HOCHSCHULE DER MEDIEN

Fully Convolutional Networks for Semantic Segmentation from RGB-D images

Master Thesis
Computer Science and Media

Submitted by:
Nicolai Harich
Mat.-Nr. 29308

First Reviewer: Prof. Dr. Johannes Maucher (Hochschule der Medien)
Second Reviewer: Dipl. -Ing. Matthias Roland (Bosch Start-Up GmbH)

March 2016

Title: Fully Convolutional Networks for Semantic Segmentation
from RGB-D images

Abstract: In recent years new trends such as industry 4.0 boosted the research and development in the field of autonomous systems and robotics. Robots collaborate and even take over complete tasks of humans. But the high degree of automation requires high reliability even in complex and changing environments. Those challenging conditions make it hard to rely on static models of the real world. In addition to adaptable maps, mobile robots require a local and current understanding of the scene. The Bosch Start-Up Company is developing robots for intra-logistic systems, which could highly benefit from such a detailed scene understanding. The aim of this work is to research and develop such a system for warehouse environments. While the possible field of application is in general very broad, this work will focus on the detection and localization of warehouse specific objects such as palettes.

In order to provide a meaningful perception of the surrounding a RGB-D camera is used. A pre-trained convolutional network extracts scene understanding in the form of pixelwise class labels. As this convolutional network is the core of the application, this work focuses on different network set-ups and learning strategies. One difficulty was the lack of annotated training data. Since the creation of densely labeled images is a very time consuming process it was important to elaborate on good alternatives. One interesting finding was that it's possible to transfer learning to a high extent from similar models pre-trained on thousands of RGB-images. This is done by selective interventions on the net parameters. By ensuring a good initialization it's possible to train towards a well performing model within few iterations. In this way it's possible to train even branched nets at once. This can also be achieved by including certain normalization steps. Another important aspect was to find a suitable way to incorporate depth-information. How to fuse depth into the existing model? By providing the height over ground as an additional feature the segmentation accuracy was further improved while keeping the extra computational costs low.

Finally the segmentation maps are refined by a conditional random field. The joint training of both parts results in accurate object segmentations comparable to recently published state-of-the-art models.

Keywords: Semantic Segmentation, Deep Learning, Convolutional Neural Networks, Fully Convolutional Neural Networks, multimodal, RGB-D, FCN, CNN, CRF
Date: March 2016

Titel: Neuronale Netze zur semantischen Bildsegmentierung von RGB-D Bildern

Abstract: Aktuelle Themen, wie zum Beispiel Industrie 4.0, haben Fortschritte im Bereich autonomer Systeme und Robotik vorangetrieben. Roboter kollaborieren mit Arbeitern oder übernehmen komplette Arbeitsschritte. Dieser hohe Automatisierungsgrad erfordert, dass solche Systeme, selbst in komplexen Situationen und Umgebungen, hochgradig zuverlässig und sicher arbeiten. Statische Modelle zur Abstrahierung der Umgebung sind unzureichend. Mobile Roboter benötigen neben dynamischen Lokalisierungskarten bestenfalls auch ein Verständnis der Umgebung. Die Bosch Start-Up GmbH entwickelt Roboter, welche zukünftig in Warenlagern eingesetzt werden sollen. Diese würden von einem solchen Verständnis profitieren. Das Ziel war es aktuelle Erkenntnisse aus der Forschung zur semantischen Segmentierung mithilfe von Deep Learning Techniken zu einer prototypischen Anwendung zu transferieren. Die entwickelte Anwendung im Allgemeinen zwar universell einsetzbar, der Fokus dieser Arbeit liegt jedoch auf der Segmentierung von Objekten aus einem typischen Warenlager (bspw. Paletten).

Die Segmentierung basiert auf den Bildern einer RGB-D Kamera und ermöglicht gleichzeitig eine räumliche Lokalisierung von Objekten. Ein spezielles tiefes neuronales Netz (FCN) führt die komplette Segmentierung durch. Die Arbeit beschäftigt sich schwerpunktmäßig mit der Adaption und dem Training eines solches Netzes. Die Bereitstellung von annotatierten Daten ist äußerst aufwändig. Um die Zahl der nötigen Daten gering zu halten wurden geeignete Techniken eingesetzt. Dazu wurden Modellparameter frei zugänglicher Netze transferiert, um eine möglichst gute Initialisierung sicherzustellen. Außerdem wurden Normalisierungsschritte in die Netzarchitektur eingeführt, sodass auch verzweigte Strukturen in einem Trainingslauf trainiert werden können. Ein wichtiger Aspekt ist zudem die Einbeziehung von Tiefeninformation in den Segmentierungsprozess. Das finale Netz berücksichtigt neben RGB-Daten auch eine Höheninformation. Dadurch wurde die Segmentierungsqualität mit nur geringem zusätzlichen Rechenaufwand verbessert. Zudem wurde ein Conditional Random Field zur iterativen Verfeinerung der Segmentierung eingesetzt. Das gemeinsame Training beider Komponenten, FCN und CRF, hat dazu beigetragen, dass die Qualität der Ergebnisse sich im Bereich aktueller Forschungsarbeiten bewegen.

Schlagwörter: Semantic Segmentation, Deep Learning, Convolutional Neural Networks, Fully Convolutional Neural Networks, multimodal, RGB-D, FCN, CNN, CRF
Datum: March 2016

Contents

List of Figures

List of Tables

List of Abbreviations

CNN

Convolutional Neural Network

CRF

Conditional Random Field

DNN:

Deep Neural Network

FCN

Fully Convolutional Neural Network

FPGA

Field Programmable Gate Array

GD

Gradient Descent

GPU

Graphics Processing Unit

H

Height

HHA

Height, Horizontal Disparity, Angle w.r.t. Surface Normals

mIU

Class-mean of the intersection over union metric.

PA

Pixel-accuracy

PCL

Point Cloud Library

mPA

Class-mean of the Pixel-accuracy

ReLU

Rectified Linear Unit

RNN

Reccurent Neural Network

SGD

Stochastic Gradient Descent

SIFT

Scale Invariant Feature Transform

ToF

Time of Flight

1 Introduction

Scene understanding is one of the most challenging tasks in computer vision and plays an important role for many novel and future applications - such as autonomous driving or automated image tagging. One can ask many questions about a scene: Which kind of objects are visible and how are they related to each other? What kind of place is it? What are the environmental conditions? Is it sunny or rainy?
The specific use case determines the choice of the questions. In case of autonomous driving, it's for example important to detect obstacles on the lane.
While humans can easily capture a scene at a glance, it's very hard for a machine. Semantic segmentation is one important step towards scene understanding. The goal is to segment a given image into coherent regions and to assign a label to it.

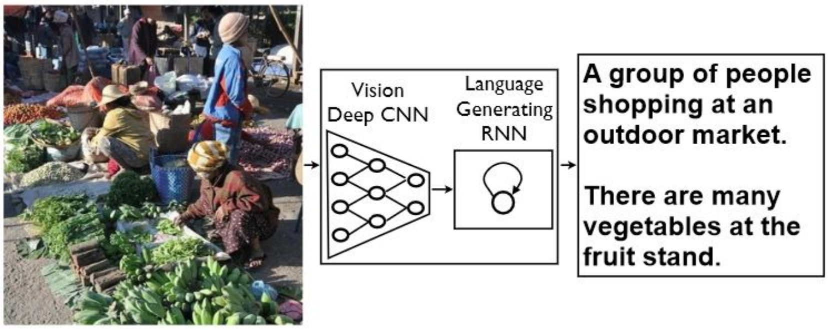

Vision
Deep CNN
Language
Generating
RNN
A group of people shopping at an outdoor market.
There are many vegetables at the fruit stand.

Figure 1.1: Scene understanding enables many novel applications. For example an "Image Caption Generator". By combining computer vision and natural language processing (NLP) techniques it's possible to generate descriptive sentences for images [1] [2].

The history of neural networks reaches far back into the past. In the mid-1980s the first deep neural networks were applied to recognition tasks. Although this was a scientific success, the interest decreased again, since the training took too much time. After many years, the development of larger, faster and cheaper computing units led to a revival of deep learning techniques. Deep Learning has already been successfully applied in many fields. New architectures, algorithms and tools popped up within a short time.
In the scope of this work I've examined on these new techniques in order to transfer them to a specific use case. The task was to develop a processing pipeline for semantic

segmentation, which could be used within an autonomous system - more specifically a forklift.
Such a system could benefit from a detailed and current scene understanding. For example, it's important to know if there are any obstacles that need to be bypassed. An improved system would even distinguish between different obstacles. For example, persons should possibly get special attention. People want to feel safe. To ensure acceptance among workers it would be a good idea to drive particularly careful in the presence of them.
The aim of this work was to elaborate intensively on the topic of semantic segmentation in order to identify opportunities. The development process was aligned on the current state of research.
Since there was no preliminary work, it was necessary to establish a general framework. Several design choices had to be done, data had to be collected and annotated. Gradually, the difficulties came to light. The main contribution of this work is the combination of current state of the art techniques to process RGB-D camera data for semantic segmentation. The main challenge was to handle difficulties due to a very limited training-set. To achieve good results, despite the difficult conditions, it was necessary to investigate on different approaches. Specifically, I have dealt with the following topics:

- neat ways for data augmentation
- extensive transfer of learning
- normalizing techniques to facilitate learning
- techniques for multi-modal deep learning

Aside from the initial motivation of this work, the final system is suitable for general use. The system can be easily adapted by exchanging the training data and specifying other categories.

The thesis is organized as follows. First I outline the topic and provide insights into the current state of research. After that the used hard- and software is introduced. In chapter 4 I jointly present my approaches and results. The chapter is divided into sections to investigate on different aspects. In order to allow an immediate assessment I will provide the experimental results directly. The results of these preliminary studies serve as a basis for the design of the final system. The last chapter summarizes essential results and gives a conclusions. Furthermore I will present ideas for future works.

2 Background and Related Work

This chapter builds the theoretical foundation of my work and provides insights into state-of-the-art research. Sub-topics are organized in a consecutive order. First I explain the task of semantic segmentation. Then I describe neural networks in general and introduce subsequently more specialized networks. Moreover, the main ideas of Conditional Random Fields (CRF) are explained, as they play an import role for this work. Furthermore I will introduce different techniques to leverage from depth information.

2.1 Semantic Segmentation

Semantic segmentation can be understood as a classification task at pixel-level. In the literature the terms "scene parsing" or "image parsing" are frequently used in the same context.
The task is to determine a two-dimensional classification map of same size as the input image. This map assigns an object-class of a pre-defined set to each pixel. The output is helpful to provide a detailed scene understanding. Such an understanding is in particular useful for autonomous systems.

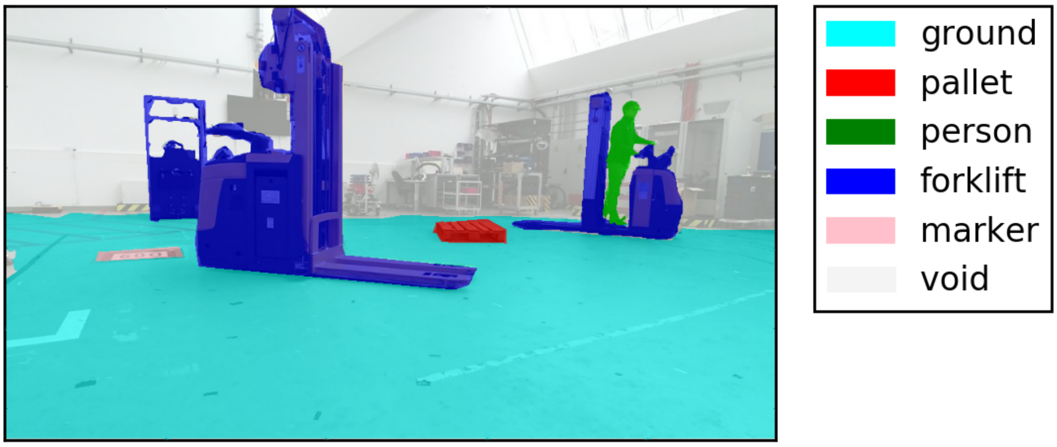


Figure 2.1: Exemplary image to illustrate semantic segmentation. The object category of each pixel is identified by a specific label.

Semantic segmentation unifies several fundamental problems of computer vision. First a proper and robust representation of the input space is needed. In particular local features are of interest, as they provide an abstract description of the image content. These features are used to carry out local classifications. A main challenge of semantic segmentation is to achieve robust classifications while preserving spatial resolution.

The attempt of this work is to solve the entire problem of semantic segmentation with the help of deep learning techniques and claims to unify all subtasks into a single trainable system. In the following I recap the most important topics related to the system. Moreover I give a compact overview of recent related works and point out alternative approaches.

2.2 Convolutional Neural Networks

The conventional approach to solve a classification problem is to decompose it into at least two independent subtasks - feature extraction and classification. The process of feature engineering can be very complex, typically requires domain expertise and a lot of development time. A popular example for a hand-crafted visual feature is SIFT developed by David Lowe. Lowe has put a lot of effort in the development of a feature, which is invariant to image scale and rotation (SIFT stands for "scale-invariant feature transform") [3]. SIFT has been successfully applied for instance for object recognition or panorama stitching.

