Right view

Figure 30 Computing the distance between the fences using the advanced approach. The light green line represents the distance between fences. The red line

Now we can just set a depth (z) value, substitute it into both systems of equations (left line and right line) and solve both determinate systems (where the unknowns are x and y). Therefore, we obtain for each system, *i.e.*, each line, the x and y values of a point situated along each line at depth z. All that remains now is simply computing the euclidean distance between these two points, obtaining the distane between left and right fences at depth z.

Note that by looking for the intersection of the road's plane with the corresponding fence's plane we ensure that computing the road's width does not require a 100%-accurate segmentation of the road, only one good enough so that a plane can be fitted to the road.

5. Results

In this chapter we will start by going over the results concerning the various semantic segmentation models we have trained in this work. In Section 5.1) we will present their performance on the Roborace datasets we have created (roborace425 and roborace750) and on images from the Cityscapes dataset.

Then, in Section 5.2 we will display the results obtained when presenting a depth estimation model, *i.e.*, monodepth [41], with images from the Cityscapes dataset and from the Roborace datasets.

Finally, in Section 5.3 we will show the results achieved by our SemanticDepth pipeline on a few images featuring streets of Munich. We will present what are the predicted distances for these scenes (both the naive and the advanced distances) at a depth of 10 m, and we will compare these predictions with the real measurements. Furthermore, we will present the inference time of our pipeline on these images.

5.1. Results on the Semantic Segmentation Task

In this section we present the results obtained in the task of semantic segmentation. As we saw in Section 4.1.2, we implemented an FCN-8s architecture, which we have trained on our own labelled datasets (roborace425 and roborace750). For each dataset, we trained the architecture for 100 and 200 epochs, using the set of hyperparameters we showed in Table 5 (Section 4.1.2). Therefore, in this section we will be referring to the models "roborace425_100Epochs", "roborace425_-200Epochs", "roborace750_100Epochs" and "roborace750_200Epochs".

In Sections 5.1.1 and 5.1.2 we present the 'IoU vs Epochs' and 'Loss vs Epochs' results on both the training and validation sets of the roborace425 and roborace750, respectively. In Section 5.1.3 we compare the mean class-IoU achieved by each one of the trained models on the test set of our Roborace datasets .

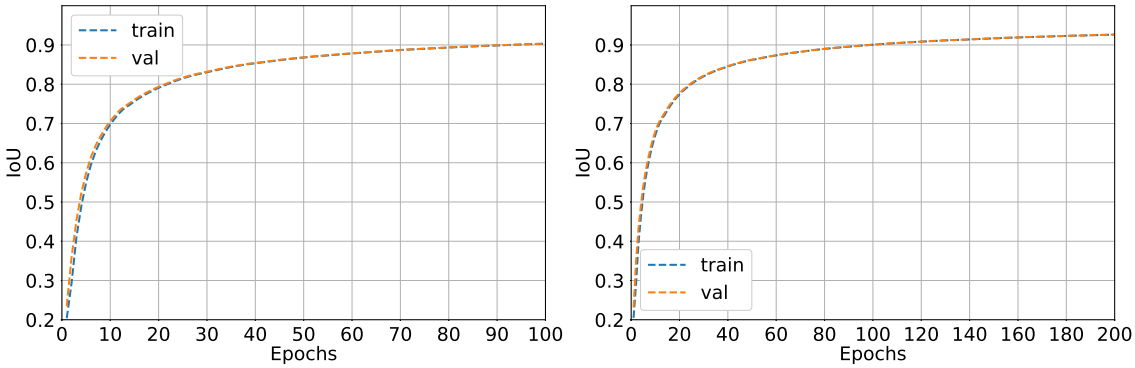
Subsequently, in Section 5.1.4, we present the metrics obtained on the training and validation sets of the Cityscapes dataset when training our FCN-8s implementation on this dataset both for 50 and 100 epochs. We will refer to these models as "cityscapes_50Epochs" and "cityscapes_-100Epochs", respectively. We will also show how these models perform on the test set of the Roborace datasets.

Finally, in Section 5.1.5 we compare the performance of the best model obtained by training on Roborace images and the best model obtained by training on Cityscapes, in an attempt to justify the creation of our own labelled dataset.

5.1.1. Results when training our FCN-8s implementation on the roborace425 dataset

5.1.1.1 IoU metric

Figure 31 shows the 'IoU vs Epochs' results on the training set (blue curves) and validation set (orange curves) when training our FCN-8s implementation both for 100 epochs and 200 epochs. The final values achieved in each case are shown on Table 9.



(a) 100 epochs on roborace425 (b) 200 epochs on roborace425

Figure 31 IoU vs Epochs results on the train set (blue curves) and validation set (orange curves) of roborace425 when training our FCN-8s implementation for 100 and 200 epochs.

Note how within each graph the IoU values are very similar on the training and validation sets, almost identical. With regards to the differences between training for 100 and for 200 epochs, *i.e.*, differences between left and right graphs, we can say that the IoU on both the training and validation sets slightly improves when training for a higher number of epochs, as it can also be seen on Table 9).

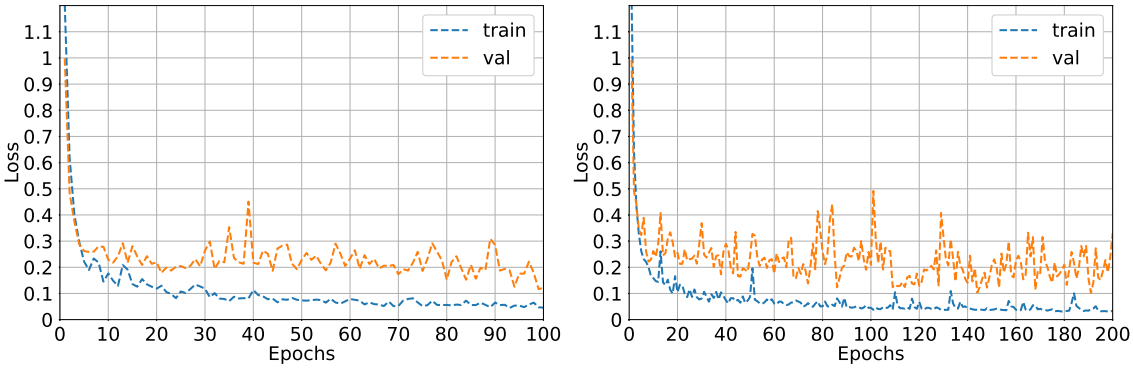
Table 9 IoU final values for training and validation sets using an FCN-8s trained on roborace425.

FCN-8s on roborace425	100 Epochs	200 Epochs
IoU (%) on training set	92.66%	93.18%
IoU (%) on validation set	92.89%	93.10%

5.1.1.2 Loss metric

Figure 32 shows the 'Loss vs Epochs' results on the training set (blue curves) and validation set (orange curves) when training our FCN-8s implementation both for 100 epochs and 200 epochs. The final values achieved in each case are shown on Table 10.

In Figure we can observe 32 how the error on the training set (blue curves) always decreases, and it would keep doing so the longer the number of epochs we train for. This can be confirmed checking the final values of the loss on the training set on Table 10: from 0.0504 when training for 100 epochs to 0.0362 when training for 200 epochs.



(a) 100 epochs on roborace425

(b) 100 epochs on roborace425

Figure 32 Loss vs Epochs results on the train set (blue curves) and validation set (orange curves) of roborace425 when training our FCN-8s implementation for 100 and 200 epochs.

With regards to the error on the validation set (orange curves in Figure 32), though a very variable curve, there is a slight upward trend from around epoch 25 onwards in both curves (100- and 200-epochs graphs).

Table 10 Loss final values for training and validation sets using an FCN-8s trained on roborace425.

FCN-8s on roborace425	100 Epochs	200 Epochs
Loss on training set	0.0504	0.0362
Loss on validation set	0.2224	0.3132

5.1.2. Results when training our FCN-8s implementation on the roborace750 dataset

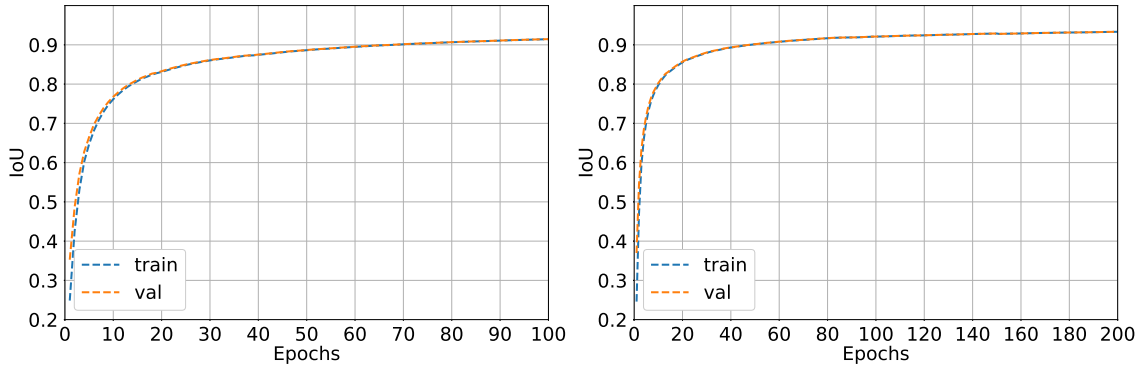
5.1.2.1 IoU metric

Figure 33 shows the 'IoU vs Epochs' results on the training set (blue curves) and validation set (orange curves) when training our FCN-8s implementation both for 100 epochs and 200 epochs. The final values achieved in each case are shown on Table 11.

Again, note how within each graph the IoU values are very similar on the training and validation sets, almost identical. With regards to the differences between training for 100 and for 200 epochs, *i.e.*, differences between left and right graphs, we can say that the IoU on both the training and validation sets slightly improves when training for a higher number of epochs (Table 11).

Table 11 IoU final values for training and validation sets using an FCN-8s trained on roborace750.

FCN-8s on roborace750	100 Epochs	200 Epochs
IoU (%) on training set	91.44%	93.33%
IoU (%) on validation set	91.46%	93.32%

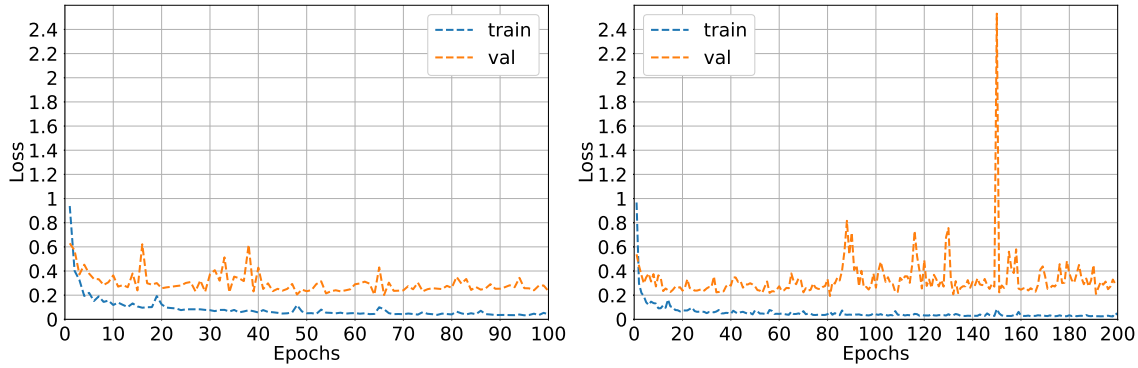


(a) 100 epochs on roborace750 (b) 200 epochs on roborace750

Figure 33 IoU vs Epochs results on the train set (blue curves) and validation set (orange curves) of roborace750 when training our FCN-8s implementation for 100 and 200 epochs.

5.1.2.2 Loss metric

Figure 34 shows the 'Loss vs Epochs' results on the training set (blue curves) and validation set (orange curves) when training our FCN-8s implementation both for 100 epochs and 200 epochs. The final values achieved in each case are shown on Table 12.



(a) 100 epochs on roborace750 (b) 200 epochs on roborace750

Figure 34 Loss vs Epochs results on the train set (blue curves) and validation set (orange curves) of roborace750 when training our FCN-8s implementation for 100 and 200 epochs.

We can observe in Figure 34 how the error on the training set (blue curves) always decreases, and it would keep doing so the longer the number of epochs we train for. This can be confirmed checking the final values of the loss on the training set on Table 12: from 0.0411 when training for 100 epochs to 0.0371 when training for 200 epochs.

With regards to the error on the validation set (orange curves in Figure 34), though a very variable curve, it shows a slight upward trend from around epoch 25 onwards in both curves (100 and 200 epochs), as it was also the case when training on roborace425 (Section 5.1.1).

Next, we will compare all four models we have presented thus far, namely roborace425_100Epochs, roborace425_200Epochs, roborace750_100Epochs and roborace750_200Epochs.

Table 12 Loss final values for training and validation sets using an FCN-8s trained on roborace750.

FCN-8s on roborace750	**100 Epochs**	**200 Epochs**
Loss on training set	0.0411	0.0371
Loss on validation set	0.2331	0.293

5.1.3. Comparison of the results achieved by each model on the Roborace test set

In order to best compare the four different Roborace models (roborace425_100Epochs, roborace425_200Epochs, roborace750_100Epochs and roborace750_200Epochs), we have applied them on the same test set and computed the mean class-IoU value for each case (Table 13). As we can observe in the table, the model that outperforms the rest is that trained on the roborace750 dataset for 100 epochs (roborace750_100Epochs).

Table 13 IoU on test set achieved by network when trained on roborace425 and roborace750 for 100 and 200 epochs.

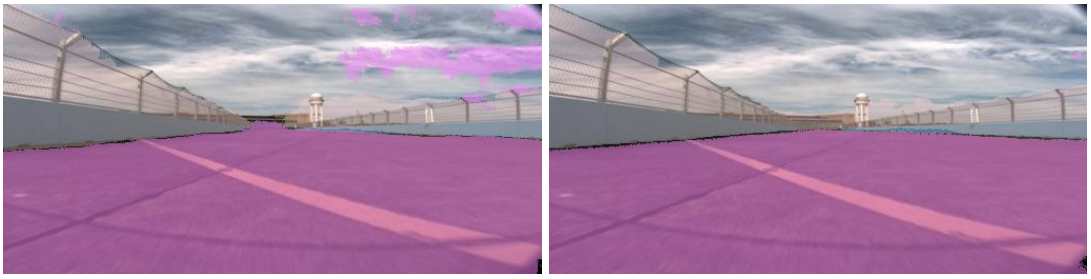
IoU (%) on test set	**FCN-8s on roborace425**	**FCN-8s on roborace750**
100 Epochs	84.07%	**87.63%**
200 Epochs	77.49%	83.86%

Figures 35 and 36 compare qualitatively the results achieved by each one of the four models on two examples from the Roborace test set. Note how the best segmentation results are always achieved by the model trained on roborace750 for 100 epochs (upper-right subimage).

(a) Model trained on roborace425 for 100 epochs (b) Model trained on roborace750 for 100 epochs

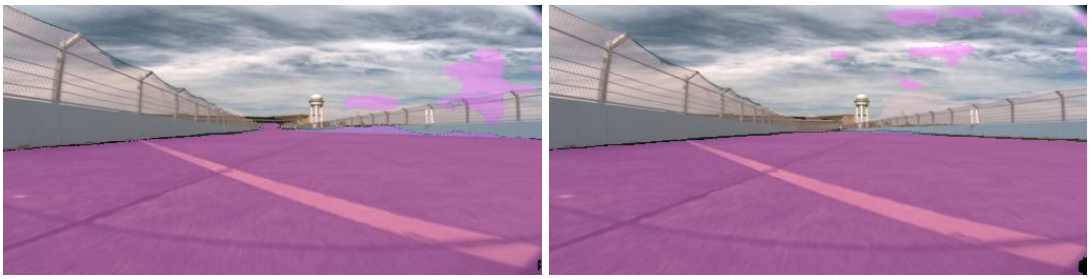
(c) Model trained on roborace425 for 200 epochs (d) Model trained on roborace750 for 200 epochs

Figure 35 Qualitative results on the roborace test set (image: berlin 00127).



(a) Model trained on roborace425 for 100 epochs

(b) Model trained on roborace750 for 100 epochs



(c) Model trained on roborace425 for 200 epochs

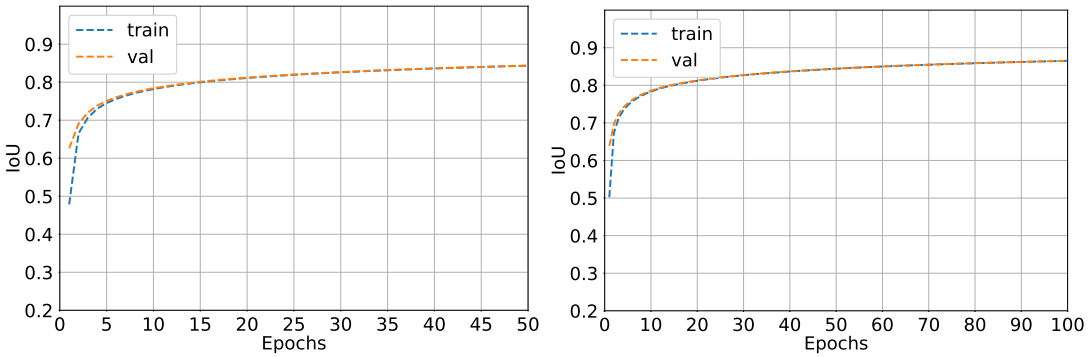
(d) Model trained on roborace750 for 200 epochs

Figure 36 Qualitative results on the roborace test set (image: berlin 00206).

5.1.4. Results when training our FCN-8s implementation on the Cityscapes dataset

5.1.4.1 IoU metric

Figure 37 shows the 'IoU vs Epochs' results on the training set (blue curves) and validation set (orange curves) when training our FCN-8s implementation both for 50 epochs and 100 epochs on the Cityscapes dataset. The final values achieved in each case are shown on Table 14.



(a) 50 epochs on Cityscapes

(b) 100 epochs on Cityscapes

Figure 37 IoU vs Epochs results on the train set (blue curves) and validation set (orange curves) of Cityscapes when training our FCN-8s implementation for 50 and 100 epochs.

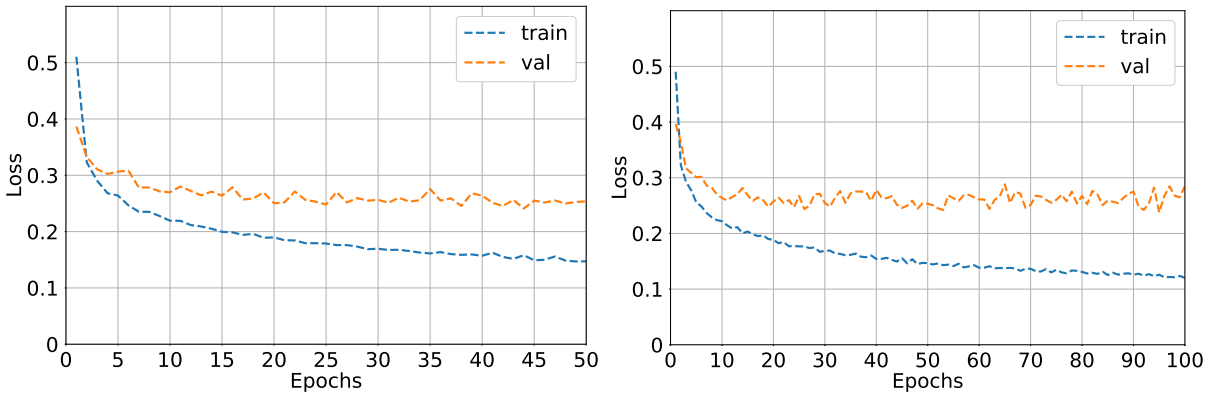
As in the previous cases, *i.e.*, when training on roborace425 (Section 5.1.1) and on roborace750 (Section 5.1.2), the IoU values are similar on the training and validation sets of each graph. With regards to the differences between training for 50 and for 100 epochs, we can once again state that the IoU on both the training and validation sets slightly improves the longer the model trains for. This can be confirmed looking at Table 14.

Table 14 IoU final values for training and validation sets using an FCN-8s trained on Cityscapes.

FCN-8s on Cityscapes	50 Epochs	100 Epochs
IoU (%) on training set	84.34%	86.51%
IoU (%) on validation set	84.37%	86.52%

5.1.4.2 Loss metric

Figure 38 shows the 'Loss vs Epochs' results on the training set (blue curves) and validation set (orange curves) when training our FCN-8s implementation both for 50 epochs and 100 epochs. The final values achieved in each case are shown on Table 15.



(a) 50 epochs on Cityscapes

(b) 100 epochs on Cityscapes

Figure 38 Loss vs Epochs results on the train set (blue curves) and validation set (orange curves) of Cityscapes when training our FCN-8s implementation for 50 and 100 epochs.

We can observe in Figure 38 how the error on the training set (blue curves) always decreases, and it would keep doing so the longer the number of epochs we train for. This can be confirmed checking the final values of the loss on the training set on Table 15: from 0.147 when training for 50 epochs to 0.12 when training for 100 epochs.

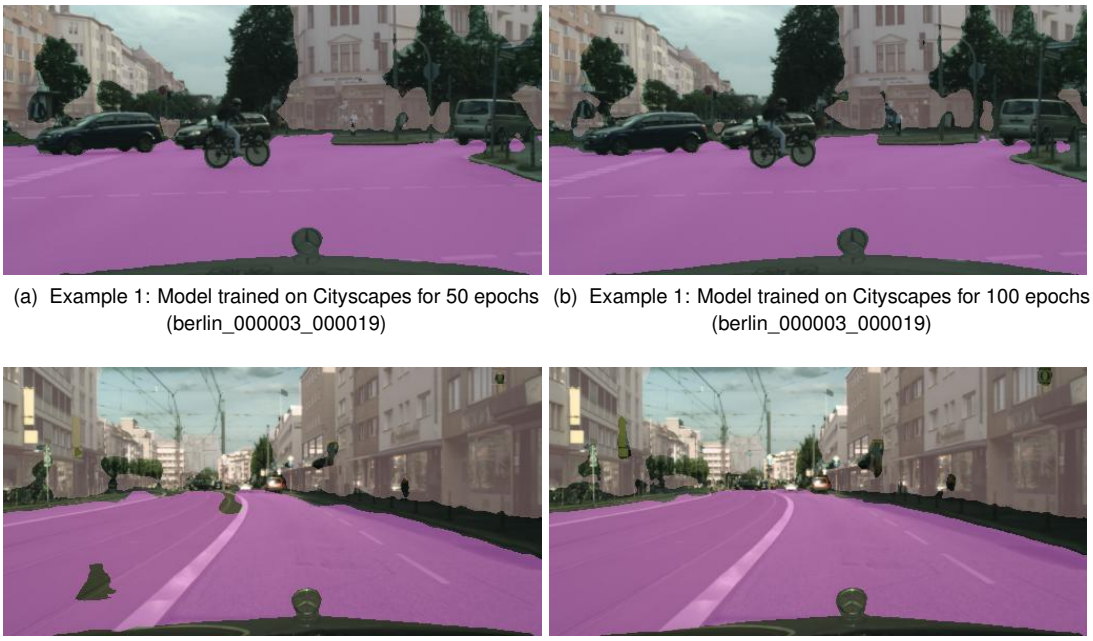
Regarding the error on the validation set (orange curves in Figure 38), it can be appreciated that the validation loss stops decreasing around epoch 25 onwards in both curves (50 and 100 epochs). Note as well how for the case of 100 epochs the validation error even starts increasing around epoch 50. This can be confirmed checking the final values of the loss on the validation set on Table 15: from 0.254 when training for 50 epochs to 0.285 when training for 100 epochs.

Table 15 Loss final values for training and validation sets using an FCN-8s trained on Cityscapes.

FCN-8s on Cityscapes	50 Epochs	100 Epochs
Loss on training set	0.147	0.12
Loss on validation set	0.254	0.285

5.1.4.3 Applying on the Cityscapes test set the models trained on Cityscapes

Given that the test set of the Cityscapes dataset does not contain ground truths, the only way to get a feeling about the models we have trained on this dataset is by comparing their metrics on the Cityscapes validation set and also by qualitatively observing some examples segmented with the models, as in Figure 39. As usual, we differentiate between the classes road (in purple), fence, *i.e.*, any kind of building, (in light brown) and background (no color).



(a) Example 1: Model trained on Cityscapes for 50 epochs (berlin_000003_000019)

(b) Example 1: Model trained on Cityscapes for 100 epochs (berlin_000003_000019)

(c) Example 2: Model trained on Cityscapes for 50 epochs (bonn_000009_000019)

(d) Example 2: Model trained on Cityscapes for 100 epochs (bonn_000009_000019)

Figure 39 Qualitative results on the Cityscapes test set using models trained on Cityscapes.

5.1.4.4 Applying on the Roborace test set the models trained on Cityscapes

We have also applied the two models we trained on Cityscapes on the Roborace test set. In Table 16 we present the class-IoU values achieved by each model (cityscapes_50Epochs and cityscapes_100Epochs) on the Roborace test set.

Table 16 IoU on test set achieved by network when trained on roborace425 and roborace750 for 100 and 200 epochs.

	Cityscapes-50Epochs model	**Cityscapes-100Epochs model**
IoU (%) on Roborace test set	52.855%	55.233%

Qualitatively, Figure 40 shows two example images from the Roborace test set on which we have applied segmentation using these two Cityscapes-trained models. By taking into account the validation loss of the two models, their class-IoU values on the Roborace test set and the qualitative results shown in Figure 40, we can conclude that the model cityscapes_100Epochs slightly outperforms the model cityscapes_50Epochs.

(a) Example 1: Model trained on Cityscapes for 50 epochs (b) Example 1: Model trained on Cityscapes for 100 epochs

(c) Example 2: Model trained on Cityscapes for 50 epochs (d) Example 2: Model trained on Cityscapes for 100 epochs

Figure 40 Qualitative results on the Roborace test set using models trained on Cityscapes.

In Section 5.1.5 we compare the performance of the best model we obtained by training on Roborace images with respect to the best model obtained by training on the Cityscapes dataset, *i.e.*, we will compare the roborace750_100Epochs model against the cityscapes_100Epochs model.

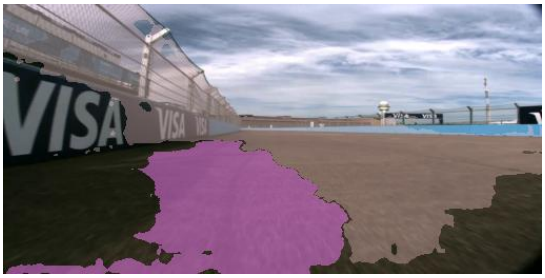
5.1.5. Comparison between the results when training on Roborace and Cityscapes

In order to justify the creation of our own semantic segmentation dataset, in this section we present a comparison between arguably the best Cityscapes-trained model (cityscapes_100Epochs) and our best Roborace-trained model (roborace750_100Epochs). Figure 41 undoubtedly shows how the model trained on our roborace750 dataset for 100 epochs outperforms the one trained on the Cityscapes dataset for 100 epochs, thus validating the creation of a Roborace dataset.

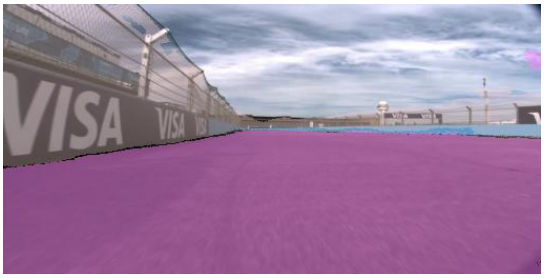
For the sake of completeness, Table 17 presents the mean IoU values obtained by the aforementioned models (cityscapes_100Epochs and roborace750_100Epochs) on the test set of our Roborace dataset. The improvement in accuracy is fairly clear when training on our roborace750 dataset.

Table 17 IoU on test set achieved by network when trained on roborace425 and roborace750 for 100 and 200 epochs.

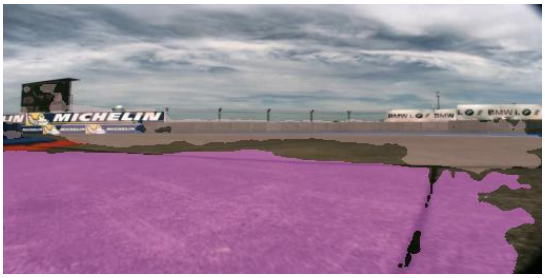
	Cityscapes-100Epochs model	**Roborace750-100Epochs model**
IoU (%) on Roborace test set	55.223 %	**87.63%**



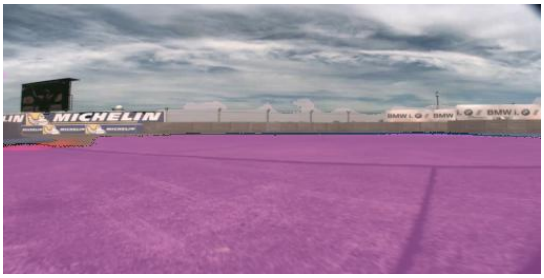
(a) Example 1: Model trained on Cityscapes for 100 epochs



(b) Example 1: Model trained on roborace750 for 100 epochs



(c) Example 2: Model trained on Cityscapes for 100 epochs



(d) Example 2: Model trained on roborace750 for 100 epochs

(e) Example 3: Model trained on Cityscapes for 100 epochs

(f) Example 3: Model trained on roborace750 for 100 epochs

(g) Example 3: Model trained on Cityscapes for 100 epochs

(h) Example 3: Model trained on roborace750 for 100 epochs

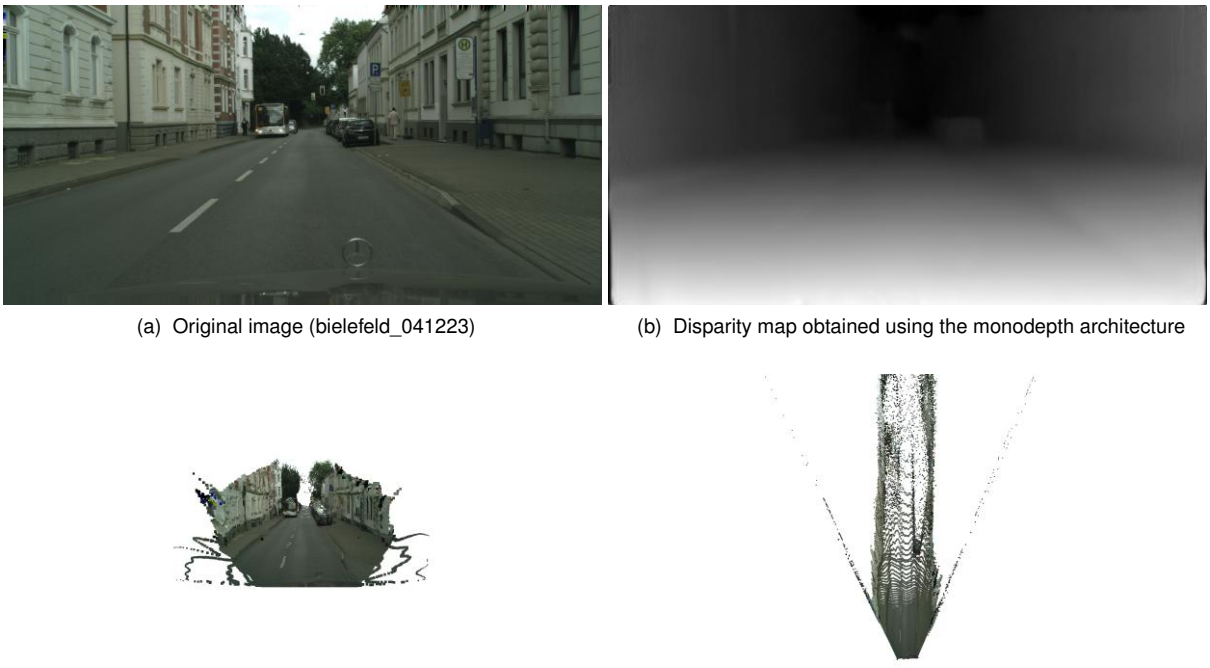
Figure 41 Qualitative comparison between the cityscapes_100Epochs model and the roborace750_100Epochs model on four images from the Roborace test set.

5.2. Results on the Depth Estimation Task

In this section we present qualitative results that a pre-computed monodepth [41] model trained on Cityscapes achieves on images from both Cityscapes and Roborace datasets. Recall that the monodepth architecture outputs a disparity map (Appendix C.2) from a single input image (Section 2.4.4). We will show this output, as well as a couple of views of the depth map (or 3D Point Cloud) we post-compute by reprojecting the disparity map back to 3D, as we already explained in Section 4.5.3.

5.2.1. Inference on Cityscapes

Figure 42 shows the original image, the disparity map and two views, *i.e.*, front view and top view, of the 3D Point Cloud of an example image from the Cityscapes dataset, more precisely from the city of Bielefeld.