The downside is that these features are very rigid and do not automatically adapt to the problem. The classification step however has always been a machine learning approach. Based on the extracted features any appropriate classification algorithm can be applied (e.g. a support vector machine).

An alternative approach is to integrate both feature extraction and classification into one learning process. Since the features are learned automatically there is no need for expert domain knowledge and time-consuming engineering. Furthermore it is possible to train feature extraction and classification in an end-to-end manner, which allows a better optimization and adaption of both parts.

This work uses a special convolutional neural network (CNN), which extracts features and classifies the input image pixel-wise for semantic segmentation.

For the sake of a step-wise introduction I will first introduce feed-forward neural networks and explain the basic concepts of neural networks. This will build the theoretical basis for my explanations in the subsequent chapters.

2.2.1 Feed-Forward Neural Networks

Feed-forward networks are collections of neurons, which are organized in layers $1, ..., L$. Every neuron of layer $l > 1$ is connected to all neurons of the previous layer $l - 1$. When deployed a feed-forward net has a distinct direction ("forward pass") and there are no cycles (feedback loops). The layers are distinguished between input, output and hidden layers. Hidden layers are located between the input and output layer and therefore do not have a direct connection to the environment.

The special case where the network consists of only two layers (in- and output) is called single-layer perceptron. Since the input layer is not parameterized, it is frequently not considered as a layer. For this reason, it is called single-layer perceptron. More usual are the so called multi-layer perceptrons, where multiple layers are stacked on top of each other. Thus, such a multi-layer perceptron has at least one hidden layer.

Figure 2.2 shows a simple network with an input-, a hidden- and an output-layer. Each connection between two neurons i and j is associated with a weight $\mathrm{w_{j,i}}^{(1)}$. The superscript l denotes the number of the layer. These weights are used to calculate a weighted sum of all J inputs. In general there are also bias-terms $\mathrm{b_i}^{(1)}$ associated to each neuron. This bias term shifts the weighted sum by a certain amount.

This intermediate result is then processed by a so-called **activation function** h. Usually this function applies a non-linear transformation. The result a_i^l is the output of the neuron and possibly serves as input for the next layer (if layer l is a hidden layer). Equation 2.1 shows the formula for calculating the activation a_i^l of a specific neuron located at position i in layer l for J inputs x_j:

$$a_i^l = h(b_i^l + \sum_{j=1}^{J} w_{j,i}^l \cdot x_j) \tag{2.1}$$

where:

a_i^l is the activation of neuron i at layer l

$h(x)$ is the activation function

$w_{j,i}^l$ is the weight of neuron i at layer l associated with input neuron j

b is the bias of neuron i at layer l

x_j input (result of neuron j when $l > 1$)

Typical activation functions are the sigmoid (logistic) or the hyperbolic tangent function. In this work so-called rectified linear units (ReLU) were used:

$$h_{ReLU}(x) = max(0, x) \tag{2.2}$$

ReLUs employ a simple ramp function and are crucial for the success of recent deep networks [4]. [5].

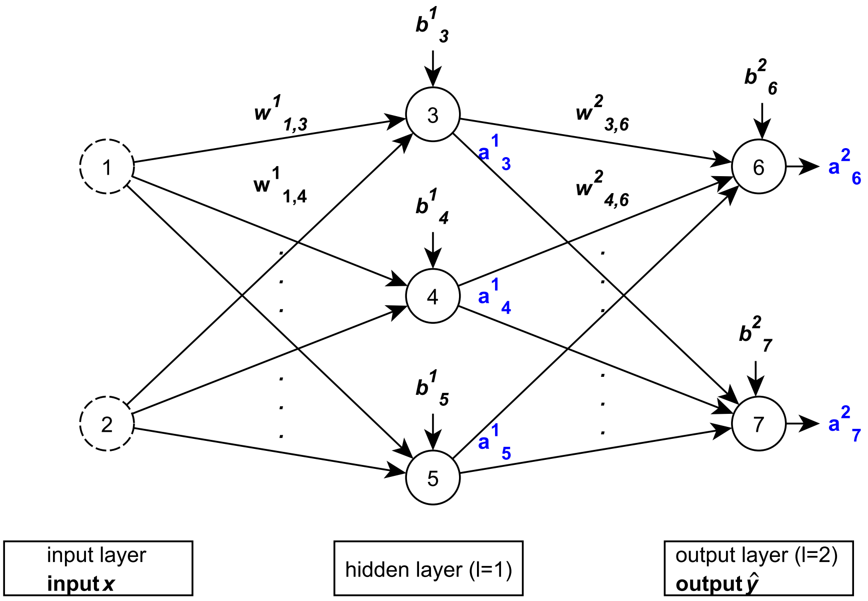


Figure 2.2: This exemplary feed-forward network consists of one input-, one hidden- and one output-layer. The neurons, visualized by solid circles, are organized in layers l. Each connection between two neurons is associated with a (learned) weight $w^l_{i,j}$. The weighted sum of the input values plus a bias term b^l_i is then fed into an activation function, which results in the activation a^l_i. The values are propagated throughout the entire network. Finally, the output layer is reached which produces the net output $\hat{y}$.

A neural network is entirely defined by its structure (topology), the activation functions and its parameters (namely weights and biases). In general a neural network can approximate any finite function [6].

To handle discrete outputs, such as it is needed for classification tasks, a probability function for each of the defined states can be trained in order to finally select the class with the highest probability. Usually, the probabilities are obtained by applying a **softmax** normalization (see equation 2.3) in the output layer. An exponential function $exp(\hat{y}_{n,y_n})$ turns the class prediction into a non-negative value. Then this value is normalized by the denominator to fit the $[0, 1]$ range. In this way all class probabilites sum up to 1.

$$\sigma_c = \frac{exp(\hat{y}_{n,y_n})}{\sum\limits_{c=1}^{C} exp(\hat{y}_c)} \tag{2.3}$$

where:

C is the number of classes

$\hat{y}_c$ is the predicted class score with $c = 1, ..., C$

$\hat{y}_{n,y_n}$ is the predicted class score of the true class y_n for sample n

The parameters of a neural network are learned based on training samples. In case of supervised learning, where a labeled training set is given $\{(x_\mathrm{n}, y_\mathrm{n})\}$, all misclassifications have to be penalized by a suitable **loss function** $l(\hat{y}_n, y_n)$, also called cost function. This loss function must be minimized. Usually, a regularization term r is added in order to preserve the model from overfitting. This occurs when individual parameters dominate over others. The weights should be kept as small as possible. This why the regularization term is also called "weight decay". This can be achieved by minimizing the L2 norm of the parameter vector. The balance between regularization and loss is controlled by a hyperparameter α. In summary the regularized error function E is as follows:

$$E(w, b) = l(\hat{y}_n, y_n) + \alpha \cdot r(w) \tag{2.4}$$

where:

w weight

b bias

y_n is the class label i.e. the true class

$\hat{y}_n$ is the class prediction

α is the multiplier for the regularization term

$r(w)$ is the regularization term (weight decay)

A popular loss function for multi-class classification objectives is the multinomial logistic loss. Passing class probabilities σ conditioned on the input data x_n, the multinomial logistic loss is the negative log-likelihood of the true class y_n. The total loss l_{tot} of one training batch is given by the sum over all samples N. Equation 2.5 shows the normalized multinomial logistic loss $\bar{l}$ which additionally includes a normalization term $1/N$:

$$\bar{l} = -\frac{1}{N}\sum_{n=1}^{N} log(\sigma_{n,y_n}) \tag{2.5}$$

where:

N is the number of training samples

y_n is the class label i.e. the true class

σ_{n,y_n} is the class probability conditioned on the input of the true class y_n

As previously mentioned, the training process aims to adopt the parameters such that the loss is minimized over all training samples. Thus, we search the function f parametrized by w, which minimizes the loss $Q(x, y, w) = l(f_w(x), y)$ averaged over the examples (i.e. pairs of x and y). A standard technique for solving non-linear optimizing problems is **gradient descent (GD)** [7]. GD iteratively adopts variables towards the direction of the negative gradient. This way a local minimum can be found. The learning rate γ determines the steps size of each adaptation.

$$w_{t+1} = w_t - \gamma \frac{1}{N}\sum_{n=1}^{N} \nabla_w \, Q(x_n, y_n, w_t) = w_t - \Delta w_t \tag{2.6}$$

where:

w are the weights

γ is the learning rate

Q is the loss function

∇_w is the gradient w.r.t. w

Δw_t is the weight update

N number of samples

According to equation 2.6 the weight update Δw_t incorporates all training samples $\{(x_{\mathrm{i}}, y_{\mathrm{i}})\}$.

An abreviation of GD is the **stochastic gradient descent (SGD)** method [8]. SGD shows its strengths when the training set is very large. Instead of computing the gradient based on all samples, only one or a few randomly chosen samples are considered. The updated weight w_{t+1} in the following equation is based on a randomly chosen example x_t.

$$w_{t+1} = w_t - \gamma_t \nabla_w Q(x_t, w_t) = w_t - \Delta w_t \tag{2.7}$$

where:

w are the weights

γ is the learning rate

Q is the loss function

∇_w is the gradient w.r.t. w

Δw_t is the weight update

An extension of SGD is momentum [7]. Momentum helps to speed up the learning process by adding a fraction m of the previous weight update.

$$w_{t+1} = w_t - (\Delta w_t + m\, \Delta w_{t-1}) \tag{2.8}$$

where:

w are the weights

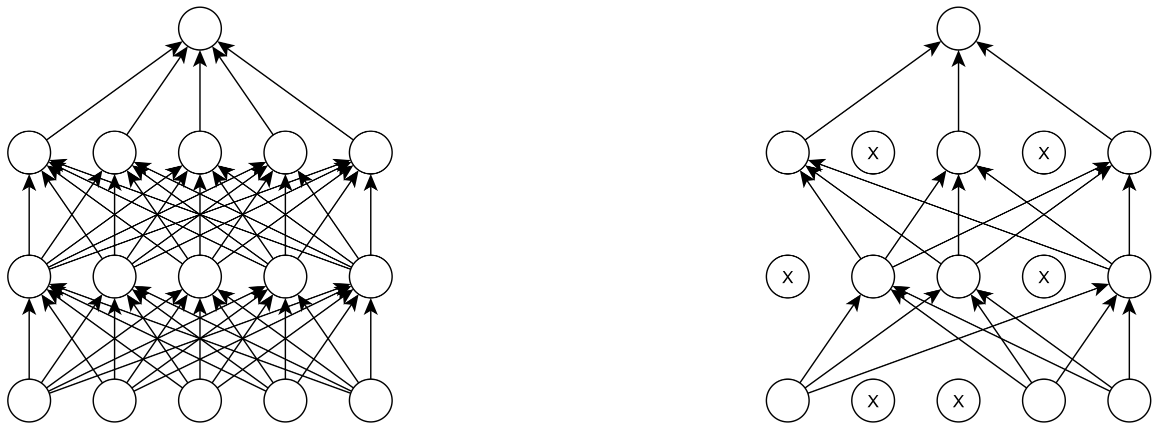
Δw_t is the weight update

m is the momentum parameter

If both weight updates point towards the same direction the step size is amplified. In case of changing gradients momentum smooths out the variations and provides stability.

By providing training samples x and y, a single layer perceptron can be optimized by any GD variant. This is because the in- and outputs of the single weight layer are known. Therefore the delta term or error $\hat{y} - y$ is directly computable. This is not the case for networks including hidden layers.
This fact is the motivation for applying the so-called **backpropagation** algorithm [7]. In order to adjust the parameters of hidden layers the error is propagated subsequentially back through the net. This process is called backward pass. The parameter adjustment (i.e. the delta term) of layer l depends on the activation of the same layer, the weights and the delta terms of the subsequent layer $l + 1$. The activations are determined during the forward pass. In this way the partial derivatives w.r.t. the parameters of each neuron can be determined (chain rule). Gradient descent uses these delta terms for updating the parameters.

Dropout invented by Hinton et al. is another powerful regularization tool for training neural networks [9]. For each iteration dropout randomly turns off a portion of neurons (see figure 2.3).



(a) before

(b) after

Figure 2.3: Demonstrating the effect of dropout applied on a simple neural network. Dropout randomly removes a portion of connections between subsequent layers and produces a thinned net [10].

That way each hidden unit is encouraged to learn meaningful features without relying to much on other hidden units. More precisely, randomly chosen submodels are trained, thus comparable to ensemble learning. At test time the predictions are averaged. This leads to significantly better generalization [10].

Neural networks trained by the backpropagation algorithm suffer from the so-called "vanishing gradient problem" [11]. Especially for very deep networks this effect is even amplified. For example the gradient h′(x) of the sigmoid function $h(x) = (1 + exp(-x))^{-1}$ tends towards zero as $|x|$ increases. This cumbers the learning process and limits the network size for end-to-end training. In practice this problem is attenuated by using ReLUs, careful initialization [12][13] and small learning rates. Furthermore, very deep networks are usually trained step by step - i.e. the size of the network is gradually increased [5].