(a) Original image (bielefeld_041223)

(b) Disparity map obtained using the monodepth architecture

(c) Front view of the 3D Point Cloud

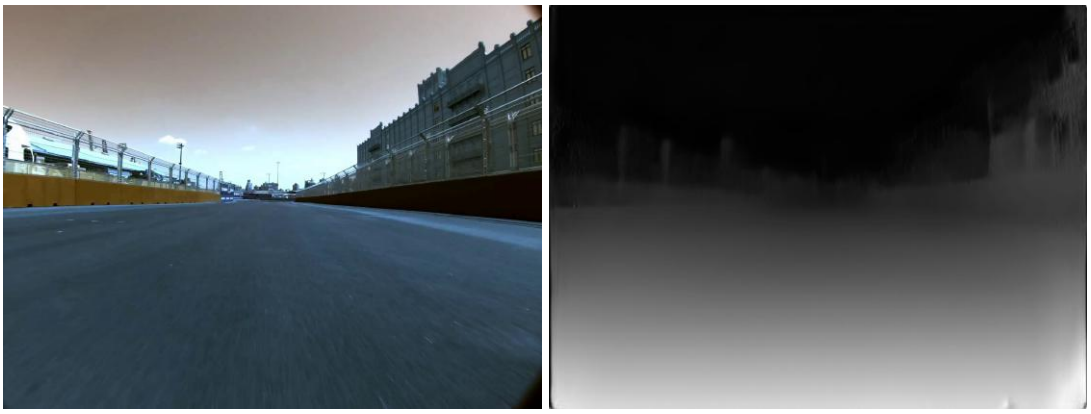
(d) Top view of the 3D Point Cloud

Figure 42 Disparity map (obtained using a pre-computed monodepth model) and depth map of an example image from the Cityscapes test set (image: bielefeld_041223).

In the case of Cityscapes, the baseline between each of the cameras of the stero rig, as well as the intrinsics of each camera, are provided for every city which constitutes the dataset. Therefore, Equation 2.13 could be used, thus following the indications provided by the author of monodepth [41] with regards to converting the disparities that the monodepth network outputs back to 3D points.

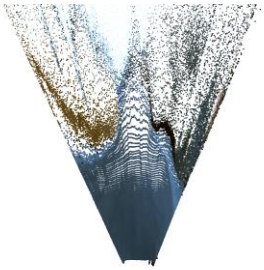
5.2.2. Inference on Roborace

Figure 43 shows the original image, the disparity map and two views of the 3D Point Cloud of a Roborace scene on which we have applied the pre-computed Cityscapes-trained monodepth model.



(a) Original image (newyork_robo_00080)

(b) Disparity map obtained using the monodepth architecture



(c) Front view of the 3D Point Cloud

(d) Top view of the 3D Point Cloud

Figure 43 Disparity map (obtained using a pre-computed monodepth model) and depth map of an example image from the roborace750 dataset (image: newyork_00080).

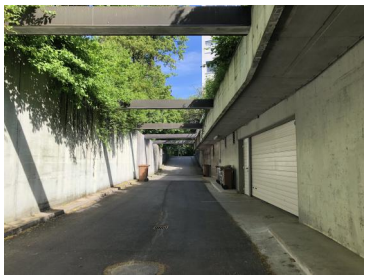
In this case, given that the images from Roborace do not share neither the same focal length nor the resolution with the images from the Cityscapes dataset, we had to apply the procedure described in Section 4.5.3, where we disregard the baseline. However, we did not have the intrinsics of the camera which took the images in Roborace, so nothing could be done to obtain better results.

5.3. Results of our SemanticDepth pipeline on images featuring streets of Munich

In this section we present the results obtained when applying our SemanticDepth pipeline (Section 3) on a series of images featuring streets of the city of Munich (Figure 44), which we will refer to as the "Munich test set". Note that some images from the Munich test set resemble a Roborace-like environment, with walls or fences very close to each side of the road. This allows us to test the performance of our best Roborace-trained semantic segmentation model, the roborace750_-100Epochs, which we will use as the pipeline's segmentation model for this particular test.

(a) Original image from the Munich test set (test_1)

(b) Original image from the Munich test set (test_2)



(c) Original image from the Munich test set (test_3)

(d) Original image from the Munich test set (test_4)

(e) Original image from the Munich test set (test_5)

Figure 44 Munich test set, on which we have tested SemanticDepth.

As for the depth estimation model, we have used the only one we had at the time of writing this work, *i.e.*, a pre-computed monodepth [41] model trained on the Cityscapes stereo dataset. Recall that, according to the author of the monodepth architecture, it is unclear how a monodepth model behaves on images taken by cameras with different focal lengths and aspect ratios than those from the images the model was trained on (Section 2.5).

Recall as well we found out that the best solution to this problem was applying the process depicted at the end of Section 4.5.3, where in the end the conversion from disparities to 3D Point Clouds just depends on the intrinsics of the camera with which the input image was taken (Equation 4.1).

To compute these intrinsic parameters, we need some information about the camera used. In the case of the Munich test set, the images were taken by an iPhone 8. The smartphone's rear camera has a focal length of 3.99 mm and a sensor size of 1.22 μm, and produces 4032 x 3024 images. The theoretical focal length can be computed through Equation 5.1, where f is the focal length in pixels, F is the focal length in mm and $sensor\_size$ is the side in mm of one of the tiny sensors that integrates the whole camera.

$$f = F \cdot \frac{1}{sensor\_size} \tag{5.1}$$

Therefore, the camera's theoretical focal length in every direction (x and y) in pixels is 3270.5 (Equation 5.2).

$$f = \frac{3.99}{1.2e-3} = 3270.5 \quad pixels \tag{5.2}$$

Given that the input images are resized to 512 x 256, we must also scale both focal lengths accordingly. Ideally, before resizing we should crop the original 4032 x 3024 image in such a way the image's aspect ratio is the same as in a 256 x 512 image. By doing so, we would be dividing both the width and the height by the same scaling factor, and correspondingly both new focal lengths, *i.e.*, f_x and f_y, would also be obtained by dividing the previously obtained f by the same factor. However, we only need a reference as to how big the focal lengths are after resizing the input. Therefore, theoretically, we would have:

$$f_x = 512 \cdot \frac{3270.5}{4032} = 415.3 \quad pixels \tag{5.3}$$

$$f_y = 256 \cdot \frac{3270.5}{3024} = 276.86 \quad pixels \tag{5.4}$$

In practice, by manually calibrating the phone's camera, *i.e.*, by using OpencCV's calibration function on a series of already resized images (256 x 512) featuring a chessboard, we obtained that f_x = 480.088 and f_y = 322.316 pixels, values which are in the same order of magnitude as the theoretical ones obtained above.

These estimates gave us an idea of how big the focal length of Equation 4.1 should be for the particular case of the Munich test set. However, the iPhone's camera automatically focusses depending on light conditions, unless you lock the autofocus using a certain option within the smartphone. When taking the images of the Munich test set, this option was not working properly on the device, so therefore all 5 images from the Munich test set were taken with slightly different focal lengths.

For this reason, we decided to run our pipeline on the Munich test set using different focal lengths in Equation 4.1. We analyzed the results and effectively discovered how different focal lengths produced better results on some of the images and worse on others. For the sake of brevity, we will present the results corresponding to focal lengths of 380 and 580 pixels.

5.3.1. Results of SemanticDepth on the Munich test set

Table 18 shows the results obtained when applying SemanticDepth at a depth of 10 m on the Munich test set with a focal length of 380 pixels (upper subtable) and 580 pixels (lower subtable). For every frame, we present (a) the distance between what the pipeline considers to be a depth of 10 m and where a depth of 10 m really is; (b) the real width of the road at a depth of 10 m; (c) the width of the road computed through the naive approach; (d) the distance between the left and right fences computed through the advanced approach; (e) the absolute error between the real width of the road and the naive distance; (f) and the absolute error between the real width of the road and the advanced distance.

Note that we know where the real depth of 10 m is located because we placed a white stick at a depth of 10 m in every scene of the Munich test set before taking the images.

Finally, at the last row of each subtable we present the mean absolute error across the five images that constitute the Munich test set for both the naive and advanced approaches. Note that all distances are given in meters. Note as well that a negative value in the second column, *i.e.*, column "error", denotes that the algorithm has mislocated the depth of 10 m at a depth which in reality corresponds to less than 10 m. Positive values, on the contrary, indicate that the target depth of 10 m has been located in reality at z value greater than 10 m.

Table 18 Results of SemanticDepth on the Munich test set at a depth of 10 m, where **error** represents the distance between what the pipeline considers to be a depth of 10 m and where a depth of 10 m really is, **real width** is the real width of the road at a depth of 10 m, **naive** represents the width of the road at a depth of 10 m computed through the naive approach, **advanced** represents the distance between the left and right fences computed through the advanced approach at a depth of 10 m, **AE naive** is the absolute error between the real width of the road and the naive distance and **AE advanced** is the absolute error between the real width of the road and the advanced distance. Finally, **MAE** is the mean absolute error across the five examples of the Munich test set.

Focal length = 380 pixels

frame	**error** (m)	**real width** (m)	**naive** (m)	**advanced** (m)	**AE naive** (m)	**AE advanced** (m)
test_1	$\sim$ 0	5.3	6.24	6.81	0.94	1.51
test_2	$\sim$ +2.7	4.4	2.58	3.86	1.82	0.54
test_3	$\sim$ +3.2	5.4	3.66	4.19	1.74	1.21
test_4	$\sim$ +1.9	3.1	3.17	4.12	0.07	1.02
test_5	$\sim$ 0	4.6	4.73	5.51	0.13	0.91
MAE (m)	-	-	-	-	0.93	1.04

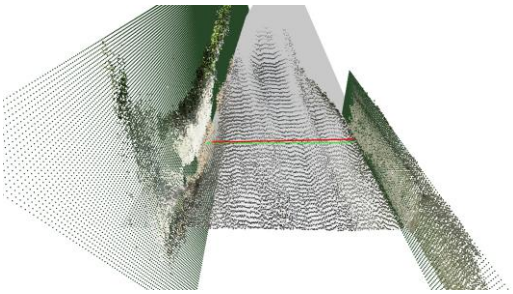
Focal length = 580 pixels

frame	**error** (m)	**real width** (m)	**naive** (m)	**advanced** (m)	**AE naive** (m)	**AE advanced** (m)
test_1	$\sim$ - 4	5.3	6.42	6.07	1.12	0.77
test_2	$\sim$ 0	4.4	4.42	4.04	0.02	0.36
test_3	$\sim$ 0	5.4	4.4	4.57	1.0	0.83
test_4	$\sim$ 0	3.1	2.74	4.07	0.36	0.97
test_5	$\sim$ - 6.8	4.6	6.30	5.91	1.70	1.31
MAE (m)	-	-	-	-	0.83	0.84

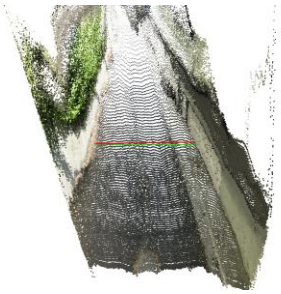
As a qualitative example, in Figure 45 we have applied SemanticDepth with a focal length of 580 on the frame "test_3" from the Munich test set. We display the frame's segmentation together with the information about the computed distances, the post-processed 3D Point Cloud with the naive and advanced distances (red and green lines, respectively) and the initial 3D Point Cloud, also with the computed distances drawn on it.

(a) Original image from the Munich test set (test_3)

(b) Output image displaying the computed distances on the segmented frame



(c) Post-processed 3D Point Cloud, featuring the fitted planes and the computed distances: naive (red line) and advanced (green line)



(d) Initial 3D Point Cloud, featuring the computed distances: naive (red line) and advanced (green line)

Figure 45 SemanticDepth applied on a frame from the Munich test set.

5.3.2. Inference time of SemanticDepth on the Munich test set

In Table 19 we have gathered the inference times of SemanticDepth on every image from the Munich test set. We have computed the total time per frame, as well as the time per task for 8 different tasks, namely (a) reading and resizing the frame; (b) segmenting the frame; (c) obtaining the disparity map; (d) converting the disparity map into a 3D Point Cloud; (e) denoising and fitting a plane to the 3D Point Cloud of the road; (f) computing the road's width through the naive approach; (g) denoising and fitting planes to each of the fences in the scene; (h) and computing the distance between left and right fences through the advanced approach.

Additionally, we have computed the mean time for every task, as well as the percentage that it represents in the mean overall inference time. These two measures are shown in the last two rows of Table 19. Note that all times given in Table 19 are in seconds.

Table 19 Inference times (in seconds) of SemanticDepth on the Munich test set, where **t_read** represents the reading and resizing of the original image; **t_seg** the inference time of the segmentation network; **t_disp** the inference time the depth estimation network needs to compute a disparity map of the scene; **t_to3D** the time needed to convert the disparity map to a 3D Point Cloud and to then use the segmetation masks obtained previously in order to compute 3D Point Clouds of the road and fences separately; **t_road** and **t_fences** the time to denoise the 3D Point Clouds of the road, left fence and right fence, as well as fitting planes to them; **t_naive** and **t_advanced** represent how long it takes to compute the width of the road (naive approach) and the fence-to-fence distance (advanced approach), respectively; finally, **t_total** is the total time needed to process a 4032 x 3024 image.

frame	t_read	t_seg	t_disp	t_to3D	t_road	t_naive	t_fences	t_advanced	t_total
1	0.3506	0.1521	0.0233	0.0106	0.1053	0.0013	0.0099	0.0016	0.6551
2	0.3605	0.1498	0.0233	0.0115	0.0952	0.0012	0.0144	0.0014	0.6577
3	0.3252	0.1688	0.0243	0.0110	0.0804	0.0012	0.0113	0.0016	0.6240
4	0.3551	0.1480	0.0221	0.0105	0.0338	0.0012	0.0240	0.0015	0.5965
5	0.3528	0.1456	0.0223	0.0094	0.1080	0.0013	0.0134	0.0014	0.6544
mean	0.3488	0.1528	0.0230	0.0106	0.0845	0.0012	0.0146	0.0015	0.6375
%	54.71%	23.97%	3.61%	1.7%	13.26%	0.19%	2.29%	0.24%	-

Note how more than half of the time needed to completely process a frame is dedicated to reading and resizing the frame (54.71%). The second most time consuming task is segmenting the image (23.97%), followed by the task of denoising the road and fitting a plane to it. The least time consuming tasks are the computation of the naive and advanced distances. We must bear in mind, however, that denoising and fitting planes to the road and fences are actually essential parts to the computation of both the naive and advanced distances.

6. Discussion of the Results

6.1. Comments on the Results in the Semantic Segmentation task

6.1.1. Overfitting

In all six semantic segmentation models we have trained in this work, *i.e.*, the four models trained on roborace425 and roborace750 for 100 and 200 epochs, and the two models trained on Cityscapes for 50 and 100 epochs, the graphs plotting the loss in the training and validation sets against the epochs show a common behaviour. On the one hand, the loss on the training set never stops decreasing. This makes perfect sense, since every epoch we are forcing our model to fit our training data slightly better.

On the other hand, we can appreciate how the loss on the validation set during training of all six models initially decreases for a certain number of epochs and then, at a certain point, stops getting smaller and even starts increasing slightly. This can be explained through the concept of overfitting, which takes place when we train a network for too long on a relatively small training set. Essentially, there is a point in training when the model starts fitting individual data points, instead of generalizing.

Qualitatively, the problem of overfitting could also be observed in Figures 35 and 36. Note how the segmentation is clearly much worse when training for 200 epochs than when doing it for 100.

Given that overfitting is indeed taking place when training for too many epochs, we must now address the question as to why the IoU on the validation set increases when training for more epochs, as tables 9, 11 and 14 show. Logic would suggest that if the model is overfitting (indicated by an upward trend of the loss on the validation set at a certain epoch), then the IoU of the validation set should not keep going upwards. On the contrary, it seems logical that it should start decreasing, since loss and IoU represent opposite concepts.

This situation we have just depicted, *i.e.*, validation loss increasing while validation IoU also increasing, seems to be a common topic of discussion on various online programming forums [62] [63]. One possible explanation could be that some examples with very bad predictions keep getting worse, *e.g.*, a road pixel predicted at 0.8 to belong to the class fence becomes predicted at 0.9 to belong to the class fence. Alternatively, this phenomenon could also be the result of some examples with very good predictions getting slightly worse predictions, *e.g.*, a road pixel predicted at 0.9 to belong to the class road becomes predicted at 0.8 to belong to the class road. Under these two circumstances, loss on the validation set would increase while accuracy would stay the same or even increase slightly. [62] [63].

However, we cannot assure completely that this behaviour is indeed rooted in what we have just depicted in the previous paragraph. In any case, what remains clear is that overfitting is indeed taking place when training for too many epochs, *i.e.*, 200 instead of 100 epochs, in the case of the Roborace datasets.

6.1.2. Selection of the best semantic segmentation model

We can clearly state that the best semantic segmentation model we have obtained using our FCN-8s implementation, capable of segmenting images into the classes road, fence and background, is the one we trained on the roborace750 dataset (Section 4.2) for 100 epochs, using the parameters shown in Table 5.

Therefore, we have made clear the importance of having created our own datasets of labelled images, since using one of the most renowned semantic segmentation datasets (Cityscapes [14]) was clearly not enough for our particular case (Table 17). We believe this is so due to the fact that images in Cityscapes and images in Roborace do not share neither the same aspect ratio, nor the same perspective, *i.e.*, images in Roborace are taken from a lower view point than those in the Cityscapes dataset. This causes the road to look notably different in each case. In Roborace, the road occupies most of the lower half of the image, while in Citysapes the road usually accounts for half of the lower half of the image.