Parametric ReLUs (PReLU) include an adjustable slope for the negative part. This slope can be controlled by a factor s_i.

$$h(x) = max(0, x) + s_i min(0, x). \tag{2.9}$$

where:

s_i factor to control slope for the negative part

He et al. have recently shown that learning the PReLU parameter s_i together with an improved random initialization helps end-to-end learning of large networks [13]. Another attempt for preventing the "vanishing gradient problem" was described by Ioffe and Szegedy [11]. They normalize the inputs of the activation functions such that the distribution remains stable during training.

2.2.2 Convolutional Neural Networks

In the previous section the main ideas of feed-forward networks were introduced. A Convolutional Neural Network (CNN) is a special type of feed-forward neural network.

CNNs are inspired by the biological processes of visual perception. Therefore CNNs are mainly used for image and video applications. In principal there are three extensions that distinguishes a CNN from a simple feed-forward network: local receptive fields, weight sharing and spatial pooling.

The input of a CNN is processed patch-wise by a number of learned convolutions.

With the help of these convolutions various filter techniques can be applied - e.g. blurring, edge or corner detection. Discrete convolutions are calculated by shifting a filter mask over the input image and calculating the sum of products. The result is written to the current center of the mask. These filter masks span the so-called **local receptive fields**. Intuitively they define the image crops that a neuron actually "see". In terms of neural networks the entries of the filter masks correspond to the weights of the neurons and the filter size defines the connectivity between neurons of subsequent layers.

Since many filters are applied several feature maps are obtained. All neurons of a feature map share the same weight matrix. This property is called **weight sharing**.

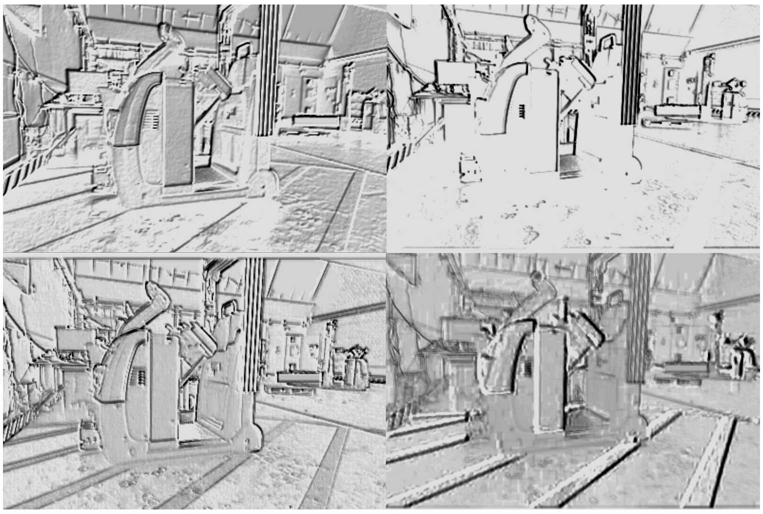


Figure 2.4: Visualization of four exemplary filter responses taken from the third convolutional layer of a CNN. Each filter responds differently to the input. For example, some filters are mainly sensitive to certain orientated structures.

Weight sharing contributes to the property of shift invariance, since the same convolutions are applied to the entire input image. At the same time weight sharing reduces the amount of free parameters dramatically. This makes the net much easier to train and has a strong regularization effect.

The learned filters expose different kind of visual features. Investigations by different researchers have shown that the early layers typically capture very simple features, similar to gabor filters and color patches [14] [15]. These features are very general and therefore apply to many use-cases. In contrast the features of higher layers are more abstract and reflect the gist of the learned classes. Exemplary filter responses of an early convolutional layer are shown in figure 2.4.

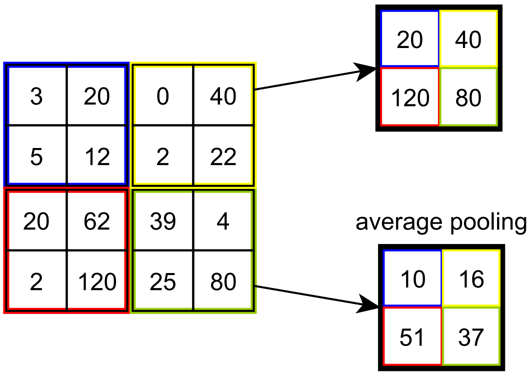


Figure 2.5: This example demonstrates the effect of max and average pooling.

Another important component of CNNs are **pooling** layers. Typically the data is sub-sequentially downsampled by combining spatial pools to a single value. Pooling reduces the feature space and grant a small amount of translational invariance. This is due to the fact that neurons within a region are mapped onto a single neuron. Frequently used pooling operations are max and average-pooling (see figure 2.5). A more recent method is stochastic pooling [16], which serves as an regularizer similar to dropout.

So far the feature maps cover a two-dimensional space. For high-level reasoning, such as classifying an image, a one-dimensional representation is needed. This can be achieved by a **fully connected layer**, which connects every single neuron to all neurons of the previous layer. Thus, fully connected layers aren't spatially located anymore. The one-dimensional output, is suitable for a traditional multi-layer perceptron classifier (compare to 2.2.1).

CNNs have shown state of the art performance for image classification [4] [5]. Many models are shared within the researcher community, which makes it possible to follow up and adopt the networks to other classification problems. This process is called "finetuning". Initializing from pretrained weights works well for many cases. Even a model trained on a very distant task is often a better starting point than initializing from random numbers (like "xavier" [12] or "msra" initialization [13]). One reason for this is that features of early layers are in fact very general. Yosinksi et al. investigated on the question of transferability [17]. Deep Learning Frameworks such as Caffe (see section 3.2) facilitate the transferal of learned weights.

After the great success of CNNs for image classification, the natural next step was to apply CNNs to local tasks, such as bounding box detection [18] [19] [20], local correspondence [21] or semantic segmentation [22] [18] [19] [20].
Even though the results for semantic segmentation tasks yield new record accuracies on various datasets, the methodologies had some major drawbacks. A common approach was to identify clusters in the image, which may hold the desired objects ("region proposals"). All clusters are then classified independently. The results are then merged to obtain a segmentation mask. This approach requires extensive pre- and post-processing, making it unsuitable for end-to-end training and includes a considerable overhead. In addition, the clustering result is likely to be suboptimal. A fully visible object may be divided into several regions.
Therefore it was a huge improvement, when Long et al. developed a new net architecture for semantic segmentation namely "Fully Convolutional Networks" (FCN) [23].

The following section will describe the differences and how a conventional CNN can be converted into a Fully Convolutional Network.

2.2.3 Fully Convolutional Neural Networks

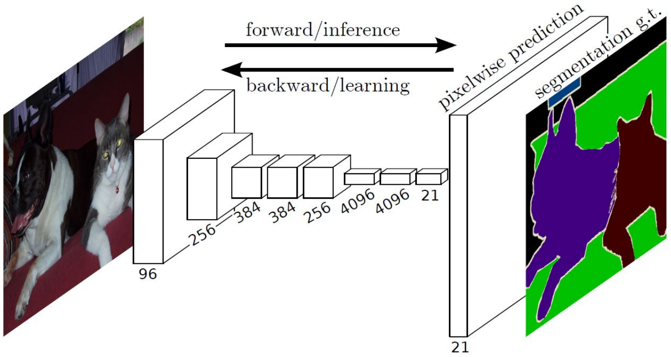


Figure 2.6: A Fully Convolutional Network (FCN) processes the entire image to make pixelwise predictions of the same size as the input image. The network is trained end-to-end by backpropagation and a pixel-wise loss. This requires dense ground truths for supervised training.

In 2014 Jonathan Long, Evan Shelhamer and Trevor Darrell published the paper "Fully Convolutional Networks for Semantic Segmentation" [23]. They presented a new net archiecture for pixel-wise prediction. There are several advantages compared to previous works. A FCN processes whole-images and produces dense predictions in the form of probability maps. The training can be applied in an end-to-end manner. Required are densely labeled images for supervision.
Another advantage is the ability to re-interpret existing classification nets as a FCN. Therefore a FCN can benefit from pre-trained CNN models.
In order to convert a CNN into a FCN one has to replace the inner products (fully connected layer) by convolutions with a kernel size of 1x1 (see figure 2.7).

While the fully connected layer of an image classification net completely discards spatial information and delivers only one feature vector for the entire image, the fully convolutional layer produces a feature vector for every pixel. Based on this feature map a pixel-wise classification can be performed. This yields a probability map for each class. To restore the original image dimensions this map is upsampled by so-called deconvolutions. Depending on the weights, a deconvolution can serve as a bilinear interpolation.

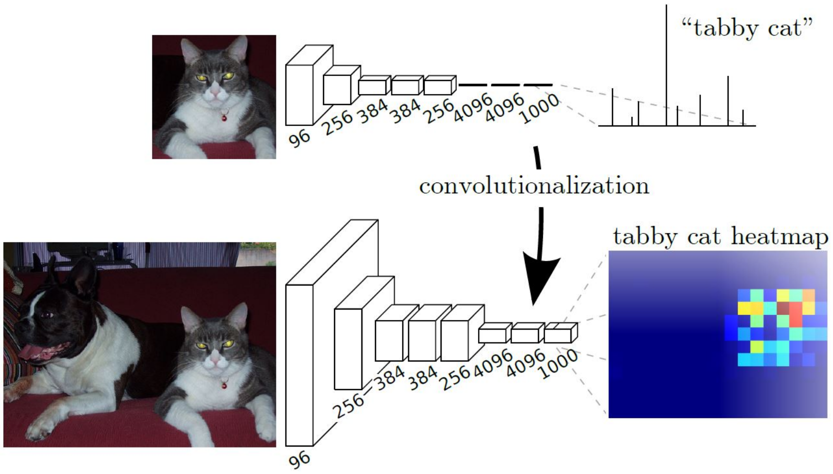


Figure 2.7: Classification at image-level versus pixel-level - To retain spatial information one can "convolutionalize" a network by replacing fully connected layers by convolutions.

While pooling improves classification accuracy, it partially neglects spatial information. This is a drawback as it limits the spatial accuracy of the segmentation. The VGG16- and Alex-net for example include 5 pooling layers of stride 2 [5][4]. In total this results in downsampling by factor 32 (2^5). In general this leads to coarse segmentation maps. Very small objects aren't even considered at all.
To solve this issue they use a so-called "skip" architecture (see figure 2.8). By fusing predictions of different strides they are able to refine the segmentation maps. In addition to the classification based on the 32-stride feature map, they do the same for the outputs of earlier pooling layers and fuse the classification scores by taking the element-wise sum. Of course such an element-wise operation affords inputs of the same size. Thus, the smaller prediction maps must be interpolated accordingly.

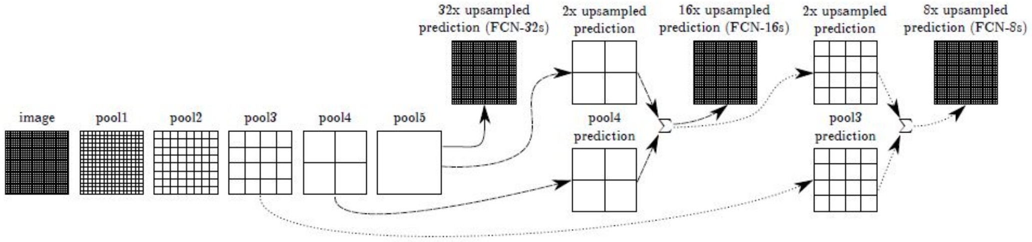


Figure 2.8: This figure illustrates the "skip" architecture developed by Long et al.. To keep the net-representation compact only pooling and classification layers are shown. All intermediate convolution layers are omitted. The coarsest prediction is based on pool5 only (solid line) and is called FCN-32s. The FCN-16s version (dashed line) first upsamples stride 32 predictions by a factor of 2 and then combine the output with the predictions of pool4. The FCN-8s variant (dotted line) combines predictions of stride 32, 16 and 8.

This way they obtained new state-of-the-art results on several data-sets.
However, this method left enough space for improving the spatial accuracy.
Noh et al. achieve finer predictions by learning a deconvolution network [24]. The general idea is to extend a convolutional network by its mirrored counterpart. By learning the inverse operations (pooling vs. unpooling, convolution vs. deconvolution) it's possible to produce a finer segmentation. On the other hand, this approach doubles the number of parameters to be learned. Another way is to refine the segmentations of a FCN by a employing a graphical model. I will describe such an approach in the following section.

2.3 Conditional Random Fields

In many cases the class instances are somehow related to each other. In such cases it's possible to model the dependencies in order to incorporate them into the classification process. Such a structured prediction is required in many different fields, for example in Natural Language Processing or Computer Vision.
Conditional Random Fields are designed exactly for this purpose. It's a discriminative model, which means that it directly describes how to assign a label $\hat{y} \in y_1, ..., y_L$ when a feature vector x is given - i.e. the conditional probability $P(\hat{y}|x)$ [25]. A generative model describes the reverse process, i.e. rules for generating a feature vector x, when a label y is given. Known relationships between observations can be expressed in a graph structure. Each node represents an instance y_i associated with an observation f_i and the connections between nodes represent the relation of those instances.