Moreover, when we trained our FCN-8s implementation on Cityscapes we defined the class fence as a bundle of other Cityscapes classes, namely building, wall, fence and guard rail. Given that the most common of those elements in Cityscapes, *i.e.*, buildings, is not exactly what we are looking for in our Roborace dataset, it makes sense that the models trained on Cityscapes do not perform as well as those trained directly on Roborace images, since the later do look directly for fences, instead of buildings in general.

With respect to why training on roborace750 is better than training on roborace425, the answer is straight forward. The larger the dataset, the better the network learns to generalize, and therefore the better it will perform on images it has never seen before. Finally, and as we have already explained in Section 6.1.1, to the question of why training for 100 epochs instead of for 200 outputs better results, we can state that when training for too many epochs on a relatively small dataset (roborace750 is relatively small compared to Cityscapes, for example), at a certain point the model starts to fit individual data points, *i.e.*, images, instead of generalizing.

6.2. Comments on the Results in the Depth Estimation task

Given that the monodepth [41] model we are using in this work was trained on Cityscapes, the fact that the 3D reconstruction of a Cityscapes scene (Figure 42) using this model produces such outstanding results came as no surprise. Indeed, Godard [41] had already tested this on his work.

For Roborace frames, however, generating the disparity map of the scene applying the same Cityscapes-trained model does not produce such optimal results. This was shown in Figure 43. As we have already mentioned in Section 2.5, in its current form, monodepth only performs to its full potential when presented with images which have a focal length identical to the dataset it was trained on. Therefore, the less optimal results achieved on Roborace images using a pre-computed monodepth model trained on Cityscapes were also expected.

6.3. Comments on the Results obtained using the SemanticDepth Pipeline on the Munich Test Set

In this section we will first discuss the results obtained using SemanticDepth on the Munich test set (a series of five images featuring streets of Munich) and compare them with the real distances (Section 6.3.1). Then, we will discuss the inference times achieved by the pipeline on each one of test images (Section 6.3.2).

6.3.1. Comments on the Naive and Advanced Approaches of SemanticDepth

To begin with, and in order to get an idea of the performance of SemanticDepth on the Munich test set, it is worth observing the results qualitatively. Figure 45 shows how both the semantic segmentation task and the depth estimation task achieve a fairly good segmentation of the image and 3D reconstruction of the scene, respectively. Recall that the semantic segmentation model used for the case of the Munich test set is the model trained on roborace750 (Section 4.2.2) for 100 epochs.

Therefore, the first conclusion we can derive from the qualitative result shown in subfigure 45b is that the aforementioned segmentation model, despite having been trained on Roborace images, is capable of generalizing to environments it has never seen before, such as the city of Munich. Not only that, but also is the model capable of performing a good segmentation despite the fact that the aspect ratio in the Munich test set is completely different from that of the roborace750 dataset (4032 x 3024 to 1200 x 1600, respectively).

Secondly, with respect to the pre-trained monodepth model used in SemanticDepth on the Munich test set, and still from a qualitative point of view, we can conclude by analyzing subfigures 45c and 45d that a Cityscapes-trained monodepth model is able to predict reasonably good depth information when presented with images that resemble those from the Cityscapes dataset, despite the fact that neither the focal length nor the aspect ratio of the Munich test set match those of Cityscapes.

All these statements can be confirmed by looking at the data presented in Table 18. Recall that, due to the fact that the focal length of the camera with which the images from the Munich test set were taken was not equal for all frames, we had to test several focal lengths in the range of the theoretical one. In doing so, we first found out how the pipeline performed well on two of the frames, namely test_1 and test_5, when using a focal length of 380 pixels and badly on the rest of the frames, namely test_2, test_3 and test_4.

Secondly, we discovered that when using a focal length of 580 pixels instead, the pipeline performed remarkably well on three of the frames, namely test_2, test_3 and test_4, and relatively badly on the other two. Esentially, we concluded that two of the five images that conform the Munich test set had been taken with a focal length of 380 pixels and the other three with a focal length of 580 pixels. The way we noticed this phenomenon was by looking at the error between what the pipeline thought was a depth of 10 meters and where a depth of 10 meters really was situated (Table 18, second column). Recall that we knew where the real depth of 10 m was located because we had placed a white stick at a depth of 10 m in every scene before taking the images.

Therefore, it is worth making a distinction between the results achieved using a focal length of 380 and 580 pixels, so that we can discuss over valid predictions. For this purpose, Table 20 merges the results obtained for frames test_1 and test_5 using a focal length of 380 together with the results obtained for frames test_2, test_3 and test_4 using a focal length of 580 pixels.

Table 20 Merged results of SemanticDepth on the Munich test set at a depth of 10 m, where **error** represents the distance between what the pipeline considers to be a depth of 10 m and where a depth of 10 m really is, **real width** is the real width of the road at a depth of 10 m, **naive** represents the width of the road computed through the naive approach, **advanced** represents the distance between the left and right fences computed through the advanced approach, **AE naive** is the absolute error between the real width of the road and the naive distance and **AE advanced** is the absolute error between the real width of the road and the advanced distance. Finally, **MAE** is the mean absolute error across the five examples of the Munich test set.

Focal length = 380 pixels

frame	**error** (m)	**real width** (m)	**naive** (m)	**advanced** (m)	**AE naive** (m)	**AE advanced** (m)
test_1	~ 0	5.3	6.24	6.81	0.94	1.51
test_5	~ 0	4.6	4.73	5.51	0.13	0.91

Focal length = 580 pixels

frame	**error** (m)	**real width** (m)	**naive** (m)	**advanced** (m)	**AE naive** (m)	**AE advanced** (m)
test_2	~ 0	4.4	4.42	4.04	0.02	0.36
test_3	~ 0	5.4	4.40	4.57	0.99	0.82
test_4	~ 0	3.1	2.74	4.07	0.35	0.97

Corrected Mean Absolute Error of Naive and Advanced Approaches

MAE (m)	-	-	-	-	0.48	0.91

An important conclusion that we can extract from the values in Table 20 is that the naive approach outperforms the advanced approach. Indeed, the Mean Absolute Error of the former is 0.48 m, whilst that of the later is 0.91 m. This could be explained by the fact that in the advanced approach the complexity is much higher than in the naive approach, given that not only do we have to denoise and fit a plane to the 3D Point Cloud of the road, but also to both 3D Point Clouds of the fences, left and right.

Another feasible explanation as to why the naive approach outperforms the advanced approach is that in the Munich test set the fences are not perfectly perpendicular to the road in all five images. On the contrary, in the Munich test set the fences can be cars. Such is the case of the frames test_1 and test_4, where the difference in performance between the naive and the advanced approach is clear.

However, regardless of what the results indicate, we consider that computing the road's width both through the naive and the advanced approaches provides a more robust understanding of the scene, since at any time either the naive or the advanced approach could suffer from a large range of different problems. By considering the output of both approaches, the decisions the vehicle would ultimately have to take would be better informed.

The problems we are referring to above are essentially rooted in not 100%-perfect segmentations of the road, scenes in which there is only one fence visible (very common in Roborace) or not optimal depth estimations of the scene. Clearly, the last problem could be easily handled by generating a depth estimation model trained on images with the same focal length and aspect ratio as the ones on which we would then apply the model. In other words, if we were to apply SemanticDepth on Roborace images, we would need to train monodepth or some other depth estimation network on Roborace images, as we already mentioned in Section 4.3.

6.3.2. Comments on the Inference Time of SemanticDepth

For the convenience of the reader, we show again in Table 21 the mean inference times computed for each one of the eight tasks (see caption in Table 21 for an explanation on each task) that SemanticDepth carries out upon the input, as well as the percentage of the total time that each task represents.

Table 21 Mean inference times (in seconds) of SemanticDepth on the Munich test set, where **t_read** represents the reading and resizing of the original image; **t_seg** the inference time of the segmentation network; **t_disp** the inference time the depth estimation network needs to compute a disparity map of the scene; **t_to3D** the time needed to convert the disparity map to a 3D Point Cloud and to then use the segmetation masks obtained previously in order to compute 3D Point Clouds of the road and fences separately; **t_road** and **t_fences** the time to denoise the 3D Point Clouds of the road, left fence and right fence, as well as fitting planes to them; **t_naive** and **t_advanced** represent how long it takes to compute the width of the road (naive approach) and the fence-to-fence distance (advanced approach), respectively; finally, **t_total** is the total time needed to process a 4032 x 3024 image.

frame	**t_read**	**t_seg**	**t_disp**	**t_to3D**	**t_road**	**t_naive**	**t_fences**	**t_advanced**	**t_total**
mean	0.3488	0.1528	0.0230	0.0106	0.0845	0.0012	0.0146	0.0015	0.6375
%	54.71%	23.97%	3.61%	1.7%	13.26%	0.19%	2.29%	0.24%	-

On the Munich test set, SemanticDepth takes an average of 0.6375 s to process a single image. The most relevant aspect to note is that the time required for reading and resizing the input image accounts for more than half of the total inference time. This makes complete sense, given that images in the Munich test set have an initial resolution of 4032 x 3024, much higher than that of the images in Cityscapes (1024 x 2048) or in Roborace (1200 x 1600).

Indeed, when applying SemanticDepth to Roborace images, this reading and resizing time falls to 0.07 s, approximately, which means that the total inference time for Roborace images would be around 0.36 s, or roughly 3 FPS. Given that the ultimate goal of SemanticDepth is to some day be applied onto Roborace images, this value is reassuring.

With regards to the second most time consuming task, segmentation, a time of 0.1528 s on 512 x 256 images using an FCN-8s model was completely expected, given the data from the literature (Table 4). If we do not take into account the reading and resizing time, segmentation is the bottleneck in the pipeline. Indeed, in future work, it is one of the tasks that would first need to be addressed with regards to its inference time. However, as a first approach to the problem, we consider that 0.1528 s is acceptable.

The third most time consuming task corresponds to denoising and fitting a plane to the crude 3D Point Cloud of the road, and it accounts for more than 13% of the total inference time. Essentially, this is due to the large number of points that belong to the 3D Point Cloud of the road in comparison to the number of points that belong to the left and right fences. For the later, *i.e.*, the fences, we only need an average of 0.0146 s to denoise the Point Clouds and fit planes to them.

All in all, the actual SemanticDepth pipeline is able of processing a 4032 x 3024 image in roughly 0.64 s and a 1200 x 1600 image (Roborace images) in approximately 0.36 s.

7. Summary and Outlook

7.1. Summary

In a nutshell, the main goal of this research has been successfully accomplished, *i.e.*, building a computer vision pipeline, SemanticDepth (Figure 19), that allows a self-driving car (particularly, the Robocar [6]) to extract information from its environment, in such a way that it is able to tell where it is located within the road it is driving on by predicting the width of the road at a certain depth.

To achieve this global goal, we have had to accomplish several smaller objectives, which we list next:

1. We have built our own implementation of an FCN-8s [18] architecture, a semantic segmentation network that achieves pixel-wise classification of an image.

2. We have created our own labelled semantic segmentation dataset (Section 4.2), comprised of a total of 750 images from 3 different Roborace race tracks (roborace750 dataset). For comparison purposes, we have also put together a smaller dataset comprised of 425 images (roborace425).

3. We have trained our FCN-8s implementation on the created datasets and tested the resulting models on a common test set. After training and comparing various different models, the one with highest performance is that trained for 100 epochs on the roborace750 dataset, using the parameters shown in Table 5. The mean class-IoU achieved by this model on the Roborace test set is 87.63% (3 classes: road, fence and background).

   Moreover, we have clearly observed a common problem in any DL task: overfitting. We have seen how training for too many epochs can lead to the network fitting noise, with the corresponding loss of generality that this problem causes.

   Additionally, we have also trained our FCN-8s implementation on Cityscapes for 50 and 100 epochs and then applied these models on the Roborace test set. Between these two models, that with highest mean class-IoU is that trained for 100 epochs, with a mean class-IoU of 52.855%, much lower than that obtained when training on roborace750 for also 100 epochs, 87.63% (Table 17). This clearly justifies the creation of our own semantic segmentation dataset of Roborace images.

4. We have successfully organized all available stereo data provided by the Roborace competition into a dataset of stereo pairs on which to train a depth estimation DNN. Particularly, we would have like to train a certain monocular depth estimation network, *i.e.*, the monodepth [41] architecture, on this stereo dataset. However, given that the stereo pairs from the Roborace competition were poorly synchronized and that we were not able to obtain the stereo rig parameters from the organization of the competition (necessary to rectify the pairs), this task has not been accomplished.

5. Even though we could not obtain a depth estimation model trained on our own Roborace stereo dataset, we have successfully integrated a pre-computed monodepth [41] model trained on Cityscapes into our pipeline.

By putting semantic segmentation and depth estimation together, SemanticDepth is able to reconstruct a 3D Point Cloud of the scene and then extract relevant information about the vehicle's location with respect to the sides of the road. For the task of distance computation in a 3D Point Cloud, we have developed two approaches. In the naive approach we essentially look for the furthest left and right (3D) points of the road at a certain depth from the vehicle.

In the advanced approach we first fit planes to the road and fences, then find the intersection between these planes and finally compute the distance between the intersected lines at a certain depth.

We have successfully tested SemanticDepth, with its two approaches, on a series of images taken in the streets of Munich (Munich test set), managing to obtain predictions very close to the real width of the road at a certain depth, as shown in Table 20. The naive approach has proven to produce predictions with a lower absolute error than those predicted through the advanced approach, 0.48 m and 0.91 m, respectively. This could lead us into thinking that only the naive approach should be used in SemanticDepth. Nontheless, we consider that both approaches are complementary, each adding robustness to the pipeline in different situations and against different problems.

Finally, we have gathered the inference times of the pipeline on the different images of the Munich test set, which have a resolution of 4032 x 3024. On these images, the actual version of SemanticDepth takes roughly an average of 0.64 s per image. Over 50% of this time is dedicated to reading and resizing the input image. On Roborace images, which have a resolution of 1200 x 1600, we have computed that this reading and resizing time is much smaller, which results in a total inference time of approximately 0.36 s per image.

All in all, given the highly positive semantic segmentation results that our roborace750_100Epochs model achieves on the Roborace test set (87.63% mean class-IoU on classes road, fence and background) and also the acceptable inference time of SemanticDepth on Roborace image, we consider that the pipeline could someday be applied, though at a relatively low speed, on the Roborace competition to increase the situational awereness of the Robocar.

7.2. Outlook

Many things can be improved in SemanticDepth, the deep learning-based computer vision pipeline developed in this work. Both in the task of semantic segmentation and in that of depth estimation the options are numerous. Improvements could also be made in the last steps of the pipeline, *i.e.*, post-processing of the 3D Point Cloud and computation of relevant distances in the reconstructed 3D scene.

In the semantic segmentation task, the first clear step towards improving results is enlarging our own semantic segmentation dataset of Roborace images. Thus far, we have labelled a total of 750 images from 3 different Roborace race tracks (250 for each city), out of a total of more than 13000 available frames. The bigger the dataset, the better the results will be, regardless of what semantic segmentation architecture we use.

Also related to the task of semantic segmentation, we believe it would be of great interest to train different semantic segmentation architectures on our Roborace dataset (or a larger version of it) and compare the results with the ones already obtained in this work by using an FCN-8s architecture. The ERFNet [23] could be an interesting option, given that it is one of the state-of-the-art architectures with lowest inference time, key trait in ITSs.

With regards to the task of camera-based depth estimation in the Roborace competition, the main concern at this point lies on the quality of the stereo pairs provided by the organization. Therefore, before exploring any new stereo vision related approaches, we recommend that the necessary stereo calibration parameters be acquired. Either directly as a list of parameters or by calibrating the stereo rig on various image pairs featuring a chessboard taken by the vehicles's cameras. Only then will it be possible to train a depth-estimation DL model, such as the monodepth [41] architecture, on the dataset of Roborace stereo pairs that we have already built. We do not recommend exploring the option of standard stereo matching, at least not until the stereo paris provided by Roborace have better synchronization.

With regards to the pipeline in general, it would be of high interest to parallelize the semantic segmentation and depth estimation processes, since in the actual version of SemanticDepth we first run semantic segmentation, then depth estimation and finally merge the results from each task. By running both processes simultaneuously, the speed of the pipeline, now around 0.36 seconds per 1200 x 1600 frame (Roborace resolution), could be reduced notably.

Finally, it would also be interesting to explore the Python Bindings for the Point Cloud Library [64], instead of using our own home-made version. It would definitely reduce the time needed to post-process the raw 3D Point Cloud of the road and fences, as well as the computation of the naive and advanced distances.

List of Figures

List of Tables

Bibliography

[1] Global status report on road safety 2015. http://www.who.int/violence_injury_prevention/road_safety_status/2015/en/.