Image segmentation is a vivid example for structured prediction. In most cases a reliable classification is only possible if the local neighborhood is considered.
A FCN takes this already into account. The classification is based on features detected within the local receptive fields. However, as already mentioned, the spatial resolution is poor.
A CRF can significantly improve this drawback. In the case of semantic segmentation it's helpful to encourage local consistency, i.e. it is very likely for neighboring pixels to share the same object class. On the other hand the appearance (e.g. color) shouldn't differ too much. These inter-relations can be modeled by an undirected graph. By enforcing the spatial and appearance constraints a coarse segmentation map can be refined.
In general the graph structure can be arbitrary shaped. In the following the description focuses on the case where each node is connected with every other node. A CRF based on such a graph is called **fully connected CRF**. On the one hand a fully connected CRF is capable to represent long-range dependencies, on the other hand the model is very complex. For a long time this made it impractical for real-world applications. One possible approximation method for fast inference was developed by Philipp Krähenbühl and Vladlen Koltun [26].

For the purpose of image segmentation, one can define an energy function conditioned on the image I. The final goal is to minimize this energy function E in order to get the most likely segmentation result in terms of the defined model. $E(x|I)$ consists of unary and pairwise potentials conditioned on the input image I:

$$E(\hat{y}|I) = \underbrace{\sum_i U(\hat{y}_i)}_{\text{unaries}} + \underbrace{\sum_{i<j} \psi_p(\hat{y}_i, \hat{y}_j)}_{\text{pairwise components}} \tag{2.10}$$

where:

I is the input image
$\hat{y}_i$ and $\hat{y}_j$ are the assigned classes for pixels i and j

The unaries can be obtained for example by the FCN output and is the negative log of the likelihood of assigning a label to a pixel i (thus the loss). In general the potentials can be arbitrarily formed. But [26] restrict the pairwise potentials to linear combinations of Gaussian distributions - i.e. Gaussian mixtures.

The actual pairwise potentials for pixels i and j (equation 2.11) are defined by the product of a label compatibility function $\mu(\hat{y}_i, \hat{y}_j)$ and a two-kernel potential $k(f_i, f_j)$ (Gaussian mixture) [26].

$$\psi_p(\hat{y}_i, \hat{y}_j) = \underbrace{\mu(\hat{y}_i, \hat{y}_j)}_{\substack{\text{label} \\ \text{compatibility}}} \; \underbrace{k(f_i, f_j)}_{\substack{\text{two-kernel} \\ \text{potential}}} \tag{2.11}$$

where:

f_i and f_j are feature vectors of pixels i and j

$\hat{y}_i$ and $\hat{y}_j$ are the class predictions for pixels i and j

The compatibility function $\mu(\hat{y}_i, \hat{y}_j)$ is represented by a symmetric matrix. This function describes the fact that some labels are more likely to coexist. For example, it's likely that labels of "sky" and "plane" are close to each other, whereas it's unlikely for the labels "sky" and "car". The entries of the matrix are learned during training.
The actual two-kernel potentials $k(f_i, f_j)$ (shown in equation 2.12) consist of an appearance and a smoothness kernel. Here, appearance is described by a color vector c. The appearance kernel underlies the intuition that nearby pixels of similar color are likely to share the same class. The smoothness kernel suppresses small isolated regions.

$$k(f_i, f_j) = w^{(1)} \underbrace{exp\left(-\frac{|p_i - p_j|^2}{2\theta_\alpha^2} - \frac{|c_i - c_j|^2}{2\theta_\beta^2}\right)}_{\text{appearance kernel}} + w^{(2)} \underbrace{exp\left(-\frac{|p_i - p_j|^2}{2\theta_\gamma^2}\right)}_{\text{smoothness kernel}} \tag{2.12}$$

where:

f_i and f_j are feature vectors of pixels i and j

p is the position vector $p = (x, y)^T$

c is the color vector e.g. $c = (r, g, b)^T$

θ_α^2, θ_β^2, θ_γ^2 are parameters to control the variance of the underlying Gaussians

The filter operation can be controlled by the hyperparameters θ_α^2, θ_β^2 and θ_γ^2. They describe the variance of the underlying Gaussian distributions. For optimization the authors were using a simple grid search.

In summary one can say that the pairwise potential describes the cost of assigning the labels $\hat{y}_i$ and $\hat{y}_j$ to a pixel pair i and j. The minimization of the energy function $E(\hat{y}|I)$ yields the most likely label composition for a given image.
This optimization problem is actually solved by a so-called iterative meanfiled approximation [26]. The key idea of a meanfield approximation is to express the exact distribution $P(X)$ by a simpler factored distribution $Q(X)$. Assuming that each factor $Q_i(X_i)$ is independent, $Q(X)$ can be expressed by the product of all factors:

$$Q(X) = \prod_i Q_i(X_i) \tag{2.13}$$

Each meanfield iteration in a fully connected CRF performs 3 essential steps. A message passing step, a compatibility transform and a local update.

1. **Message passing:** The pairwise potentials are calculated:
2. **Compatibility transform:** The compatibility transform $\mu(\hat{y}_i, \hat{y}_j)$ is applied.
3. **Local update:** In this step $Q_i(X_i)$ is updated by combining the unaries and pairwise potentials.

These explanations should provide a high-level understanding of the algorithm. Implementation details can be found in [26]. Moreover, the meanfield approximation is discussed in the next section (see section 2.3).

Since all nodes are connected with each other, which implies that the kernels spans over the entire image, the filtering for obtaining the pairwise potentials (message passing) is a computational bottleneck. Classical filtering with a filter mask would require to evaluate N times a sum over all N variables. Thus, the naive approach has a quadratic complexity in terms of the number of variables.

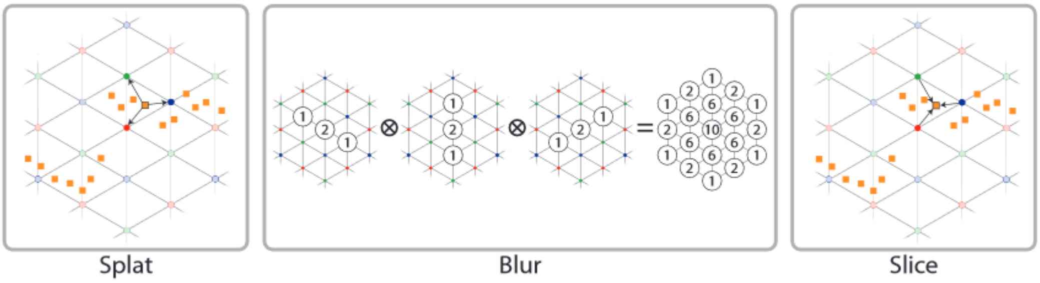


Figure 2.9: With the help of the permutohedral lattice high-dimensional Gaussian filtering can be approximately performed in three steps: (1) splatting the instances onto vertices of the permutohedral lattice; (2) apply a one-dimensional (discrete) convolution along each axis; (3) interpolate (slice) the results to the original positions.

One possible way for approximate filtering of high-dimensional input, is based on a data structure called permutohedral lattice [27]. The permutohedral lattice models uniform structures. Figure 2.9 shows the permutohedral lattice and the required steps for applying a high-dimensional Gaussian filter. This way the inference time became practical (e.g. 0.2 seconds compared to 36 hrs). The work [26] also demonstrates improvements in labeling and segmentation accuracy.

On the coupling of FCN and CRF

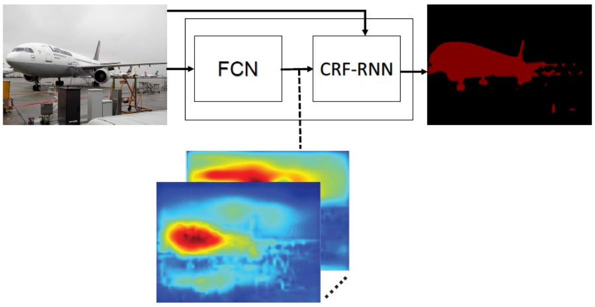


Figure 2.10: The coupling of a FCN and CRF can be used to benefit from the strengths of both models. A FCN provides a coarse, but robust segmentation. The CRF is then used for refining the segmentation accuracy.

Chen et al. were the first who combined FCNs and fully connected CRFs [28]. The work shows how to consolidate the strengths of both models. In this way the coarse FCN results are refined by a fully connected CRF, as described before. They employ the implementation of Krähenbühl and Koltun [26], together with a FCN as described by Long et. al [21]. The combination of FCN and CRF led to new state-of-the-art on

the PASCAL VOC-2012 image segmentation task. However the authors point out further opportunities by learning both models in an end-to-end fashion.
In 2015 Zheng et al. suggested a CRF implementation in the form of a Recurrent Neural Network (RNN) [29]. In contrast to feed-forward neural networks a RNN has connections back to earlier layers [30].

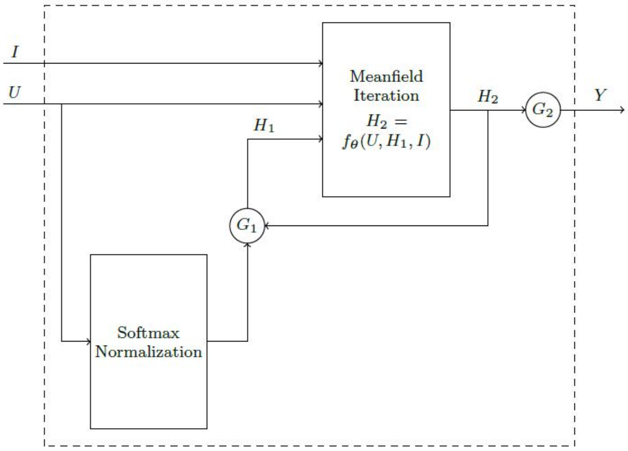


Figure 2.11: A general overview of the CRF-RNN network. Iteratively the energy function, defined by the unary and pairwise potentials, is minimized by a meanfield approximation. The termination criterion is a fixed number of iterations T.

The basic idea of Zheng et al. was to implement an iterative meanfield approximation as a stack of CNN layers. In the following this method is called **CRF-RNN**.
As input serves the given color image I and the previously computed FCN unaries U (see figure 2.11). A copy of the unaries is normalized by a softmax operation in order to obtain H_1. These three ingredients I, U and H_1 are then fed into the meanfield iteration. While I and U are preserved, H_1 is updated before the next iteration is started. In total T number of iterations are performed (Zheng et al. use $T = 10$ for inference). Each meanfield iteration consists of 5 steps:

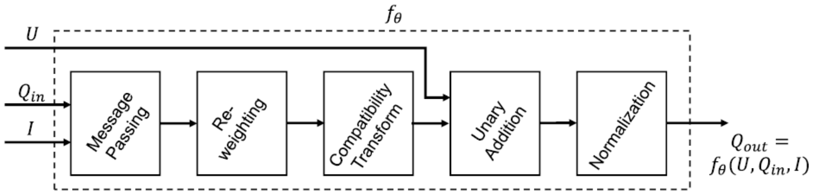


Figure 2.12: Subsequent steps of a meanfield iteration. One interation can be interpreted as a sequence of common CNN operations [29].

1. **Message passing:** The components of the two-kernel potential (appearance and smoothness; see equation 2.12) are obtained by high-dimensional filtering as illustrated in figure 2.9. The coefficients of the spatial and bilateral filter are based on the pixel locations and the RGB values.
2. **Weighting filter outputs:** A weighted sum combines the outputs of the previous step (compare to equation 2.12). This can be implemented as a convolution with 1x1 filter size, two input channels (the output of appearance and smoothness kernel) and one output channel.
   Since the relevance of color and position differs for different classes, the weighting is class-dependent. The weights are learned during backpropagation.
3. **Compatibility transform:** The compatibility transform is another class-dependent operation which is learned during backpropagation. This operation can again be viewed as a classical convolution (1x1 filter size). The compatibility function $\mu(\hat{y}_i, \hat{y}_j)$ is part of the two-kernel potential $k(f_i, f_j)$ and is described in equation 2.12.
4. **Local update (unary addition):** In this step the unary and pairwise potentials are added (compare to equation 2.10).
5. **Normalizing:** Finally the refined potentials are again normalized by a softmax function. The output may serve as an input for the next iteration (if iteration count is $\leq T$).

In order to learn the CRF parameters by backpropagation all steps have to be differentiable. This way the error differentials can be passed backwards for parameter adaption.
The CRF-RNN implementation of [29] integrates into the famous deep learning framework Caffe. This way a given FCN model was jointly finetuned with a CRF. This yield a new Pascal VOC 2012 record for the segmentation task.

Shortly after the publication of Zheng et al. a very similar work has been published [31]. Their implementation mainly differs in the feature extraction part as both finetune from different models.
A somehow different approach was presented by Liu et al.[32]. Due to further optimizations, they even outperform the results of Zheng et al.. They carefully designed the unary and smoothness terms and conducted several experiments to find an optimal configuration [32].