[2] Aurora innovation, inc. https://aurora.tech/.

[3] Waymo - the google self-driving car project. https://www.google.com/selfdrivingcar/.

[4] An oral history of the darpa grand challenge, the grueling robot race that launched the self-driving car. https://www.wired.com/story/darpa-grand-challenge-2004-oral-history/.

[5] Mike Daily, Swarup Medasani, Reinhold Behringer, and Mohan Trivedi. Self-driving cars. *Computer*, 50(12):18–23, 2017.

[6] Roborace. https://roborace.com/.

[7] Gurveen Kaur and Dinesh Kumar. Lane detection techniques: A review. *International Journal of Computer Applications*, 112(10), 2015.

[8] Anik Saha, Dipanjan Das Roy, Tauhidul Alam, and Kaushik Deb. Automated road lane detection for intelligent vehicles. *Global Journal of Computer Science and Technology*, 2012.

[9] ZuWhan Kim. Robust lane detection and tracking in challenging scenarios. *IEEE Transactions on Intelligent Transportation Systems*, 9(1):16–26, 2008.

[10] Mohamed Aly. Real time detection of lane markers in urban streets. In *Intelligent Vehicles Symposium, 2008 IEEE*, pages 7–12. IEEE, 2008.

[11] Cs231n convolutional neural networks for visual recognition. http://cs231n.github.io/neural-networks-1/.

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

[13] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[14] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[15] Jamie Shotton, Matthew Johnson, and Roberto Cipolla. Semantic texton forests for image categorization and segmentation. In *Computer vision and pattern recognition, 2008. CVPR*

2008. IEEE Conference on, pages 1–8. IEEE, 2008.

[16] Jamie Shotton, Toby Sharp, Alex Kipman, Andrew Fitzgibbon, Mark Finocchio, Andrew Blake, Mat Cook, and Richard Moore. Real-time human pose recognition in parts from single depth images. *Communications of the ACM*, 56(1):116–124, 2013.

[17] Dan Ciresan, Alessandro Giusti, Luca M Gambardella, and Jürgen Schmidhuber. Deep neural networks segment neuronal membranes in electron microscopy images. In *Advances in neural information processing systems*, pages 2843–2851, 2012.

[18] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015.

[19] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

[20] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[22] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *arXiv preprint arXiv:1511.00561*, 2015.

[23] Eduardo Romera, José M Alvarez, Luis M Bergasa, and Roberto Arroyo. Efficient convnet for real-time semantic segmentation. In *Intelligent Vehicles Symposium (IV), 2017 IEEE*, pages 1789–1794. IEEE, 2017.

[24] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. Semantic image segmentation with deep convolutional nets and fully connected crfs. *CoRR*, abs/1412.7062, 2014.

[25] Shuai Zheng, Sadeep Jayasumana, Bernardino Romera-Paredes, Vibhav Vineet, Zhizhong Su, Dalong Du, Chang Huang, and Philip HS Torr. Conditional random fields as recurrent neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1529–1537, 2015.

[26] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015.

[27] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *arXiv preprint arXiv:1606.00915*, 2016.

[28] Guosheng Lin, Anton Milan, Chunhua Shen, and Ian Reid. Refinenet: Multi-path refinement networks with identity mappings for high-resolution semantic segmentation. *arXiv preprint arXiv:1611.06612*, 2016.

[29] Tobias Pohlen, Alexander Hermans, Markus Mathias, and Bastian Leibe. Full-resolution residual networks for semantic segmentation in street scenes. *arXiv preprint*, 2017.

[30] Golnaz Ghiasi and Charless C Fowlkes. Laplacian pyramid reconstruction and refinement for semantic segmentation. In *European Conference on Computer Vision*, pages 519–534. Springer, 2016.

[31] Adam Paszke, Abhishek Chaurasia, Sangpil Kim, and Eugenio Culurciello. Enet: A deep neural network architecture for real-time semantic segmentation. *arXiv preprint arXiv:1606.02147*, 2016.

[32] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. *arXiv preprint arXiv:1706.05587*, 2017.

[33] Hengshuang Zhao, Jianping Shi, Xiaojuan Qi, Xiaogang Wang, and Jiaya Jia. Pyramid scene parsing network. *arXiv preprint arXiv:1612.01105*, 2016.

[34] Guosheng Lin, Chunhua Shen, Anton Van Den Hengel, and Ian Reid. Efficient piecewise training of deep structured models for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3194–3203, 2016.

[35] Ziwei Liu, Xiaoxiao Li, Ping Luo, Chen-Change Loy, and Xiaoou Tang. Semantic image segmentation via deep parsing network. In *Computer Vision (ICCV), 2015 IEEE International Conference on*, pages 1377–1385. IEEE, 2015.

[36] Michael Treml, José Arjona-Medina, Thomas Unterthiner, Rupesh Durgesh, Felix Friedmann, Peter Schuberth, Andreas Mayr, Martin Heusel, Markus Hofmarcher, Michael Widrich, et al. Speeding up semantic segmentation for autonomous driving. In *MLITS, NIPS Workshop*, 2016.

[37] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International journal of computer vision*, 47(1-3):7–42, 2002.

[38] Yasutaka Furukawa, Carlos Hernández, et al. Multi-view stereo: A tutorial. *Foundations and Trends® in Computer Graphics and Vision*, 9(1-2):1–1 48, 2015.

[39] René Ranftl, Vibhav Vineet, Qifeng Chen, and Vladlen Koltun. Dense monocular depth estimation in complex dynamic scenes. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4058–4066, 2016.

[40] Austin Abrams, Christopher Hawley, and Robert Pless. Heliometric stereo: Shape from sun position. In *Computer Vision–ECCV 2012*, pages 357–370. Springer, 2012.

[41] Clément Godard, Oisin Mac Aodha, and Gabriel J Brostow. Unsupervised monocular depth

estimation with left-right consistency. In *CVPR*, volume 2, page 7, 2017.

[42] Jure Zbontar and Yann LeCun. Stereo matching by training a convolutional neural network to compare image patches. *Journal of Machine Learning Research*, 17(1-32):2, 2016.

[43] L'ubor Ladickỳ, Christian Häne, and Marc Pollefeys. Learning the matching function. *arXiv preprint arXiv:1502.00652*, 2015.

[44] Wenjie Luo, Alexander G Schwing, and Raquel Urtasun. Efficient deep learning for stereo matching. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5695–5703, 2016.

[45] Nikolaus Mayer, Eddy Ilg, Philip Hausser, Philipp Fischer, Daniel Cremers, Alexey Dosovitskiy, and Thomas Brox. A large dataset to train convolutional networks for disparity, optical flow, and scene flow estimation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4040–4048, 2016.

[46] Alexey Dosovitskiy, Philipp Fischer, Eddy Ilg, Philip Hausser, Caner Hazirbas, Vladimir Golkov, Patrick van der Smagt, Daniel Cremers, and Thomas Brox. Flownet: Learning optical flow with convolutional networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2758–2766, 2015.

[47] Adrien Gaidon, Qiao Wang, Yohann Cabon, and Eleonora Vig. Virtual worlds as proxy for multi-object tracking analysis. *arXiv preprint arXiv:1605.06457*, 2016.

[48] Ashutosh Saxena, Min Sun, and Andrew Y Ng. Make3d: Learning 3d scene structure from a single still image. *IEEE transactions on pattern analysis and machine intelligence*, 31(5):824–840, 2009.

[49] Derek Hoiem, Alexei A Efros, and Martial Hebert. Automatic photo pop-up. In *ACM transactions on graphics (TOG)*, volume 24, pages 577–584. ACM, 2005.

[50] Fayao Liu, Chunhua Shen, Guosheng Lin, and Ian Reid. Learning depth from single monocular images using deep convolutional neural fields. *IEEE transactions on pattern analysis and machine intelligence*, 38(10):2024–2039, 2016.

[51] David Eigen and Rob Fergus. Predicting depth, surface normals and semantic labels with a common multi-scale convolutional architecture. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2650–2658, 2015.

[52] David Eigen, Christian Puhrsch, and Rob Fergus. Depth map prediction from a single image using a multi-scale deep network. In *Advances in neural information processing systems*, pages 2366–2374, 2014.

[53] Bo Li, Chunhua Shen, Yuchao Dai, Anton van den Hengel, and Mingyi He. Depth and surface normal estimation from monocular images using regression on deep features and hierarchical crfs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1119–1127, 2015.

[54] Yuanzhouhan Cao, Zifeng Wu, and Chunhua Shen. Estimating depth from monocular images as classification using deep fully convolutional residual networks. *IEEE Transactions on Circuits and Systems for Video Technology*, 2017.

[55] Iro Laina, Christian Rupprecht, Vasileios Belagiannis, Federico Tombari, and Nassir Navab. Deeper depth prediction with fully convolutional residual networks. In *3D Vision (3DV), 2016 Fourth International Conference on*, pages 239–248. IEEE, 2016.

[56] Junyuan Xie, Ross Girshick, and Ali Farhadi. Deep3d: Fully automatic 2d-to-3d video conversion with deep convolutional neural networks. In *European Conference on Computer Vision*, pages 842–857. Springer, 2016.

[57] Ravi Garg, Vijay Kumar BG, Gustavo Carneiro, and Ian Reid. Unsupervised cnn for single view depth estimation: Geometry to the rescue. In *European Conference on Computer Vision*, pages 740–756. Springer, 2016.

[58] Alberto Garcia-Garcia, Sergio Orts-Escolano, Sergiu Oprea, Victor Villena-Martinez, and Jose Garcia-Rodriguez. A review on deep learning techniques applied to semantic segmentation. *arXiv preprint arXiv:1704.06857*, 2017.

[59] Tensorflow. `https://www.tensorflow.org/`.

[60] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

[61] Marius Cordts. Cityscapes scripts for the cityscapes dataset. `https://github.com/mcordts/cityscapesScripts`.

[62] Validation loss increases while validation accuracy is still improving. `https://github.com/keras-team/keras/issues/3755`.

[63] How is it possible that validation loss is increasing while validation accuracy is increasing as well? `https://stats.stackexchange.com/questions/282160/how-is-it-possible-that-validation-loss-is-increasing-while-validation-accuracy`.

[64] Python bindings for the point cloud libray. `http://strawlab.github.io/python-pcl/`.

[65] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 234–241. Springer, 2015.

[66] Philipp Krähenbühl and Vladlen Koltun. Efficient inference in fully connected crfs with gaussian edge potentials. In *Advances in neural information processing systems*, pages 109–117, 2011.

[67] Adrian Kaehler and Gary Bradski. *Learning OpenCV 3: computer vision in C++ with the OpenCV library*. O'Reilly Media, Inc., 2016.

[68] Opencv - camera calibration and 3d reconstruction. https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html.

[69] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[70] IEEE. International Symposium on Biomedical Imaging. http://biomedicalimaging.org/2015/program/isbi-challenges/.

[71] Kaggle Inc. Kaggle Competitions. https://www.kaggle.com/competitions.

[72] Protocol buffers. https://developers.google.com/protocol-buffers/.

[73] Kurt Konolige. Small vision systems: hardware and applications. *Kanagawa, Japan*, 1997.

A. Appendix. Semantic Segmentation Architectures

In this Appendix we have gathered some of the most relevant architectures used along the years for the taks of semantic segmentation. For every architecture we provide a short summary (in essence, the abstract from the original paper), an explanation of its key features, a listing of its most important contributions and a table showing their performance on the test sets of VOC2012 [19] and Cityscapes [14],

A.1. FCN: Fully Convolutional Networks for Semantic Segmentation

Submitted on 14 Nov 2014.

Summary:

The FCN, or more specifically the FCN-8s architecture [18], adapts contemporary classification networks (AlexNet [12], VGG [20] and GoogLeNet [69]) and transfers their learned representations by fine-tuning to the segmentation task. It also defines a novel architecture (the decoder) that combines semantic information from a deep, coarse layer with appearance information from a shallow, fine layer to produce accurate and detailed segmentations. It achieved state-of-the-art segmentation on PASCAL VOC (20% relative improvement to 62.2% mean IU on 2012), NYUDv2, and SIFT Flow. Inference using this architecture takes around 0.2 seconds for a 480x640 image.

Key features:

The fully connected layers (fc6, fc7) of classifcation networks such as VGG16 are replaced in the original FCN-8s implementation by 1-by-1 convolutions in order to preserve spatial information. This convolutionalizing produces a class presence heatmap in low resolution, which is then upsampled using billinearly initialized deconvolutions. Besides, at each stage of upsampling takes place an addition between the upsampled layer and features from coarser but higher resolution feature maps from lower layers in the VGG16 (conv4 and conv3). This addition procedure is known as skip connections. [18]

Key contributions and Benchmarks:

- Popularized the use of end to end CNNs for semantic segmentation.
- Re-purposed ImageNet pretrained networks for segmentation.
- Introduced skip connections to improve over the coarseness of upsampling.

Table 22 FCN-8s Benchmarks on VOC2012 and Cityscapes.

Architecture	**Score on VOC2012** (mean IoU (%))	**Score on Cityscapes** (mean IoU (%))
FCN-8s	67.2	65.3

A.2. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation

Submitted on 2 Nov 2015.

Summary:

The novelty of SegNet [22] lies in the manner in which the decoder upsamples its lower resolution input feature map(s). More specifically, the decoder uses pooling indices computed in the max-pooling step of the corresponding encoder to perform non-linear upsampling. This eliminates the need for learning to upsample. The upsampled maps are sparse and are then convolved with trainable filters to produce dense feature maps.

Key features:

The FCN-8s, despite upconvolutional layers and a few skip connections, produces coarse segmentation maps. In the SegNet architecture more skip connections are introduced in order to counteract this effect. However, instead of copying the encoder features as it was done in the FCN-8 implementation, indices from each max-pooling layer in the encoder are stored and later used to upsample the corresponding feature map in the decoder. This makes SegNet much more memory-efficient than the FCN-8s.

Key contributions and Benchmarks:

- Max-pooling indices transferred to decoder to improve the segmentation resolution.

Table 23 SegNet Benchmarks on VOC2012 and Cityscapes.

Architecture	**Score on VOC2012** (mean IoU (%))	**Score on Cityscapes** (mean IoU (%))
SegNet	59.9	57.0

A.3. U-Net: Convolutional Networks for Biomedical Image Segmentation

Submitted on 18 May 2015.

Summary:

The U-Net [65] architecture consists of a contracting path to capture context and a symmetric expanding path that enables precise localization. In their paper, they show that such a network can be trained end-to-end from very few images and that it outperforms the prior best method (a sliding-window convolutional network) on the ISBI challenge [70] for segmentation of neuronal structures in electron microscopic stacks. Using the same network trained on transmitted light microscopy images (phase contrast and DIC) they won the ISBI cell tracking challenge 2015 in these categories by a large margin.

Key features:

U-Net achieved state-of-the-art results on EM Stacks dataset, which contained only 30 densely annoted medical images and other medical image datasets. While U-Net was initally published for bio-medical segmentation, thanks to the utility of the network and its capacity to learn from very little data, it has been successfully used in other fields, such as satellite image segmentation. It has also been part of winning solutions of many Kaggle contests [71] on medical image segmentation. Its architecture is shown in Figure 46.

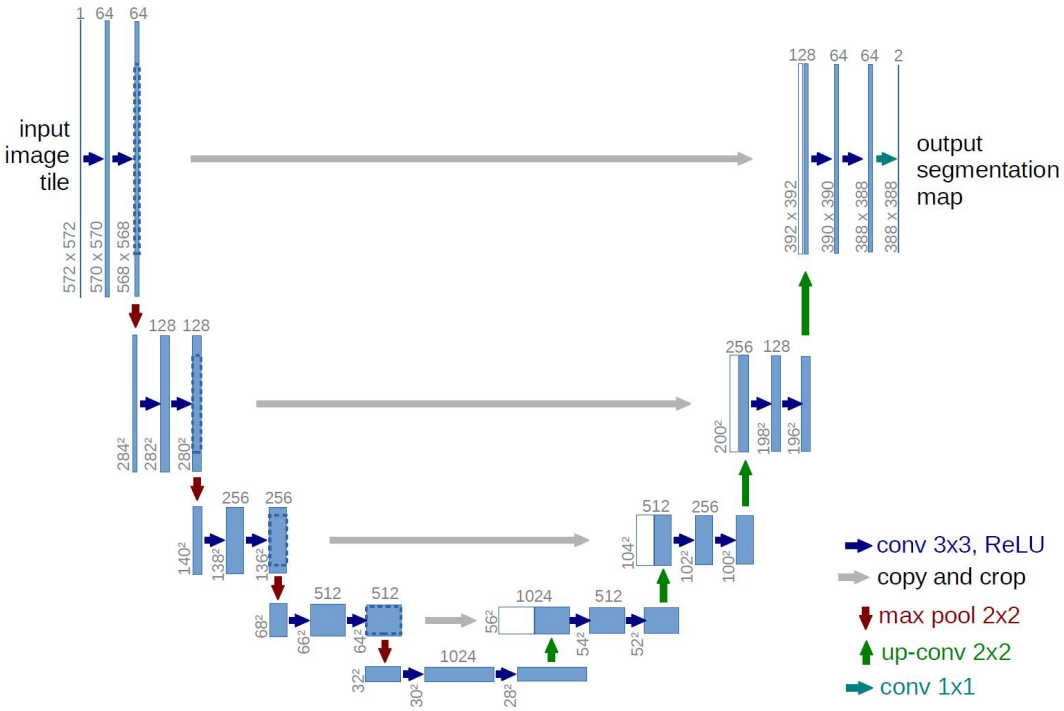


Figure 46 U-Net architecture [65].