Other subsequent works deal for example with the efficiency of the CRF learning process and the incorporation of further contextual informations [33][34].

2.4 Incorporating Depth into CNNs

A general problem in computer vision is that images provide only incomplete information about the captured scene. Depending on the given conditions an image may also be a misleading representation.
Imagine a scene illuminated by diffuse light in which objects are indeed shaped differently, but have the same opaque material and color.
What do we see? The contours of each object disappear and are no longer distinguishable. In such a scenario, it's impossible to classify or segment objects based on the visual appearance. Of course, this is an corner case, but there are many nuances in between. Depth has some uncontroversial advantages over conventional images. It provides another piece of information - the three-dimensional shape of an object. Many objects are distinguished only by their shape and not by their color or texture. Thus it is a meaningful complement and enables hopefully a more robust basis for classifications.
However the technical implementation of a depth sensor is crucial for the application. In the case described above a Time of Flight (such as the Kinect v2 used in this work 3.1) would be suitable. Whereas a stereo camera for depth estimation would be useless, as it depends on the availability of visual features. In addition there are further technical limitations such as resolution, recording rate, field of view and depth range. Another question is how to use depth in a CNN setting, which was primarily designed for two-dimensional input data? The three dimensional space must be reduced to two dimensions. But what is an appropriate representation?

2.4.1 Depth Encoding

As most CNNs are designed for RGB-input, a three-channel representation becomes very handy. That way developers can leverage from standard models pre-trained on RGB-images. This becomes important since there may be a lack of (very) large and suitable depth data sets. But large training sets are required for a training from scratch.
The fact that depth and color represent two different things, is actually not a big

problem. Just like a conventional color image a depth encoded image shows different shapes and colors (see figure 2.13). Therefore it's practicable to initialize a depth-based network from a color-based network. At least features of early layers are universally applicable.
There are several possible encodings:

1. One simple solution for the problem is to use the distance between the camera and the objects (depth image). The measurement considers the z component perpendicular to the camera sensor. That means the other two components x and y (perpendicular to the optical axis) are discarded. In order to fulfill the input requirements, the resulting grayscale image is then replicated over all three channels.

2. For a better utilization of the given input space, the gray values could also be expanded over all channels by using pseudo colors. Eitel et al. employ for example a jet colormap [35].

3. A more extensive method includes the estimation of surface normals. Each dimension of the surface normal is represented by a separate channel.

4. Gupta et al. propose an even more involved strategy [19]. They encode depth into height over ground, horizontal disparity and angle w.r.t. gravity. According to the first character of each component this method is called **HHA**-encoding. The A-component requires again the estimation of surface normals.

Since methods 3 and 4 require the estimation of surface normals they are computationally more expensive. Gupta et al. defend this effort by presenting impressive results. Furthermore they argue that it's unlikely that a CNN will learn such a representation on its own. Compared to 3 and 4 methods 1 and 2 are computationally inexpensive. Eitel et al. report good results even compared with the more sophisticated approaches 3 and 4. However, one potential reason for this result is that the height component was not fully exploited. They use a data-set, which includes only table-top images. According to the authors all objects had the same height over ground. Hence, the height component became useless for this test case. Possibly this could have been improved. As positions below the table and far beyond are needless for this application one could consider that in the encoding. This could be done by mapping a smaller range starting from the tabletop onto the image domain (e.g. [0, 255]).

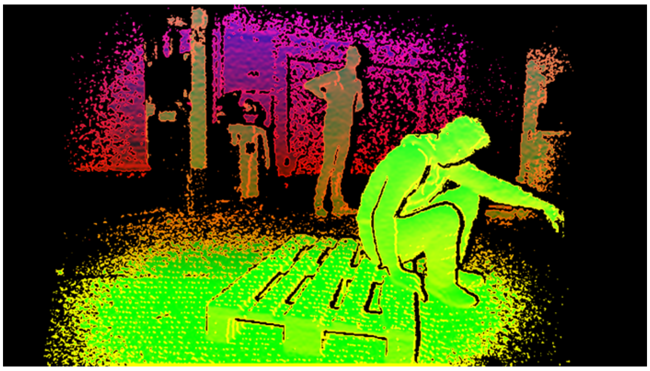


Figure 2.13: Exemplary HHA encoded image. Black represents missing depth values. In order to reconstruct these missing values an inpainting technique can be applied.

Depending on the method and the properties of the depth camera, as well as the perceived scene, there occur more or less pixels with missing depth information. With the help of the local neighborhood these holes can be filled with meaningful values. For example, small holes can be easily closed, with a median filter. Larger holes require more sophisticated methods. There exist several **inpainting** variants for this purpose. For time critical applications, such as it is here, an efficient solution is required. Most inpainting methods borrow small pieces from the original image to fill the missing values.

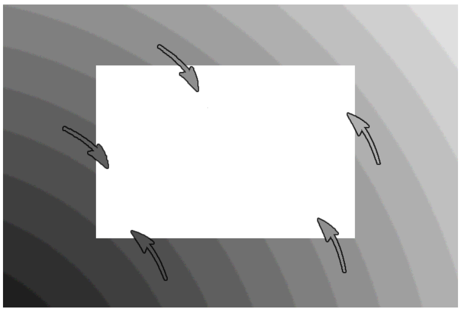


Figure 2.14: The Navier-Stokes method for inpainting propagates the intensity at boundaries along the so-called isophote lines. The direction of an isophote points towards the smallest intensity change.

This work deploys the so called Navier-Stokes method [36]. The method processes each color channel separately. The main idea of the Navier-Stokes method is to model image intensity in terms of fluid dynamics. The intensity flows from the boundary along the so-called isophotes. Isophotes are lines of equal gray levels. By continually propagating gray values along the isophotes it's possible to contract the hole. In section 4.1 I will describe how the method is used within the application and show a method for further speed up.

2.4.2 Multimodal Deep Learning

This section discusses different techniques for fusing multimodal inputs. While these techniques are generally applicable, the focus of this work is on color and depth input.

The previous chapter dealt with pre-processing steps to make depth data compliant for the use in CNNs. Finally different ways to fuse color and depth information are demonstrated. Basically there are two strategies: Late and early fusion [37]. In case of late fusion both modalities are processed and classified separately and finally fused typically by a weighted sum. Whereas only one classifier is learned in case of the early fusion scheme. Thus, the fusion already takes place in the feature space.

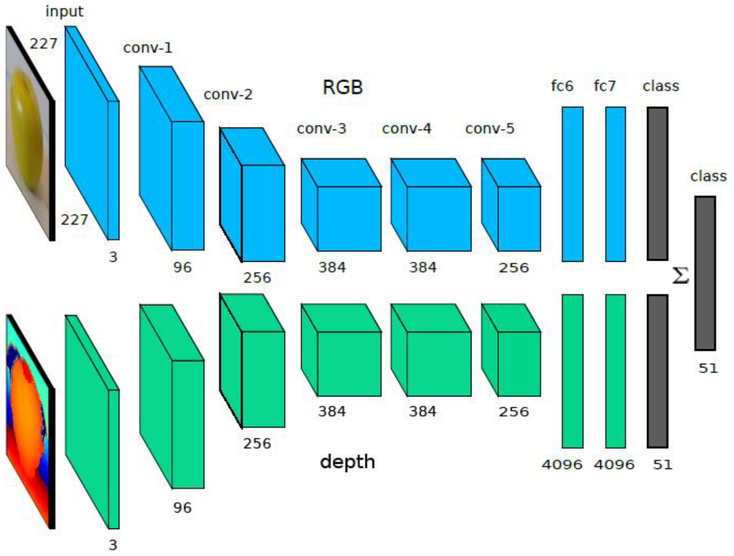


Figure 2.15: The two-stream convolutional network as it was used by Long et. al [21]. It follows a "late fusion" approach - the outputs of two classifiers are fused by a sum operation.

Long et al. chose the **"late fusion"** approach [23]. They evaluated teir two-stream FCN-architecture on the "NYU Depth Dataset" [38]. In a previous step they encode depth into a HHA-image as described before. Both inputs, namely RGB and HHA, are processed by two separate streams. Two independent classifiers are learned on top of each stream. The classification scores are merged by a pixelwise sum [23]. Although both streams are connected and learned jointly the strict separation until classification stage limits the freedom for co-adaptions. However, this is not always disadvantageous [39].

On the other hand this approach includes slightly more parameters and arithmetic operations (e.g. due to separate deconvolutions).

In contrast Eitel et al. follow the **"early fusion"** approach. Both streams are merged already in feature space. The "fc7" features (see figure 2.16) are concatenated, so that the following classification layer can access both feature sets simultaneously. This way the classifier can learn connections between both modes.

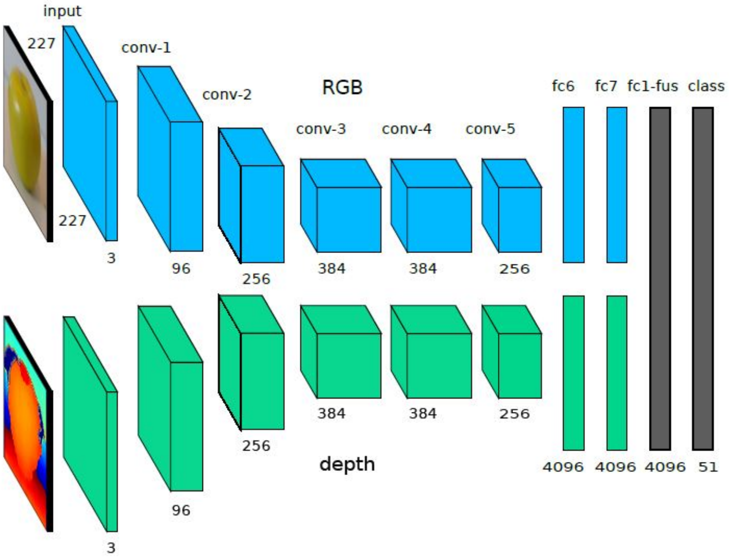


Figure 2.16: The two-stream convolutional network proposed by [35]. The feature-maps are fused by a learned layer. Classification is based on the merged output of fc1-fus.

In general it's not clear which strategy is the best. This is highly dependent on the individual case. Classification based on very dissimilar feature spaces usually works best with the "late fusion" approach [40]. Whereas early fusion can be promising for similar inputs (like RGB-D data). In practice it's therefore reasonable to test both variants.

3 Technical Approach

This chapter discusses the components, used for the technical implementation of this work.
I describe the main building blocks, consisting of hard- and software, required for the development and implementation of the system. Apart from describing them I emphasize the advantages, which have led me to the specific selection.

3.1 Kinect v2

Since its launch in 2010 the Kinect camera has become a popular tool for researchers in the field of computer vision and robotics [41]. The Kinect is an affordable and easy-to-use RGB-D camera. Color is perceived by a conventional RGB sensor. Depth is determined by an active measurement process. For this purpose the Kinect has a second image sensor and a light emitter.

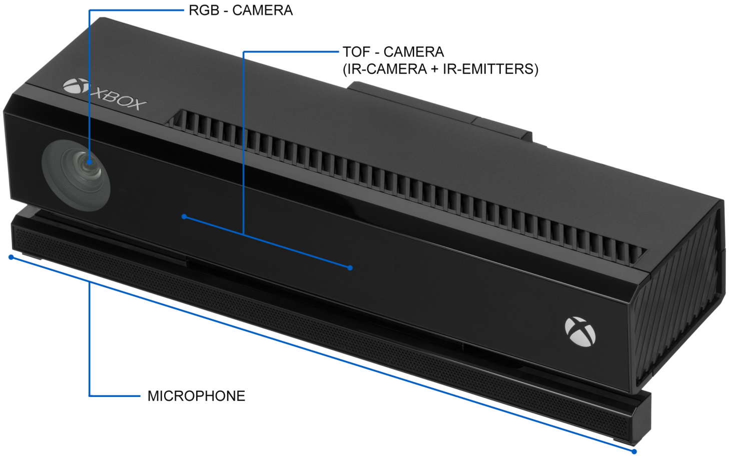


Figure 3.1: The second generation Kinect uses a Time-of-Light sensor for 3D depth sensing.

In this work the second generation of the Kinect is used. In contrast to the first version, which uses structured light for depth estimation, a **time-of-flight (ToF)** sensor is used. The main principle of ToF cameras is comparable to that of an echolot. A sensor receives light, which was previously emitted and then reflected by objects of the surrounding. The time between sending and receiving is directly correlated to the covered distance. The speed of light, however, is many times greater than that of sound. Light needs, for example, only about 33 picoseconds to travel a distance of one centimeter. Hence, the technical implementation of time measurement is difficult.

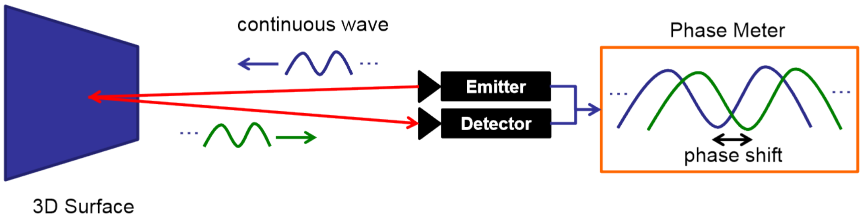


Figure 3.2: Illustration of the ToF principle (continuous wave) - The basic idea is to measure the time difference between emission time and received time. By using modulated light it's possible to determine the time difference by measuring the phase shift of in- and outgoing light waves. Since the amplitude repeats periodically, the result is ambiguous for phase shift greater 180 degrees. This fact limits the depth range.

There are direct and indirect ways to measure the time difference. The Kinect v2 uses the later. By using amplitude modulated infrared light it's possible to measure the phase shift between in- and outgoing light. The phase shift is proportional to the distance. By knowing the speed of light c, the modulation frequency f_m and the measured phase shift ϕ one can determine the distance d within a certain range (equation 3.1). The range is limited to the half of the wavelength $c/(2f_m)$, because the phase shift is only unambiguous in the interval between 0 and 180 degrees.

$$d(\phi) = \frac{c}{4\pi f_m}\phi \tag{3.1}$$

where:

$c = 299792458$ [m/s] (speed of light)

f_m [Hz] is the modulation frequency

ϕ is the phase shift

According to the formula $c/(2f_m)$, one might conclude that a small modulation frequency is ideal for obtaining a large depth range. Unfortunately, the standard deviation of the noise in the phase shift measurements is inversely proportional to the modulation frequency [42]. Hence, a suitable balance between depth range and accuracy needs to be found. In order to reduce the measurement noise and increase the range in which ϕ is unambigous, the Kinect v2 combines the results of 3 different frequencies - namely 16, 80 and 120 MHz [43].

Other difficulties, arise by optical effects, inaccuracies in measurement and background illumination. Further technical details can be found in [42] and [44].

The following table compares the most import specifications of both Kinect cameras: As shown in table 3.1 the resolution of Kinect v2 was increased for both depth and RGB-sensor. Moreover, the field of view is wider and the recent USB standard is used, which allows higher transfer rates.

Table 3.1: Kinect v1 vs. v2 - specs

		Kinect v1	Kinect v2
RGB-resolution	[px]	640x480 @ 30fps	1920x1080 @ 30 fps
D-resolution	[px]	320x240	512x424
depth range	[m]	0.4 - 4.5	0.5 - 5
method (D)	-	structured light	ToF
horiz. field of view	[deg]	57	70
vert. field of view	[deg]	43	60
USB standard	-	2.0	3.0

3.2 Caffe - A Deep Learning Framework

Caffe is recently one of the most popular deep learning frameworks. Other common libraries are Theano [45], Torch [46] and recently also Tensorflow [47]. Mainly for practical reasons I've decided to use Caffe. This work is mainly based on two preceding works [23] [29] and both implementations are dependent on Caffe.

The Caffe project was started by Yangqing Jia and the Berkeley "Vision and Learning" team [48]. The code is entirely open source and has an active community of developers. At the time of writing this thesis (Dec. 2015) the official repository on github has more than 4000 forks! Even big companies like NVIDIA are involved. NVIDIA distribute their own caffe-version [49] and a graphical interface called DIGITS [50]. With the help of cuDNN, a high performance library, developers can further boost the training of neural networks on GPUs [51].

Caffe addresses both efficiency and simplicity. The code is written in C++ and CUDA.

Furthermore there exist MATLAB, Python and command line interfaces. Net definitions are defined by plain text schema, which avoids hard coding and provides more flexibility.
Many well-known models are freely available and distributed throughout the "Model Zoo" [52]. Another advantage of Caffe is that developers can easily switch between GPU and CPU execution. Whereas the costly training is usually performed on high-performance GPU's, developers have the choice to deploy the neural network on more light-weight devices such as embedded systems.
Due to the active community Caffe is constantly evolving. The latest features are usually found in one of the many fork repositories. Some of them are included (after extensive testing) into the official version. Since this work requires some of the latest features I've not worked with the official master branch.

Used Hardware

All models presented within this work are trained on a single graphical processing unit (GPU) with 12 GB memory. Where available, the cuDNN implementations were used (e.g. pooling, ReLU, softmax). These are especially optimized for GPU usage. More details about the technical specifications can be found in table 3.2.

Table 3.2: Tesla K-80 board - specification [53]

Specifications	Tesla K-80 board
Number of GPUs:	2 x Tesla GK210B
Core clocks:	Base clock: 560 MHz
	Boost clocks: 562–875 MHz
Memory clock:	2.5 GHz
Memory size:	12 GB (per GPU)
Memory bandwidth:	240GB/s (per GPU)

Caffe with Spearmint

Spearmint is a software package to perform optimization tasks. It is designed to find an optimal set of hyperparameters, while considering the evaluation costs. For example, the amount of hidden units of a neural network has a considerable impact on the evaluation time.

Spearmint with Caffe brings this functionality to Caffe. It enables a comfortable way to optimize any kind of hyperparameters - such as parameters required for net definition and solvers. Instead of specifying a fixed value for a parameter, a certain range can be specified. Spearmint searches then for the optimal values in those ranges by using Bayesian optimization methods [54].

3.3 Robot Operating System (ROS)

The Robot Operating System (ROS) is an open-source framework for developing robotic software [55]. The design is following the philosophy and beliefs of an easy-to-use robot framework. The developer postulated the following requirements:

- peer-to-peer connections
- tool-based
- multiple languages support
- modular
- free and Open-Source

The basic ingredients for a ROS application are nodes, messages, topics and services. A **node** is a software module typically arranged within a graph of peer-to-peer communications. The communication between nodes is based on interchanging **messages**. A message is published to a distinct **topic** and can be consumed by multiple nodes. All messages are shared via immutable, reference-counted smart pointers. This allows safe access from multiple processes, prevents unnecessary copy operations and is developer-friendly, since all pointers are automatically deleted.

Furthermore ROS provides filter algorithms for synchronizing multiple messages based on user-defined policies. For example one can either force equal time stamps for synchronization or use a more relaxed condition - e.g. synchronize messages which are chronologically close to each other.

In contrast to this publish-subscribe models a **service** follows a request-response scheme and is therefore more appropriate for synchronous transactions. The interface of a service requires a fixed request and response message-type.

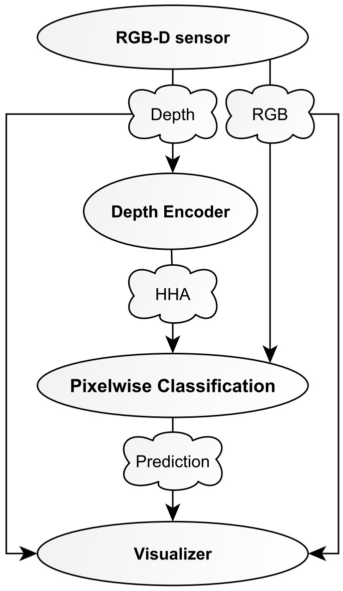


Figure 3.3: Exemplary ROS graph for semantic segmentation from RGB-D data. The nodes, represented by oval shapes, listen for certain message types ("clouds"). Some of them require more than one input. Each node publishes its outputs independently. Therefore it's important to synchronize all input messages in a suitable way.

An possible communication pipeline for the segmentation task is shown in image 3.3. The RGB-D sensor provides color images and depth information. A "bridge" component publishes the images and pointclouds to the ROS ecosystem. ROS nodes can then register for these message events. For example, the "Depth Encoder" listens for pointclouds. The results are then published again and received by the "Pixelwise Classification" node. Details about the actual implementation are not relevant for the usage of the interface. In my application, this node wraps a FCN based on Caffe. The segmentation result can be visualized or used for several other tasks.

In addition to the modular software architecture and the communication processes ROS provides some handy tools for various purposes. For this work I've used for example a tool called "rviz". Rviz is a visualization tool for 3D and 2D data. One can simply register for certain topics. The received contents e.g. pointclouds are then visualized continually.
Very useful for developing ROS applications is the opportunity to record and playback

certain messages. The messages are stored in so-called ros-bags. In this way one can simulate a live-system, while developing components.

ROS intends not to offer an all-in-all solution, instead it is a very general framework for robotic applications. The diversity of robot applications requires an easy expandability. One can expand ROS through the integration of other libraries. For example many applications have to deal with 2D and 3D sensor data. Therefore, ROS supports an easy integration of OpenCV and pcl functionalities. I will introduce these libraries in the following.

3.4 Point Cloud Library (PCL)

The Point Cloud Library is a famous Open Source project, which includes data-structures and algorithms mainly for the processing of 3D data. It is released under the BSD license and is freely available for commercial and educational usage [56][57]. PCL can be run on different platforms such as Linux, MacOS, Windows, Android and iOS.
The high dimensionality of the data requires specific and optimized solutions to enable efficient processing. PCL provides several implementations of state-of-the-art algorithms, capable for filtering, model fitting, feature extraction, reconstruction, segmentation and registration. As shown in figure 3.4 the code is organized in smaller, independent libraries.

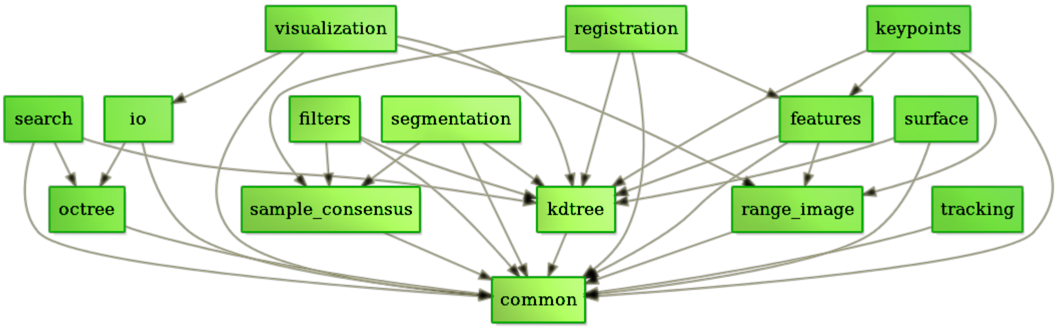


Figure 3.4: The Point Cloud Library consists of small, independent C++ libraries.

Similar to ROS, PCL relies on other open-source libraries, such as Eigen and Boost. Eigen provides efficient linear algebra operation, whereas Boost shared pointers are used to pass data around without the need of expensive re-copying. In addition, many operations can be executed in parallel on multiple cores.

In this work, I've mainly used the data structures and special algorithms for estimating surface normals and plane fitting.

3.5 Open Source Computer Vision (OpenCV)

OpenCV is an open-source library for computer vision applications [58][59]. By providing efficient building blocks capable for real-time applications, it has been extensively used within various types of applications.

The roots of the OpenCV library come from an Intel Research Project launched in 1999. The functionalities have been developed in C/C++. Meanwhile OpenCV is published in version 3.1. In addition to many image processing and computer vision algorithms, more and more machine learning algorithms have been added.

Recent algorithms are now developed in the C++ interface. There exist bindings in Python, Java and MATLAB, which certainly contributed to the extensive usage in research and development. OpenCV is cross-platform, which means that it can be employed on most desktop and mobile platforms.

OpenCV supports hardware-dependent execution for acceleration. For example some operations can be performed on graphical processing units (GPU).

The wealth of functionality is not that relevant for this work. OpenCV was mainly used to organize and display images, but also for recovering missing image parts (inpainting).

3.6 Image Annotation Tool

Supervised learning requires an adequate amount of annotated samples. Since the aim of this work was to perform classification at pixel level, every single pixel has to be labeled. Compared to annotating at image or object level, this is particularly time-consuming.

To facilitate this work I've used a browser-based tool, which is freely available at github [60]. The tool is implemented in plain Javascript. Users can create segmentation maps by clicking onto highlighted regions.

All settings, e.g. class definitions and image-paths, are stored in a JSON file. Classes are mapped to integer-labels organized from 0 to number classes−1. This is already the same format as Caffe expects.

The regions are determined by a clustering algorithm called SLIC [61]. Cluster sizes are adjustable via the graphical interface. Furthermore the tool provides several visualization features and an image export functionality.