Key contributions and Benchmarks:

- This architecture showed that networks can also learn from very little data if data augmentation is enforced.

Table 24 U-Net Benchmarks on VOC2012 and Cityscapes.

Architecture	**Score on VOC2012** (mean IoU)	**Score on Cityscapes** (mean IoU)
U-Net	-	-

Note that even though the U-Net architecture was not tested neither on the PASCAL VOC2012 nor on the Cityscapes benchmarks, it was still a very relevant work that provided important knowledge to the semantic segmentation research world. Thus, it is worth taking it into account in our revision.

A.4. Dilated Convolutions: Multi-Scale Context Aggregation by Dilated Convolutions

Submitted on 23 Nov 2015.

Summary:

State-of-the-art models for semantic segmentation are based on adaptations of convolutional networks that had originally been designed for image classification. However, dense prediction and image classification are structurally different. The Dilated Convolutions [26] architecture is specifically designed for dense prediction. It uses dilated convolutions to systematically aggregate multi-scale contextual information without losing resolution. The architecture is based on the fact that dilated convolutions support exponential expansion of the receptive field without loss of resolution or coverage. Dilated Convolutions managed to increase the accuracy of state-of-the-art semantic segmentation systems.

Key features:

As mentioned before, pooling helps in classification networks because the receptive field increases. However, this is not the optimal approach for segmentation, since pooling decreases the resolution. The solution proposed by Yu is the use of dilated convolutions. Dilated convolution layers (also known as atrous convolution) allow for exponential increase in field of view without decrease of spatial dimensions. [26]

In essence, the two last pooling layers from a pretrained classification networks such as VGG are removed, at the same time that classical convolutional layers are replaced with dilated convolutions.

Key contributions and Benchmarks:

- Uses dilated convolutions, a convolutional layer for dense predictions.

Table 25 U-Net Benchmarks on VOC2012 and Cityscapes.

Architecture	**Score on VOC2012** (mean IoU)	**Score on Cityscapes** (mean IoU)
Dilation10	75.3	67.1

A.5. DeepLab (v1 and v2): Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs

v1: Semantic Image Segmentation with Deep Convolutional Nets and Fully Connected CRFs [24].

Submitted on 22 Dec 2014

v2: DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs [27].

Submitted on 2 Jun 2016

Next, we will cover only the version 2 of the DeepLab architecture.

Short summary:

This work addresses the task of semantic image segmentation with Deep Learning and makes three main contributions. First, the use of convolution with upsampled filters, or 'atrous convolution'. Second, it proposes ASPP (Atrous Spatial Pyramid Pooling) to robustly segment objects at multiple scales. Third, it improves the localization of object boundaries by combining methods from Deep CNNs and probabilistic graphical models by using fully connected CRF.

Key features:

Through the use of atrous convolution the network explicitly controls the resolution at which feature responses are computed within Deep CNNs. This also allows to effectively enlarge the field of view of filters to incorporate larger context without increasing the number of parameters or the amount of computation.

Secondly, DeepLabv2 introduces the use of ASPP. This technique probes an incoming convolutional feature layer with filters at multiple sampling rates and fields-of-views, thus capturing objects as well as image context at multiple scales.

A third essential item in DeepLabv2 is the use of fully connected CRF. The motivation is that the commonly deployed combination of max-pooling and downsampling in classical deep CNNs achieves invariance but has a toll on localization accuracy. This is overcome by combining the responses at the final deep CNN layer with a fully connected CRF, which is trained separately as a post processing step.

What exactly is a CRF is explained in [66]. Shortly, this postprocessing method is usually used to improve the segmentation. CRFs are graphical models that soften segmentation based on the underlying image intensities, that is, based on the idea that similar intensity pixels tend to be labeled as the same class. By using CRFs a score boost of 1-2% can be obtained (Figure 47).

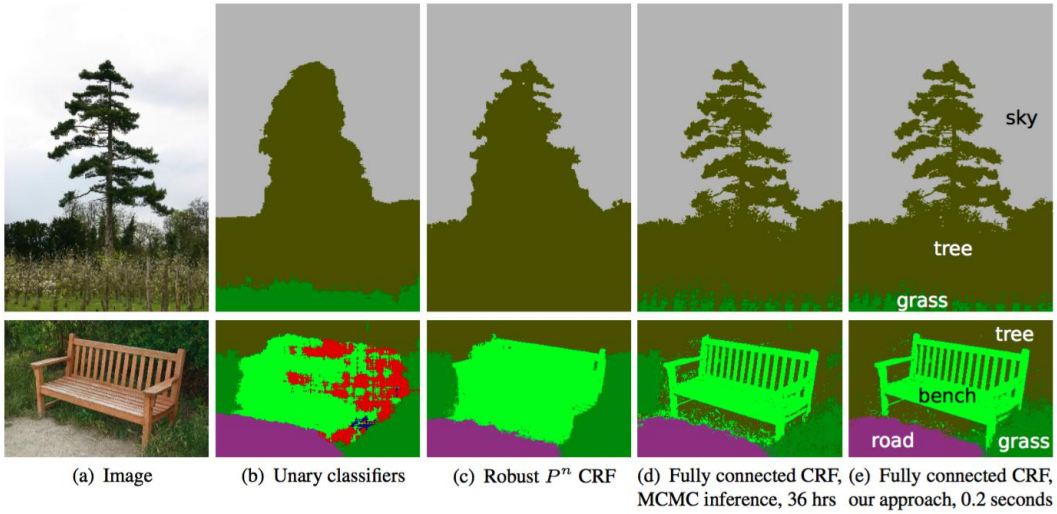


Figure 47 CRF illustration. (b) Unary classifiers is the segmentation input to the CRF. (c, d, e) are variants of CRF with (e) being the widely used one [66].

Key contributions and Benchmarks:

- Uses atrous/dilated convolutions.
- Proposes ASPP.
- Uses Fully Connected CRF.

Table 26 DeepLabv2 Benchmarks on VOC2012 and Cityscapes.

Architecture	**Score on VOC2012** (mean IoU)	**Score on Cityscapes** (mean IoU)
DeepLabv2	79.7	70.4

A.6. RefineNet: Multi-Path Refinement Networks for High-Resolution Semantic Segmentation

Submitted on 20 Nov 2016

Summary:

Even though very deep CNNs have been the first choice for dense classification problems, such as semantic segmentation, the repeated subsampling operations that take place in them (like pooling or strided convolutions) lead to a significant decrease in the initial image resolution. RefineNet [28] exploits all the information available along the down-sampling process to enable high-resolution prediction using long-range residual connections.

Key features:

In conventional ConvNets, the use of dilated/atrous convolutions to approach the problem of spatial reduction is computationally very expensive. Moreover, it takes a lot of memory due to the fact that they have to be applied on large number of high resolution feature maps. This does not allow the computation of high-res predictions (DeepLabv2's predictions [27], for example, are 1/8th of the size of original input).

To overcome this problem, RefineNet proposes the use of an special enconder-decoder architecture. Encoder part is formed by ResNet-101 [21] blocks; the decoder has special RefineNet blocks which allow to iteratively pool features for several ranges of resolutions. Finally, a high-resolution segmentation map is produced.

Key contributions and Benchmarks:

- Encoder-Decoder architecture with well thought-out decoder blocks.
- Introduces CRP (Chained Residual Pooling), where all the components follow residual connection.

Table 27 RefineNet Benchmarks on VOC2012 and Cityscapes.

Architecture	**Score on VOC2012** (mean IoU)	**Score on Cityscapes** (mean IoU)
RefineNet	84.2	73.6

A.7. PSPNet: Pyramid Scene Parsing Network

Submitted on 4 Dec 2016

Summary:

The PSPNet (Pyramid Scene Parsing Network) [33] exploits the capability of global context information through the use of a pyramid pooling module. The proposed approach achieved state-of-the-art performance on various datasets, such as VOC2012 and Cityscapes, and still is one of the best architectures in both these benchmarks. [33]

Key features:

Global scene categories matter because they provide clues on the distribution of the segmentation classes. The pyramid pooling module that the PSPNet employs captures this information by applying large kernel pooling layers onto the input image.

The dilated convolutions that Yu introduced in his work [26] are also used in the PSPNet in order to modify ResNet [21]. The pyramid pooling module concatenates the feature maps from the modified ResNet model with upsampled output of parallel pooling layers . This is accomplished by using kernels that cover whole, half and small portions of the input image.

Additionally, an auxiliary loss, complementary to the loss on the main branch, is applied after the fourth stage of ResNet [21] (i.e input to pyramid pooling module).

Key contributions and Benchmarks:

- Proposes pyramid pooling module to aggregate the context.
- Uses auxiliary loss.

Table 28 PSPNet Benchmarks on VOC2012 and Cityscapes.

Architecture	**Score on VOC2012** (mean IoU)	**Score on Cityscapes** (mean IoU)
PSPNet	85.4	81.2

A.8. DeepLab (v3): Rethinking Atrous Convolution for Semantic Image Segmentation

Submitted on 17 Jun 2017

Summary:

The DeepLabv3 [32] network revisits atrous convolution, a tool to explicitly adjust filter's field-of-view as well as control the resolution of feature responses. To handle the problem of segmenting objects at multiple scales, DeepLabv3 proposes modules which employ atrous convolution in cascade or in parallel to capture multi-scale context by adopting multiple atrous rates.

Furthermore, it proposes to augment the previously proposed ASPP module from the DeepLabv3 [27] network, which probes convolutional features at multiple scales, with image-level features encoding global context.

The proposed DeepLabv3 network significantly improves over the previous DeepLab versions without DenseCRF post-processing and attains comparable performance with other state-of-art models on the PASCAL VOC 2012 semantic image segmentation benchmark.

Key features:

ResNet [21] model (encoder part) is again modified to use dilated/atrous convolutions as in DeepLabv2 [27]. Improved ASPP involves concatenation of image-level features, a 1-by-1 convolution and three 3x3 atrous convolutions with different rates. Besides, batch normalization is used after each of the parallel convolutional layers. Authors note that the improvement in DeepLabv3 comes from this batch normalization and from the better way to encode multi scale context.

Key contributions and Benchmarks:

- Introduces a module which employs atrous convolutions in cascade.
- Improved the ASPP module, already introduced in DeepLabv2 [27].

Table 29 DeepLabv3 Benchmarks on VOC2012 and Cityscapes.

Architecture	**Score on VOC2012** (mean IoU)	**Score on Cityscapes** (mean IoU)
DeepLabv3	85.7	81.3

B. Appendix. Freezing and Optimazing a Deep Learning Model for Inference in TensorFlow

B.1. Freezing a Graph in TensorFlow

Freezing a graph in the context of DL and TensorFlow is the process of converting TensorFlow variables into constants. This is done due to the fact that, during inference, the values stored by these variables do not change, and by having only constants the computer can work much faster. Additionally, we obtain the following benefits when freezing a graph:

- We remove unneccessary nodes related to training.
- The model can be entirely compressed into one single protobuf [72] file.
- The graph structure becomes simpler.
- Since everything resides in one file, it is easier to deploy.

B.1.1. Before freezing the graph

When saving a graph in TensorFlow, we obtain the following files. Note that some files are under a folder named 'variables' simply because we decided to do so when saving the model.

- base_graph.pb
- variables/checkpoint
- variables/base_graph.data-00000-of-00001
- variables/base_graph.index
- variables/base_graph.meta

The file *base_graph.pb* is a protobuf representation of the graph and it does not include weight values, simply the graph's structure. The rest of the files are all related to the saved weights and associated metadata.

B.1.2. Freezing the graph

TensorFlow provides a Python script named *freeze_graph.py* which outputs a frozen graph. When calling it, we need to provide five inputs:

- The input graph, **input_graph** (*e.g.* 'base_graph.pb')
- The input graph checkpoint, **input_checkpoint** (*e.g.* 'checkpoint')
- **input_binary**: must be set to *True* when the input graph is a binary file (*i.e.* a '.pb' file)
- The desired name of the output graph, **output_graph** (*e.g.* 'frozen_graph.pb')
- The name of the output nodes. Naming key nodes in the graph can come in handy.

In our case, when freezing our semantic segmentation base graphs, we used the following bash command to call the aforementioned Python script:

```
TUM_Roborace $ python freeze_graph.py --input_graph=base_graph.pb
   --input_checkpoint=variables/checkpoint --input_binary=True
   --output_graph=frozen_graph.pb --output_node_name=logits
```

This command should generate the desiderd 'frozen_graph.pb' file, where many extraneous operations have been removed from the initial graph.

B.2. Optimizing the Graph for Inference in TensorFlow

Now that we have a frozen graph, we can perform different transformations. TensorFlow has a tool (a Python script) called *optimize_for_inference.py* which:

- Removes training-specific and debug-specific nodes
- Fuses common operations
- Removes entire sections of the graph that are never reached

Again, will have to provide several parameters:

- The previously frozen graph will be our input graph, **input** (*e.g.* 'frozen_graph.pb')
- The desired name of the output graph, **output** (*e.g.* 'optimized_graph.pb')
- Since the *optimize_for_inference* tool works for both frozen and unfrozen graphs, we need to specify whether our input graph is already frozen. We do that by setting the **frozen_graph** flag to 'true'
- *input_names* and **output_names** are the names of the input and output nodes respectively.

```
TUM_Roborace $ python optimize_for_inference.py --input=
    frozen_graph.pb --output=optimized_graph.pb --frozen_graph=
    true --input_names=input_image --output_names=logits
```

C. Appendix. Theory behind Stereo Vision

In autonomous driving, reconstructing a three dimensional scene with a camera is indeed an relevant topic, even critical if we wanted to dispose of LIDARs or radars. To accomplish this task, multiple images from the scene are needed. The easiest and most obvious case in which multiple images are used to reconstruct a 3D scene is stereo vision.

In stereo vision, features in two (or more) images taken at the same time from separate cameras are matched with the corresponding features in the other images. Any differences found (disparity effects) are then studied to obtain depth information by applying triangulation.

The other case in which we can use multiple images for 3D reconstruction is structure from motion. In this case, which will not be covered in this work, multiple images are taken at different times and from different places, with the objective of computing the so called fundamental matrix, which relates two different view together.

In any of the previous two cases, we need calibrated cameras. But even before that, we need a mathematical model to represent our sensor. In subsection C.1 we will cover the basics of camera models and calibration. After that, in subsection C.2, we will dive into stereo imaging, covering all its relevant steps and elements. After covering these two subsections, we will have presented what is the basic knowledge and tools one needs to gather about stereo vision in order to be able to tackle more advanced depth-related approaches.

All information presented in this appendix has been extracted to a greater or lesser extent from the book written by Gary Bradski and Adrian Kaehler ("Learning OpenCV") [67], more precisely from Chapters 11 and 12. An attempt has been made to synthesize these two dense chapters as much as possible, presenting here only what was considered relevant information for the development of this work.

C.1. Camera Models and Calibration

In this subsection we will cover the easiest camera model: the pinhole model. We will also learn how it is possible to correct for the main deviations from this simple pinhole model that the use of lenses creates. This is accomplished by calibratig the camera, which not only helps correcting these deviations but also is key in the process of relating camera measurements with distances in the real 3D world.

C.1.1. Camera Model

The simplest model of a camera is the pinhole camera model, where from any particular point of a scene only a single ray of light enters the camera. In a physical pinhole camera, this point is then "projected" onto an imaging surface, known as the image plane, thus obtaining a focused image. The size of the image relative to a distant object is given by a single parameter of the camera: its

focal length. In the pinhole model, the distance from the pinhole aperture to the screen is what we define as focal length. We can see this in Figure 48, where f is the focal length of the camera, Z is the distance from the camera (more precisely from the pinhole aperture) to the object, X is the lenght of the object, and x is the object's image on the imaging plane. By similar triangles, we can deduce that $-x/f = X/Z$, or:

$$-x = f\frac{X}{Z} \tag{C.1}$$

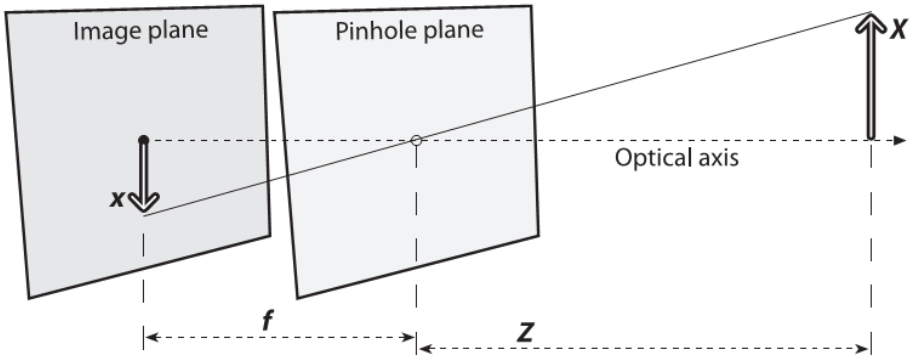


Figure 48 Pinhole camera model: a pinhole (the pinhole aperture) lets through only those light rays that intersect a particular point in space; these rays then form an image by "projecting" onto an image plane [67].

In order to make the math come out easier, we can perform the mathematical abstraction of swaping the pinhole and the image plane, obtaining now the object rightside up (Figure 49). The point in the pinhole is reinterpreted as the *center of projection*. Moreover, we also define the intersection between the image plane and the *optical axis* as the *principal point*.

In this new organization, every ray leaves a point on the distant object and heads for the center of projection. The image is generated by intersecting these rays with the image plane, which is exactly a distance f from the center of projection. From the similar triangles relationship, we obtain $x/f = X/Z$, where the negative sign is gone due to the fact that the object is no longer upside down (again, Figure 49).

One false assumption is to think that the principle point (intersection of optical axis with the image plane) is equivalent to the center of the imager (or image plane). For this to be true, we would need micron accuracy when attaching the imager to the camera. In reality, the center of the chip is usually not on the optical axis, what introduces two new parameters, c_x and c_y to model this possible displacement of the center of coordinates on the projection screen away from the optical axis. All this results in a model in which a point Q in the physical world, with coordinates (X, Y, Z), is projected onto the screen at some pixel location given by (x, y), as can be observed in Figure 49. Another possible (and maybe clearer) notation denotes the projection of the point on the imager as (u, v), as shown in Figure 50.

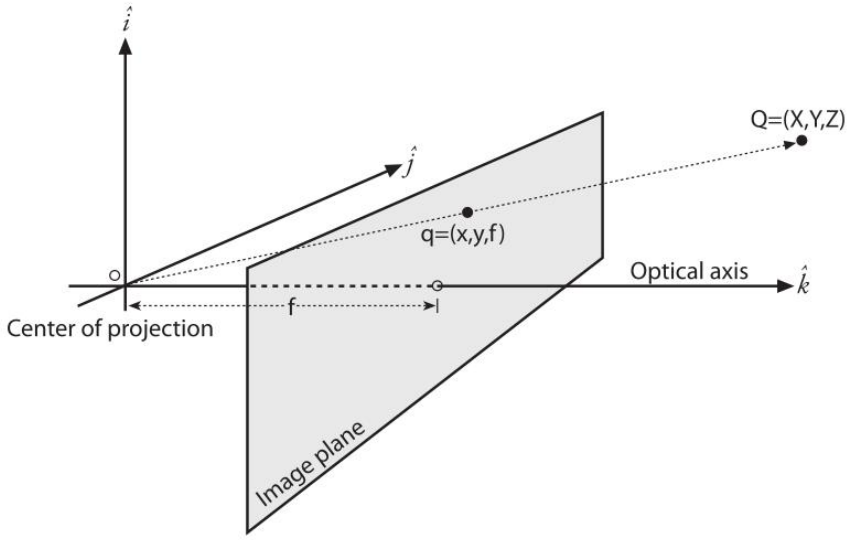


Figure 49 Restructured pinhole camera model: A point $Q = (X, Y, Z)$ is projected onto the image plane by the ray passing through the center of projection, and the resulting point on the image is $q = (z, y, f)$; the image plane is really just the projection screen "pushed" in front of the pinhole (the math is equivalent but simpler this way) [67].

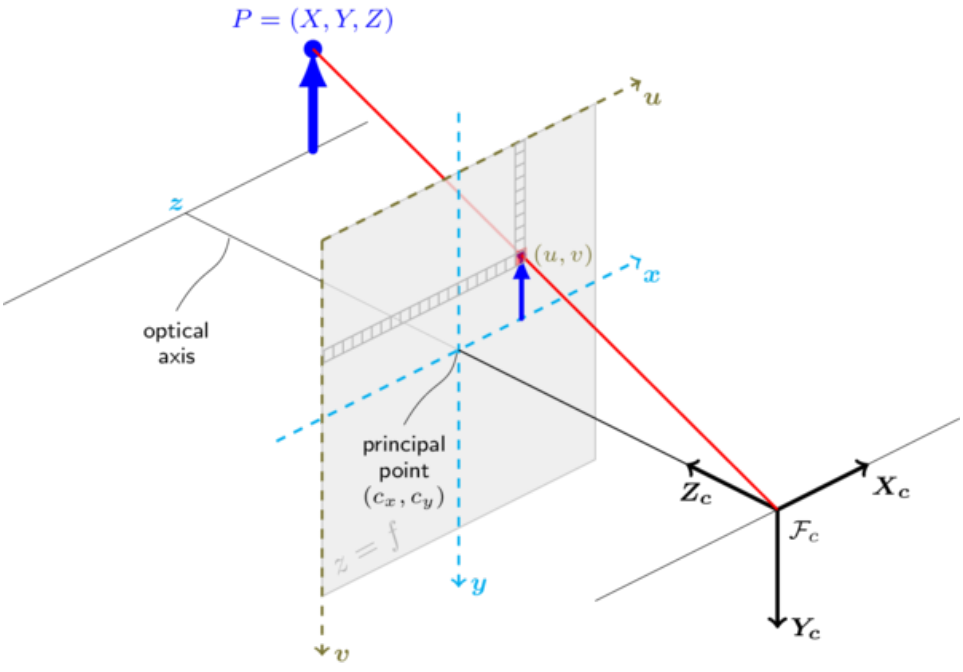


Figure 50 A point $P = (X, Y, Z)$ is projected onto the image plane by the ray passing through the center of projection (X_c, Y_c, Z_c), and the resulting point on the imager is (u, v) [68].

Using the second notation, we can obtain Equation C.2, where we have also introduced two focal lengths, the reason being that the individual pixels on a standard imager are rectangular, not square. Thus the focal length f_x is the product of the physical focal length of the lens and the size s_x of the individual imager elements, and correspondigly for f_y.

$$u = f_x \left(\frac{X}{Z}\right) + c_x, \quad v = f_y \left(\frac{Y}{Z}\right) + c_y \tag{C.2}$$

We can express this relationship between the 3D real world points and their corresponding projections in the imager by introducing the *camera intrinsic matrix, M*, which arranges the parameters that define our camera (f_x, f_y, c_x, c_y). The projection of the points in the physical world into the camera can then be summarized by Equation C.3.

$$q = MQ, \quad where \quad q = \begin{bmatrix} x \\ y \\ w \end{bmatrix}, \quad M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \quad Q = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{C.3}$$

Note that we are working with *homogeneous coordinates*. Hence, after multiplying this out and finding that $w = Z$, we will need to divide the elements of q (that is, x, y, w by w (or Z) in order to recover the definitions from C.2.

Up to this point, we have fully described the pinhole model. Now we need to introduce lenses, which help us gather a lot of light over a wider area and bend that light to converge at the point of projection. This will introduce distorsions: radial distorsions (Figure 51), which arise as a result of the shape of the lens, and tangential distorsions (Figure 52), which arise from the assembly process of the camera.

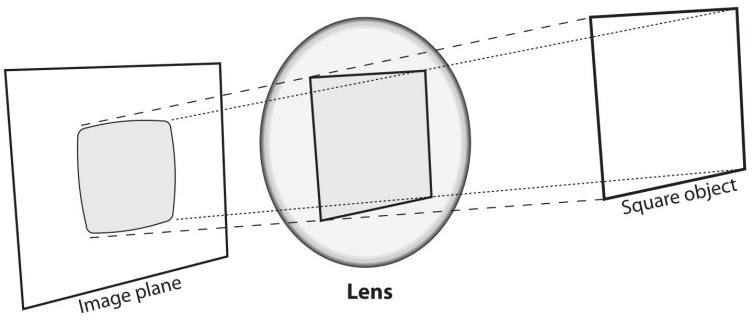


Figure 51 Radial distorsion: rays further from the center of a simple lens are bent too much compared to rays that pass closer to the center; thus, the sides of a square appear to bow out on the image plane[67].

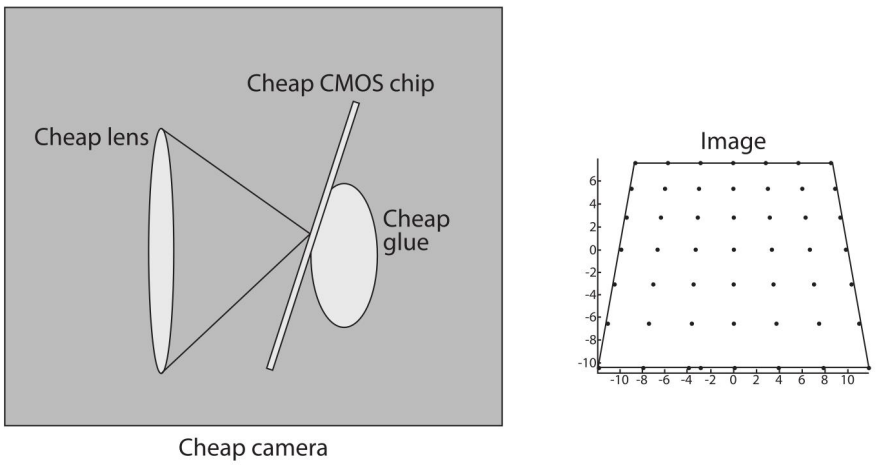


Figure 52 Tangential distorsion: it exists when the lens is not fully parallel to the image plane; in cheap cameras, this can happen when the imager is glued to the back of the camera [67].

In total there are five distorsion coefficients (k_1, k_2, k_3 for radial distorsion and p_1, p_1 for tangential distorsion). They are typically bundled into one distorsion vector, containing the 5 coefficients in the order they have been mentioned here. Many of the operations in OpenCV require that we provide this distorsion vector.

C.1.2. Calibration

The process of camera calibration allows us to obtain the intrinsic and distorsion properties of a specific camera. This process is based on targetting the camera on a known structure that has many individual and identifiable points. By viewing this structure from different angles and distances, we can compute the relative location and orientation of the camera at the time of each image, as well as the intrinsic parameters of the camera.

C.1.3. Undistorsion

There are usually two things that one may want to do with a calibrated camera. The first, to correct for distortion effects; the second, to construct a 3D representation of the images the camera receives, which we will cover in subsection C.2.

With respect to undistorting an image, we can say that it is basically taking the raw image and the distorsion coefficients of the camera (which we should already have after having calibrated the camera) and then producing a corrected image. The method behind this process is to compute a distortion map, which is then used to correct any new incoming image.

Figure 53 Undistorsion example: camera image before undistorsion (left) and after undistorsion (right) [67].

C.2. Stereo Imaging

Now that we have some basic understanding of how one single camera works, and only after having calibrated it, it is possible to unambiguously project points in the physical world to points in the image. In other words, given a location in the 3D physical coordinate frame attached to the camera, we can compute where on the imager (in pixel coordinates) this external 3D point is. And we can also do the inverse operation: finding the 3D coordinates of any pixel on the imager.

The task of stereo imaging is based on finding correspondences between point that are seen by one imager and the same points as seen by the other imager. These correspondences plus the knowledge of the baseline separation between cameras allow us to compute the 3D location of the 2D points. More precisely, we can divide the process of stereo imaging in 4 steps:

- **Undistortion**: mathematically remove radial and tangential lens distorsion, thus obtaining undistorted images.
- **Rectification**: adjust for angles and distances between cameras, obtaining as an output "row-aligned" and rectified images. By "row-aligned" we understand that the two images planes are coplanar and that the image rows are exactly aligned, *i.e.*, having the same y-coordinate.
- **Correspondence**: find the same features in the left and right camera views. From this step we obtain a disparity map, *i.e.* a map of the differences in x-coordinates on the image planes of the same feature viewed in the left and right cameras: $x^l - x^r$.
- **Reprojection**: if the geometric disposition of the cameras than conform the stereo rig is known, then we can translate the disparity map into distances by triangulation, thus obtaining a depth map.

We will start with the last step to motivate the first three.

C.2.1. Triangulation

We will assume that our stereo rig is perfetly undistorted, aligned and measured, as in Figure 54. That is, we have two cameras whose image planes are exactly coplanar with each other, that are a known distance apart, with equal focal lengths $f_l = f_r$ and with exactly parallel optical axes (recall the optical axis, or principal ray, is the ray that goes from the center of projection O through the princial point c). We will also assume that the principal points c_x^{left} and c_x^{right} have been calibrated to have the same pixel coordinates in their respective left and right images.

Recall that a principal point does not necessarily coincide with the center of the image. The principal point is where the principal ray intersects the imaging plane, and this intersection depends on the optical axis of the lens. Because the image plane is rarely aligned with the lens, the center of the image is almost never exactly aligned with the principal point.

Let's further assume the images from our stereo rig are row-aligned and that every pixel row of one camera aligns exactly with the corresponding row in the other camera. Such a camera arrangement is known as frontal parallel. Last assumption will be that we can find a point P in the physical world in the left and the right image views at p_l and p_r, which will have the respective horizontal coordinates x_l and r_r.

In this simplified structure (Figure 54), taking x_l and r_r to be the horizontal positions of the points in the left and right imager (respectively) allows us to show that the depth is inversely proportional to the disparity between these views, understanding disparity as $d = x_l - x_r$. The equation we can derive is as follows:

$$\frac{T - (x_l - x_r)}{Z - f} = \frac{T}{Z} \Rightarrow Z = \frac{fT}{x_l - x_r} \tag{C.4}$$

This inversely proportional relationship between depth and disparity creates obviously a nonlinear relationship between both terms (Figure 55). When disparity is near 0, small disparity differences produce large depth differences, and viceverse. Therefore, we conclude than in stereo vision systems one can only achieve high depth resolution for nearby objects.

We need now to make a comment on coordinate systems. Figure 56 shows the coordinate systems that are usually used in Stereo Vision. Note that the left and right imager pixels have image origins at the upper left corner in the image, with pixels being denoted by coordinates (x_l, y_l) and (x_r, y_r) for each camera respectively. Moreover, the center of projection of each camera is at O_l and O_r, respectively, with principal rays intersecting the image plane at the principal point (not the center of the image) (c_x, c_y) for both cameras. After mathematical rectification, the cameras would be row-aligned (coplanar and horizontally aligned) and displaced from one another by T. Recall also that we assumed they would have same focal length f.

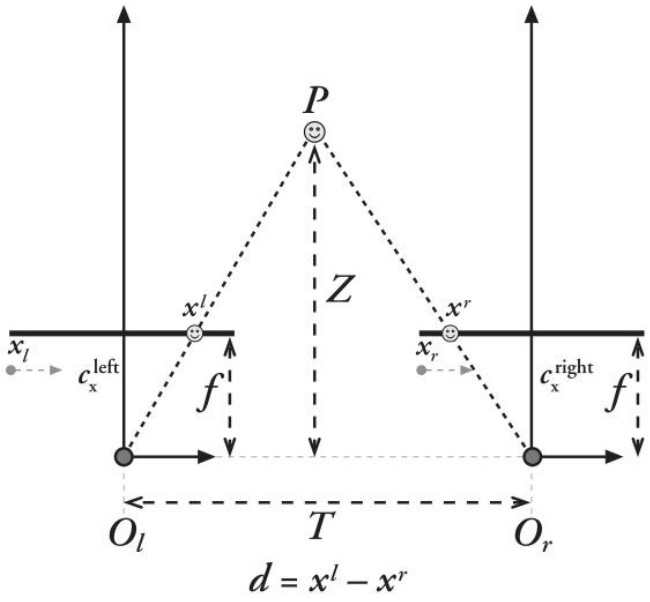


Figure 54 Frontal parallel stereo rig: with a perfectly undistorted, aligned stereo rig and known correspondence, the depth Z of a point P can be found by similar triangles; note that in this disposition the principal rays of the imagers begin at the centers of projection O_l and O_r and extend through the principal points of the two image planes at c_l and c_r [67].