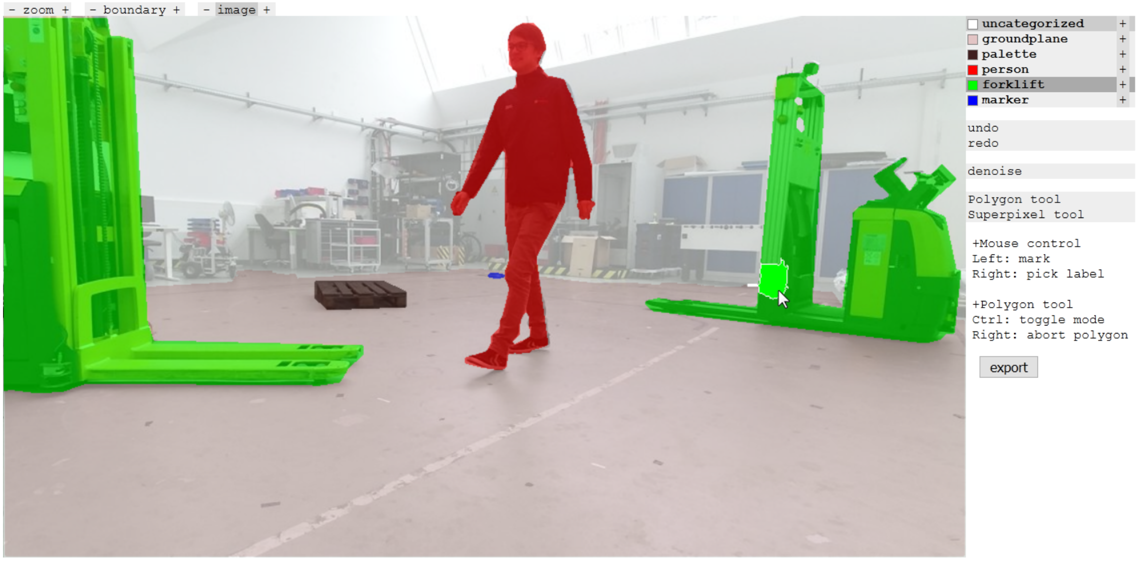


Figure 3.5: Screenshot of the browser-based annotation tool. The annotations are visualized by a colored transparent overlay. Hovered regions are highlighted accordingly. Moreover users can select from various settings for controlling appearance and tool properties.

The SLIC superpixels become very handy for fine details such as hairs. However, especially for man-made objects with straight boundaries (e.g. palettes) a simple polygon-tool is more suitable. Hence, I've extended the tool by this functionality. In addition I've added keyboard shortcuts to accelerate the labeling process. My extension was merged into the original repository [60].
Since the tool is browser-based it's suitable deploy it for large scale projects, where several workforces operate in parallel. Instead of exporting the image to a local disk, the result could be saved on a central server.

4 Implementation and Results

Having constructed the theoretical and technical basis, I will now explain important implementation details. In addition I present and discuss the achieved results. First of all I give an overview about the general conditions. Which data was used? What was challenging about the data? What special arrangements have been made?
Then I subsequently introduce working solutions and evaluate their performance with regard to different metrics. In this way I built a basis for selecting the final net architecture.
The starting point is the base net, which corresponds to the architecture presented by Long et al. [23]. An important aspect of this work are possibilities to facilitate the learning process. Instead of performing a stage-wise training, I was able to do this in one pass (see section 4.4.1). This is either done by selectively "transplanting" weights from an existing model or by introducing a normalization step.
After that I will highlight further important aspects. How is it possible to improve the spatial accuracy? In particular I present the concrete use of the skip-architecture (introduced in section 2.2.3) and a joined conditional random field (see section 2.3). Then I elaborate on strategies for incorporating depth-information (see section 4.4.3). Finally, the results of the preliminary tests are used to design an appropriate training methodology and net architecture. In addition I demonstrate how to deploy the network by embedding it into a robotic system. In this way other components can leverage from the segmentation results. This feature enables a variety of future applications or the improvement of existing ones, such as path planning or decision making.

4.1 Data

The supervised training requires pixel-wise annotations per image. In addition, annotations are needed for both validation- and test-set. I've defined object categories suitable for a range of potential applications within a warehouse environment. However, the required categories depend on the exact use case. For instance, the localization of the autonomous forklift, developed by the Bosch Start-Up, is based on artificial markers. These markers must be detected for this purpose. Therefore the "marker" class was included into the class set. To my knowledge there aren't any public-available data sets which satisfy all the requirements:

- RGB-D images
- pixel-wise annotations
- warehouse environment
- object categories:
  - groundplane
  - pallet
  - human
  - forklift
  - marker
  - void/background

Therefore it was necessary to build a new image set. All images were recorded by a Kinect v2 camera (see section 3.1) mounted on a trolley. In this way, an approximately constant height above ground was ensured. This is important for the processing of the depth-encoded image.

A tool developed by the ETH Zurich was used to record the pointclouds and RGB-images in rosbags [62] [63]. Although this rosbags are not required to train and test the model, it becomes useful to simulate a live-system, which is actually the final objective of this work. A rosbag allows to publish recorded sequences on demand, which gives perfect conditions for testing the final system. Every ROS message includes a header which contains meta-data such as a timestamp. The timestamp is important for the application, since it consumes two kind of input types - namely pointclouds and images. These are provided by different sensors. In general it is not guaranteed that both messages are published at the same rate and times. Thus, the messages must be synchronized. Fortunately ROS provides this functionality.

But before the actual deployment, the model has to be trained and tested offline on plain images. First I've selected suitable samples, which are then pre-processed and saved to disk. I've developed an interactive tool for this purpose. The tool iterates through a rosbag, synchronizes the desired message types approximately by their timestamps and displays the color images. On this basis the user can decide, which samples are further processed (see HHA depth encoding 4.1) and stored. In this way image pairs are obtained, which represent color and depth information.
Next, the data must be annotated. All images were labeled with the help of a browser-based tool (see section 3.6). The resulting triplets of RGB-, HHA- and annotated images are then used for training, validation and testing.

Dense labeling is very time consuming and makes it hard to build a huge dataset with little manpower. I've searched for possibilities to extend my own database by external images. As already mentioned there are no public RGB-D datasets that meet the requirements. However, the chosen net architectures allows to (pre-)train depth- and color-models separately. Hence, it's possible to exploit the availability of large RGB-datasets at least for the color-based model.
In addition to my own dataset I use parts of the Pascal VOC 2012 [64] and SUN database [65]. To meet the requirements, those labels were mapped on the defined label set. Moreover, only images showing at least on of the defined objects (excl. void and groundplane) were included. Other images may (it's a randomized selection!) serve as a background image for data-augmentation (see section 4.1).

Table 4.1: Number of RGB-D images in the randomly splitted train-/val- and test- set.
The number of RGB images is shown in parenthesis.

	train	validation	test
quantity	100 (1047)	20	30

The actual splitting of the available annotated images is shown in table 4.1. For the sake of the model quality I've chosen very small sizes for the test- and validation set. This is a crucial point for criticism. However, the construction of a larger data set was impossible in the scope of this work. Therefore, the results must be interpreted with caution. The presented results should be viewed as a rough indication.
As you can see in table 4.2 the number of samples (i.e. pixels) for each class are very imbalanced. My experiments have shown that this leads to very biased results. The

Table 4.2: The pixel-wise class frequencies in percent [%]:

	train	validation	test
groundplane	32.8	36.3	38.7
pallet	4.3	3.8	3.2
person	2.2	1.5	6.0
forklift	4.5	1.3	10.3
marker	0.9	1.6	0.2
void	55.3	55.7	41.7

model tends to focus on the dominating class - hence "groundplane" and "void". This is primarily problematic for the initialization phase.
In general there are different strategies to counteract class imbalance. I will explain my thoughts and attempts in section 4.1.

Depth Encoding (HHA)

As described in section 2.4.1 the pointclouds must be transformed into a meaningful image representation. One way to do this is the so-called HHA encoding (see example image in figure 4.1(b)).
My implementation loosely follows the HHA depth encoding scheme described by Gupta et al. [19]. Their Matlab implementation expects depth images as input. Thus, the 3D data is already encoded and must be decoded again. In contrast to them, I use pointclouds as input.
Fortunately the pointclouds are in an "organized" format. This means the points are stored in the same order as the image. Therefore the mapping between the pointcloud and the image is known.
The depth encoder was implemented as a templated class in C++. A pointcloud is expected as input. Besides the x,y,z coordinates this pointcloud must provide surface normals for each point. This extra information is required for the "A" component. Thus, the local surface must be estimated for each point, which requires the incorporation of the neighborhood. This makes it a potential bottleneck of the processing pipeline. An efficient method makes use of integral-images [66]. An implementation of this algorithm exists in the PCL library and was used in this work.
The "extract" method of the depth encoder returns a three channel image in PCL

format ("PCLimage"). The calculation of each component has been implemented as follows:

- **"height over ground" (H)**
  This component is calculable as a per-element-operation. For each point the distance to the floor must be computed. For simplifications I've assumed a static model of the groundplane. Such a plane can be described in its general form:

  $$ax + by + cz + e = 0 \tag{4.1}$$

  where:

  a, b, c and e are the model parameters
  x, y, z indicate the coordinates of a 3D point

  Computing the distance d between the plane and point (x_0, y_0, z_0) is then straightforward:

  $$d = \frac{ax_0 + by_0 + cz_0 + e}{\sqrt{a^2 + b^2 + c^2}}$$

  The assumption of a static plane is reasonable, since it's described w.r.t. the vehicles local frame and the surface is approximately flat. As long as the camera pose remains fixed w.r.t. the vehicles local frame, the model remains also valid. Minor errors caused by vibrations or small bumps were neglected.
  A more sophisticated approach would include steady realignment by incorporating knowledge about the orientation (e.g. provided by an accelerometer). Another approach would be to frequently re-estimate the plane.

  In this work the groundplane was estimated only once at the beginning of each sequence. This was possible since the camera was mounted at a fixed height. The estimation process requires to manually select a region, which consists of a part of the floor (in the image plane). However, this process can be fully automated. For example, a detected marker on the ground can be used for automatic field selection.
  The selected region allows to extract the underlying subcloud. This selection is a good starting point to fit a plane.
  Robustness in the presence of outliers is assured by using a RANSAC-scheme [67]. RANSAC is a non-deterministic algorithm for estimating model parameters out of measured data. To determine the parameters, a random sample of minimum size is drawn. In the case of a plane, exactly 3 points are needed to

determine a unique solution. It is then checked which points of the remaining set support the model assumption. A point is considered as a valid supporter if it's described by the model within a certain tolerance range. This parameter should be chosen depending on the expected noise. The resulting set of points is the so-called "consensus set". This procedure is repeated until a predefined termination criterion is reached (e.g. a maximum number of iterations). After termination the largest consensus set is chosen for estimating the final model. Finally the height values in range [0,5] meter were linearly mapped onto the 8bit domain. The decision for this certain range is based on the assumption that the objects will occur only within the defined range. Ideally, the optimal range should be determined experimentally.

- **"horizontal disparity" (H)**
  Since a ToF camera is used and not a stereo-camera the term "disparity" might be misleading. In fact a defined stereo rig is imitated. For the mapping from depth to disparity one has to define the baseline B and the focal length f. Fulfilling these prerequisites, the z-values (i.e. the point-wise distance to the sensor) can be mapped by the given formula:

  $$d = \frac{B \cdot f}{z} \tag{4.2}$$

  where:

  B is the baseline between two cameras
  f is the focal length
  z is the depth value (along the optical axis)

  Suitable values for B and f were determined empirically. In combination with a linear mapping from [9, 60] to 8bit domain, I found 120 a suitable value for $B \cdot f$.

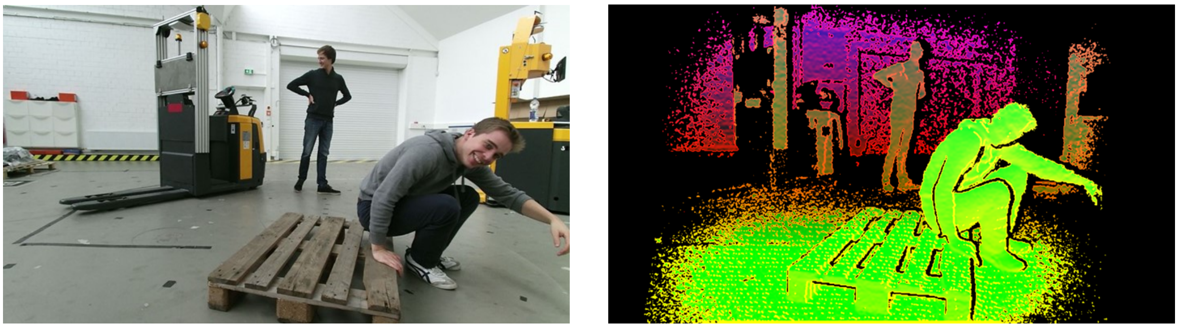
- **"angle" (A)**
  This value describes the enclosed angle α between the surface normal $\vec{n}$ and the gravity vector $\vec{g}$. In this work, the gravity vector was assumed to be fixed. This assumption is based on the fact that the camera system is mounted on a trolley, which moves perpendicular to the normal of the groundplane. This surface normal points approximately in the same direction as $\vec{g}$. The enclosed angle α between $\vec{n}$ and $\vec{g}$ is obtained by equation 4.3.

$$\alpha = \arccos\left(\frac{\vec{n}\cdot\vec{g}}{|\vec{n}||\vec{g}|}\right) \tag{4.3}$$

$\vec{g}$ gravity vector (approx. the groundplane normal)

$\vec{n}$ surface normal

α enclosed angle i.e. orientation of the surface



(a) RGB

(b) HHA

Figure 4.1: An exemplary image pair used for training - The left image shows a conventional RGB image. Depth (right) is encoded into three stacked representations (height, horizontal disparity, angle). Black pixels represent missing values. These are filled by inpainting in the next processing step (see next section).