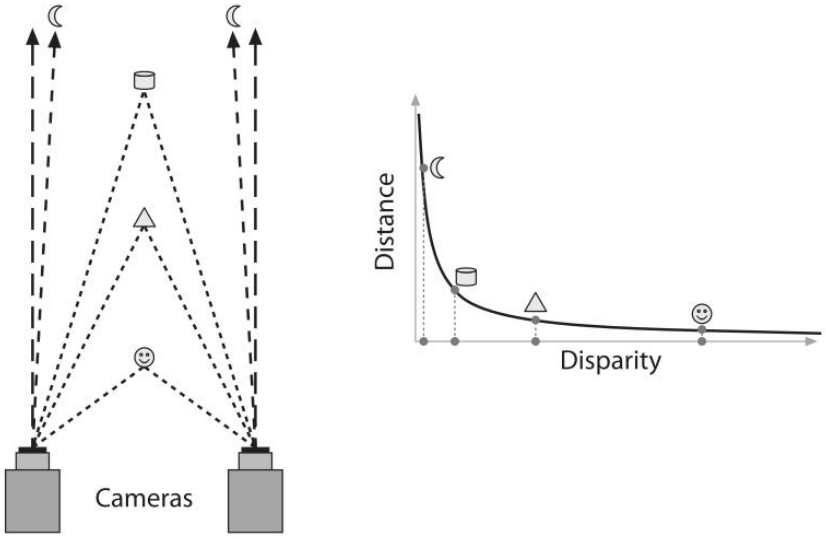


Figure 55 Inverse relationship between depth and disparity: depth and disparity are inversely related, so high depth resolution is only possible for nearby objects [67].

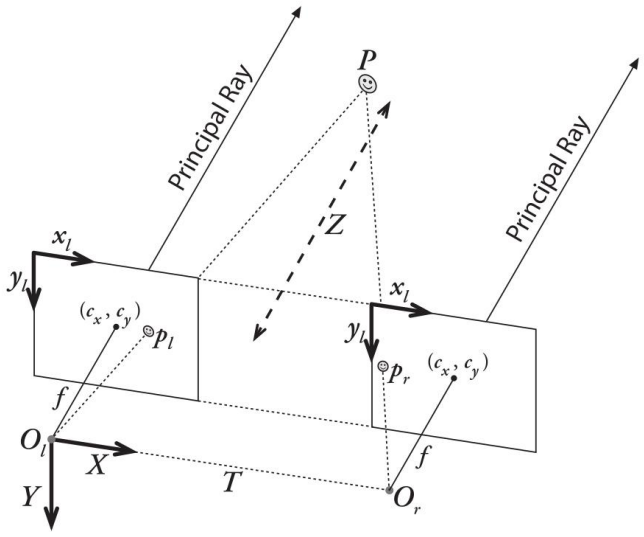


Figure 56 Stereo coordinate systems used by OpenCv for undistorted rectified cameras: the pixel coordinates are relative to the upper left corner of the image, and the two planes are row-aligned; the camera coordinates are relative to the left camera's center of projection [67].

The problem is that in a real-world camera setup, cameras will almost never be exactly aligned in the frontal parallel configuration we saw in Figure 54. So we ned to mathematically find image projections and distortion maps that will rectify the left and right images into a frontal parallel arrangement. Still, it would be best practice to always try to design your stereo rig following the frontal parallel configuration, where the cameras are as close to horizontally aligned as possible. It is also extremely important to synchronize the cameras, meaning that they must capture image pairs at the exact same time. Not doing this will yield horrible results either if something is moving in the scene or if the cameras themselves are moving.

Figure 57 shows the real situation between two cameras we might encounter and the mathematical alignement we want to achieve. Before performing this mathematical alignment, we need to understand the geometry of two cameras viewing a scene.

C.2.2. Epipolar Geometry

The basic geometry of a stereo imaging system is referred to as epipolar geometry. Figure 58 concisely summarizes the key terminology of epipolar geometry. The new concept we introduce now are the epipoles (e_l and e_r), situated on their corresponding image planes (Π_l and Π_r) and defined as the image of the center of projection of the other camera O_r or O_r, respectively. The plane in space formed by the actual viewed point P and the two epipoles e_l and e_r is called the *epipolar plane*, and the lines $p_l e_l$ and $p_r e_r$ are known as the *epipolar lines*.

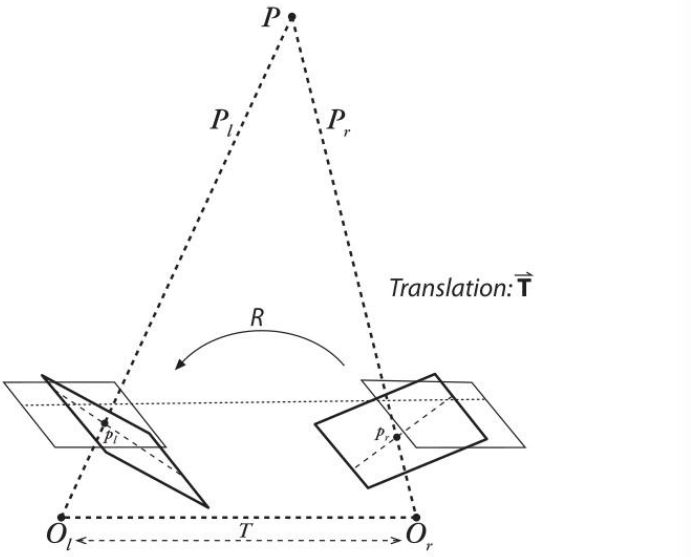


Figure 57 Mathematical aligment of the stereo rig: the goal is to mathematically (rather than physically) align the two cameras into one viewing plane, so that pixel rows between the cameras are exactly aligned with each other [67].

Basically, the utility behind epiplar geometry is that when we see a point in the real world (P in Figure 58) projected onto our right (for example) image plane (p_r), that point could be anywhere along an entire line of points formed by the ray going from O_r out through p_r, because with one single camera we cannot know the distance to the point in the 3D world. What we can do is to ask what that line looks like projected onto the left image plane π_l; in fact, it is the *epipolar line* defined by p_l and e_l.

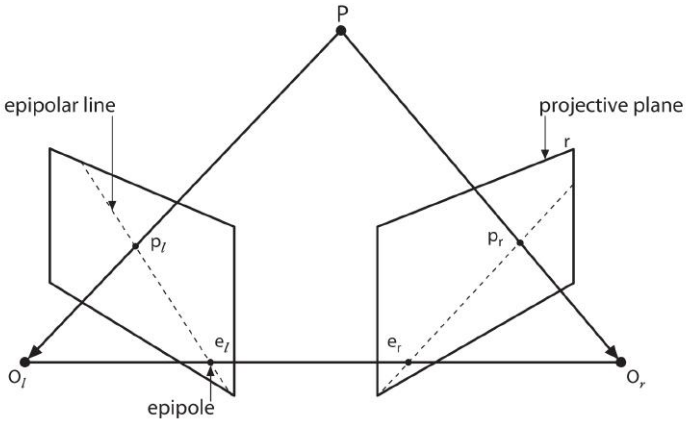


Figure 58 Epipolar geometry: the epiplar plane is defined by the observed point P and the two centers of projection, O_l and O_r. The epipoles are located at the point of intersection of the line joining the centers of projection and the two projective planes [67].

In other words, the image of all the possible locations of a point seen in one imager is the *line* that goes through the corresponding point and the epipolar point on the other imager. This is known as the *epipolar constraint*, which means that the possible two-dimensional search for matching features across two imagers becomes a one-dimensional search along the epipolar lines once we know the epipolar geometry of the stereo rig.

C.2.3. The Essential and Fundamental Matrices

For being able to compute the epipolar lines, we first need the *essential matrix* E and the *fundamental matrix* F. The matrix E is purely geometrical (knows nothing about the imagers) and contains information about the translation and rotation that relate the two cameras in the physical space (Figure 59). It relates the location, in physical coordinates, of the point P as seen by the left camera to the location of the same point as seen by the right one (*i.e.* it relates p_l and p_r). Matrix F contains the same information as E in addition to information about the intrinsics of both cameras. Since F embeds information about the intrinsic parameters, it relates the two cameras in pixel coordinates (*i.e.* F relates the points on the image plane of one camera to the points on the image plane of the other camera in image coordinates(pixels)).

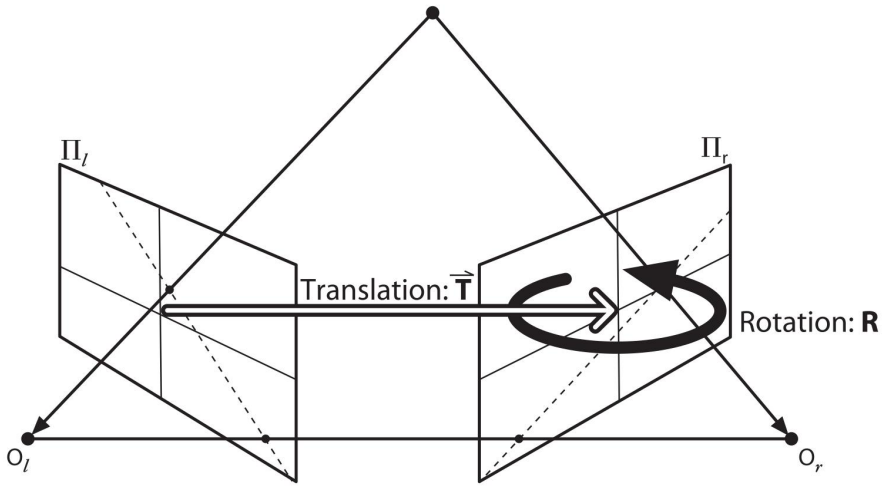


Figure 59 The essential matrix E: the essential geometry of stereo imaging is captured by the essential matrix E, which contains all of the information about the translation T and the rotation R that describe the location of the second camera relative to the first in global coordinates [67].

The math necessary for computing E and F is not complicated, but OpenCV handles it for us completely, so for the sake of simplicity, we will not dive into it in this work. What we need to know is that, once we have the fundamental matrix F, we can compute the epipolar lines in one image that correspond to a list of points in the other image.

C.2.4. Stereo Calibration

Stereo calibration is the process of computing the geometrical relationship between the two cameras in space. Basically, we seek a single rotation matrix and translation vector that relate the right camera to the left camera. We can achieve this by computing the single-camera calibration for each camera in order to put P in the camera coordinates $P_l = R_l P + T_l$ and $P_r = R_r P + T_r$, for the left and right cameras, respectively. On the other hand, from Figure 59 we can get that $P_l) = R^T(P_r - T)$, where R and T are, respectively, the rotation matrix and translation vector between the cameras. In this 3-equation system we can solve for R and T, obtaining:

$$R = R_r(R_l)^T \tag{C.5}$$
$$T = T_r - RT_l \tag{C.6}$$

With many joint views of chessboard corners, we can solve for rotation and translation parameters of the chessboard views for each camera separately. Then, plug these values into equations C.6 and compute the rotation and translation parameters between the two cameras. Due to noise and rounding erros, each chessboard pair will result in slightly different values for R and T, so OpenCV will take the median values for each parameter as the initial approximation of the true solution and then run an iterative algorithm to find the (local) minimum of the reprojection error of the chessboard corners for both camera views.

In summary, the stereo calibration will output the rotation matrix that puts the right camera in the same plane as the left one. This will make the two image planes coplanar but not row-aligned (this will be accomplished in the Stereo Rectification section (C.2.5)).

C.2.5. Stereo Rectification

Stereo rectification is the process of "correcting" the individual images so that they appear as if they had been taken by two cameras with row-aligned image planes (as in Figure 54). With such a rectification, the optical axes (or principal rays) of the two cameras are parallel and so we say that they intersect at infinity. The algorithm that is usually applied when one has the rotation matrix R and translation vector T is the *Bouguet's algorithm* for stereo rectification. It simply tries to minimize the amount of change reprojection produces for each of the two images, and thereby minimize the resulting reprojection distortions, while maximizing common viewing area.

Once we have our stereo calibration terms, we can pre-compute left and right rectification lookup maps for the left and right camera views.

The whole process of rectification is illustrated in Figure 60. Note that the actual rectification process works backwards from (c) to (a) in a process known as *reverse mapping*. For each integer pixel in the rectified image (c), we find its coordinates in the undistorted image (b) and use those to look up the actual (floating-point) coordinates in the raw image (a). The floating-point coordinate pixel is then interpolated from the nearby integer pixel locations in the original source imgage, and that value is used to fill in the rectified integer pixel location in the destination image (c). Finally, the rectified image is cropped to just keep the overlapping areas between left and right images.

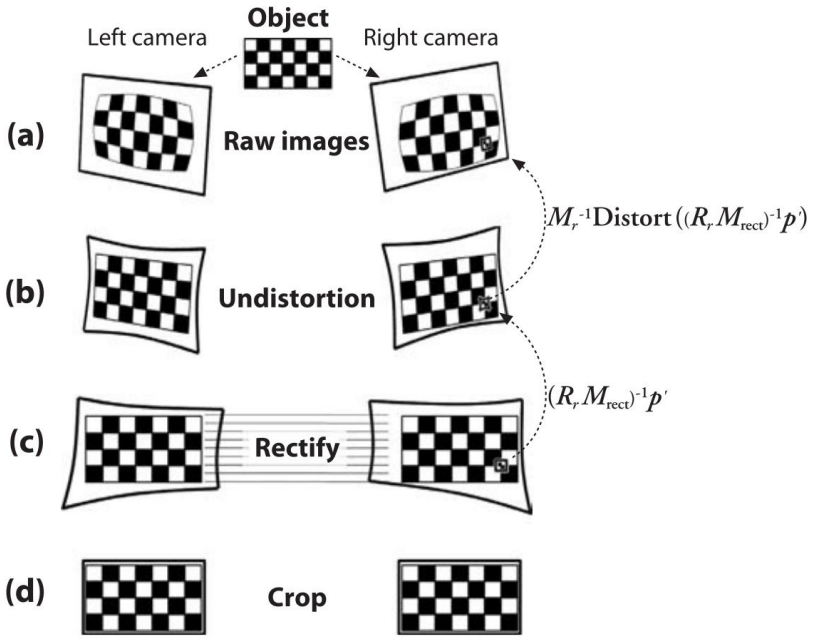


Figure 60 Stereo rectification: for the left and right camera, the raw image (a) is undistorted (b) and rectified (c), and finally cropped (d) to focus on overlapping areas between the two cameras; the computation actually works backwards from (c) to (a) [67].

Figure 61 shows an example of how a stereo pair must be undistorted and rectified. Note how feature points become horizontally aligned in the undistorted rectified images.

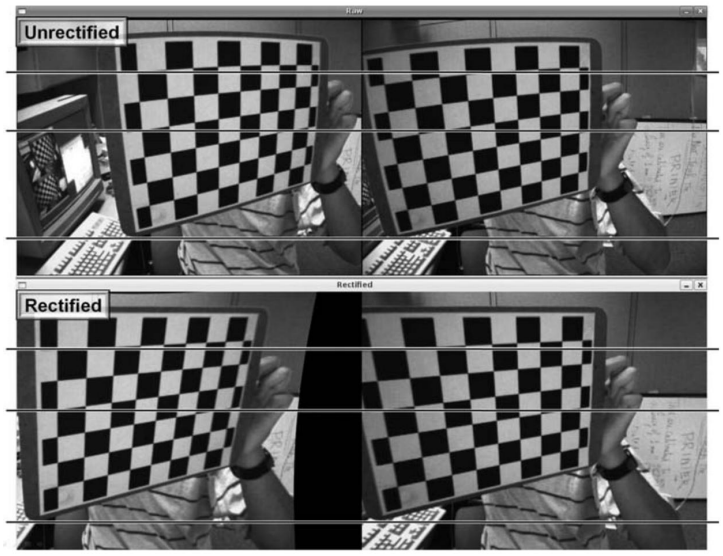


Figure 61 Stereo rectification example: upper pannels correspond to the unrectified pair; lower pannels, to the rectified pair [67].

C.2.6. Stereo Correspondance

Stereo correspondance - matching a 3D point in the two different camera views - can only be computed over the visual areas in which the views of the two cameras overlap. So, once we know the physical coordinates of the cameras or the sizes of objectes in the scene, we can derive depth measurements from the triangulated disparity measures $d = x^l - x^r$ between the corresponding points in the two different camera views.

OpenCV implements a fast and effective block-matching stereo algorithm that is similar to the one developed by Kurt Konolige [73]. This algorithm finds only strongly matching (high-texture) points between the two images. This means that in a highly textured scene, such as a forest, every pixel might have computed depth. However, in a very low-textured scene, such as an indoor hallway, very few points might register depth.

C.2.7. Depth Maps from 3D Reprojection

At this point, we finally have a disparity map. Many algorithms will just use this disparity map directly (for example, to detect whether or not objects are on a table). But for 3D shape matching, 3D model learning, robot grasping and so on, we need the actual 3D reconstruction or depth map. This can be easily achieved by using the so called *reprojection matrix* Q and the equation C.7, where the 3D coordinates are then $(X/W, Y/W, Z/W)$.

$$Q \begin{bmatrix} u \\ v \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix}, \quad where \quad Q = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 0 & f \\ 0 & 0 & 1 & (c_x - c'_x)/T_x \end{bmatrix} \tag{C.7}$$

Note that in Q all parameters are from the left image, except for c'_x, which is the principal point x coordinate in the right image. If the principal rays intersect at infinity, then $c_x = c'_x$ and the term in the lower right corner would be 0.