Inpainting

Depth images usually include areas with missing values. There are several causes for this phenomenon. In the case of a ToF camera, the emitted light is sometimes reflected or absorbed. Moreover ToF cameras have a limited working range (described in section 3.1).

In order to fill these holes, I've applied an inpainting-method. An efficient algorithm is the Navier-Stokes method [36].

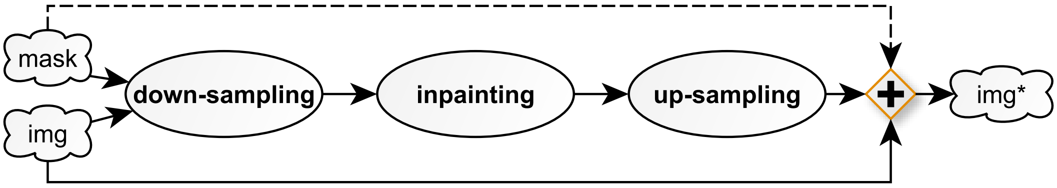


Figure 4.2: Scheme of an efficient inpainting pipeline. The computational intensive inpainting algorithm is performed on a downsized version of the image.

A further performance boost is achieved by down-sampling the input image before the actual inpainting is applied. I have used a scaling factor of 0.25. Inpainting is performed within a radius of 3 pixel. The resulting image is up-sampled again and used to fill the holes of the original image. This way valid values of the original depth image are kept untouched.

Data Augmentation

Deep Neural Networks are capable to represent complex models. The downside is that complex models are more prone to overfitting. Because of that it's necessary to apply regularization techniques and train on a huge training set. Fortunately the required amount of training data is significantly lower, when finetuning the model from an existing one. However there is no general rule of thumb how many images are actually needed. Many factors affect this question. For example, the similarity of source and target task plays an important role [17].
In addition, previous works have shown that a good performing model can be further improved by providing more training data. It's therefore common to combine different data sets to enlarge the training set. For example, Zheng et. al [29] use a combination of Pascal VOC and Microsoft COCO dataset [68].

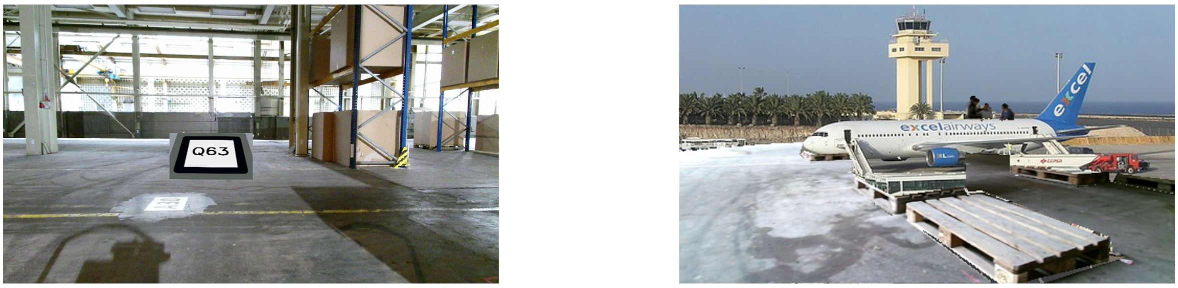
Furthermore, augmentation techniques can help to make the most of the available annotations. By altering the samples the model can be trained towards a more general solution. In this way the model becomes more robust and invariant. Of course, the applied augmentation techniques must fit to the requirements of the system. For autonomous vehicles one has to consider very different conditions since the point of view and other conditions are constantly changing. In addition objects are moving around and occlude each other. Therefore it makes sense to apply a broad range of image transformations:

- mirroring
- scaling
- cropping
- rotations
- contrast/color "jitter"

These techniques are widely used. Furthermore, the database was augmented by adding new content to the images.

- One way to do this is to combine different images. The annotations are useful to selectively paste certain regions into other images.
  Many images in PASCAL VOC 2012 contain no objects of my predefined classes. Consequently, the entire image counts as part of the "void" class. I've used these images to modify my own images by replacing regions marked as "void". Since "void" can be anything, it's important that the samples of "void" include a lot of variance. Unfortunately, the recorded data set contains only three different scenes. Consequently, the background often looks very similar. Therefore, this technique is particularly useful.
  Actually, the background image can come from a different context. The aim is that the network learns how to distinguish between classes regardless of the environment. An exemplary image can be seen in figure 4.3(b).
  In addition I've used this "cut and paste" technique in order to randomly transfer certain objects. Underrepresented classes were preferred - i.e such classes had a higher probability of being chosen.

- As shown in table 4.2, the marker class is particularly underrepresented. I have therefore developed an augmentation technique for this class. Since the appearance of a marker follows a certain pattern, it's possible to generate an "artificial" marker. First, an outlined white box with a randomly chosen naming (e.g. "Q63") was generated. Then a randomly chosen image transformation was applied. This included scaling, rotation, perspective transform and brightness or color adjustments. Finally, the resulting image was pasted into the target image.



(a) marker

(b) image merging

Figure 4.3: Besides standard augmentation techniques, such as mirroring or rotations, I've also altered the image content itself. By generating markers (a) or merging two images (b).

Class Balancing

Class imbalance is a prevalent hurdle for many machine learning applications. Sampling real-world measurements often leads to imbalanced datasets. Per-pixel labeling even promotes this issue, since different objects usually appear at different scales. There is no general rule at which point class imbalance becomes problematic. Because of that it's sufficient to supervise the learning process. While training, I've periodically checked the per-class performance (pixel accuracy). Doing so, it turned out that the model neglected underrepresented classes. Especially crucial for the training success is the initialization phase. In order to prevent this unwanted effect, I've tested two kind of approaches.

First I've implemented a custom loss function E which accounts the class frequencies of a training batch (see equation 4.4). Basically a multinomial logistic loss is used, where each sample is weighted by the inverse class frequency f_l. This way each class (with label l) within a training batch of size N contributes equally.

$$E = -\frac{1}{N}\sum_{n=1}^{N} f_l^{-1} \cdot log(p_{n,l}) \tag{4.4}$$

where:

N is the number of samples (i.e. pixel in a batch)

p is the class probability of label l

f_l is the frequency of label l occurring in a batch

An advantage of this approach is that all training samples can be retained. On the other hand this rule is often too hard. A single pixel of one class may have the same influence as thousands of pixels of another class.

The second approach compensates class imbalance by omitting training samples of overrepresented classes. This was done by randomly setting labels to a "ignore" label. The fraction of ignored labels has been manually set for each class. The setting was approximately based on the class frequencies. Table 4.3 shows the class frequencies after balancing:

Table 4.3: Class frequencies after balancing in percent [%]. The original frequencies are shown in parenthesis.

	train	validation	test
groundplane	24.9 (32.8)	36.3	38.7
pallet	15.7 (4.3)	3.8	3.2
forklift	8.4 (2.2)	1.5	6.0
human	16.2 (4.5)	1.3	10.3
marker	3.4 (0.9)	1.6	0.2
void	31.5 (55.3)	55.7	41.7

4.2 Evaluation

As already mentioned I've evaluated the final and interim results on a relatively small test set. It's therefore appropriate to interpret them carefully. This fact also applies to the error metrics. There is no perfect metric to describe the semantic segmentation in a comprehensive and accurate way. In the next section I'll broach this in more detail. I've developed a tool which helps to evaluate a specified model. By a pixel-wise comparison of the segmentation result and the corresponding ground truth different error metrics are calculated. It should be noted that the ground truth data, which was created by a human, is erroneous as well. Finally, all measurements were saved to a csv (comma-separated-values) file.

4.2.1 Metrics

This section discusses different possibilities to measure the segmentation accuracy. In order to build a comparable basis I will focus on the same metrics as described in the work of Long et al. [23]. The authors use four variations on pixel accuracy (PA) and intersection over union (IU).

Especially the mean intersection over union (mIU) is a standard method for segmentation evaluation. For example the Pascal VOC challenge uses this measurement since 2008.

However, all metrics have different pros and cons. One major drawback of all metrics used in [23] is that the accuracy nearby object boundaries isn't sufficiently reflected. Therefore this work uses an additional variant, the so called trimap.

Next, I will state the mathematical definitions and discuss the pros and cons of each metric. All metrics are based on the confusion matrix C. C_{ij} is the count of all pixels annotated as class i and predicted as class j. For example C_{ii} is the number of true positives (TP) for class i. Further conventions are the total number of class instances t_i and the total number of classes n_{cl}.

- **pixel accuracy (PA)**:
$$\frac{\sum_i c_{ii}}{\sum_i t_i} \tag{4.5}$$
The overall per-pixel accuracy measures the ratio of correctly classified pixels compared to the total number of samples (true positives). A disadvantage of the metric is that it's strongly influenced by class imbalance. A classifier betting always on the majority class can achieve high performance rates. Here, the "void" class covers approximately 55 percent of an image. Thus, a high PA score does not necessarily mean that the accuracy is acceptable for all classes.

- **mean pixel accuracy (mPA):**
$$\frac{1}{n_{cl}} \cdot \frac{\sum_i C_{ii}}{t_i} \tag{4.6}$$
First the pixel accuracy is evaluated for each class separately. Next the average over all classes is taken. This way it's possible to lower the impact of class imbalance, but the risk of overvaluation remains. A classifier, which correctly classifies the region covered by an object but tends to include some pixels of the surrounding, will achieve a respectable score since mainly one class suffers from this behavior - namely the background class. The impact of this effect increases as the number of classes increases.

- **mean IU (mIU):**
$$\frac{1}{n_{cl}} \cdot \frac{C_{ii}}{(t_i + \sum_j C_{ji} - cii)} \tag{4.7}$$
This metric measures the intersection over union of the predicted labels of each class and forms the average. The advantage is that the mIU considers both the false positives and the false negatives. This solves the issues described before. For that reason it is nowadays the standard metric for semantic segmentation. Nevertheless it does not explicitly measure how precise the segmentation boundaries are.

- **frequency weighted IU (f.w. IU):**
$$\frac{1}{\sum_k t_k} \cdot \frac{\sum_i t_i C_{ii}}{(t_i + \sum_j C_{ji} - cii)} \tag{4.8}$$

The frequency weighted intersection over union again targets the overall accuracy. Underrepresented classes receive a smaller weight than others. In particular this means for this case, that mainly the accuracy of the classes "background" and "floor" are considered. Such a measurement is not interesting for this application and therefore wasn't considered.

- **trimap:**
  The trimap was introduced by Kohli et al. to evaluate the accuracy at object boundaries [69]. By defining a narrow band around object outlines (see figure 4.4) the evaluation can focus on the results nearby class transitions. For my experiments I've used a width of 20 px. This width has to be chosen very carefully. As most inaccuracies in the ground truth data occur at object boundaries a too narrow band would result in an insignificant measure. On the other hand a too wide band barely says something about the spatial accuracy. One solution to overcome this limitation is to plot the accuracy over the trimap widths [69]. Since I have chosen a very moderate width, I've considered only one for this work. Moreover, the trimap metric was always considered in the context of other metrics. Both the mIU and the mPA was evaluated within trimap band.

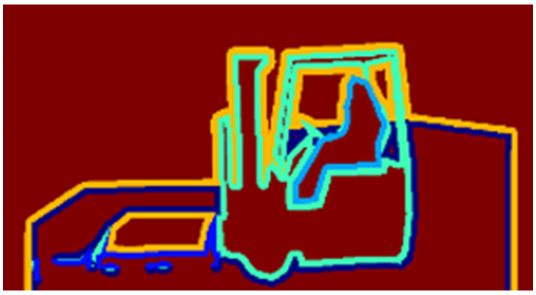


Figure 4.4: An examplary trimap mask. The metrics are evaluated within narrow bands. In this illustration areas marked in red are ignored for the specific evaluation.

The most general metric is probably the mean intersection over union (mIU). The trimap allows to give a statement about the accuracy nearby boundaries. PA and mPA are still very common and can be found in many publications.

4.3 On the Base Network: Model Definition and Training

There are many degrees of freedom while designing a deep neural network. For this reason, but especially because of the immense cost of training, a design by trial and error is impractical. In addition, the complexity and high dimensionality of the parameter space hinders to grasp a neural network. Although one can describe the system mathematically, it appears more or less like a "black box".
The experiences and results of other researchers are important to determine a good starting point. Mainly based on the work of Long et al. [23] and Zheng et al. [29], various experiments were carried out in order to find a way to improve the system in general or to adapt it to certain circumstances.
But first I give a short introduction to the basic setting, especially the net architecture. Both works [23] and [29] built on top of the results of Simonyan and Zisserman [5]. In their work "Very deep convolutional networks fo large-scale image recognition" they describe network architectures with up to 19 weight layers. The research groups of Long and Zheng used the pretrained model of the 16 layer version visualized in figure 4.5 (downloadable at the modelzoo [52]).

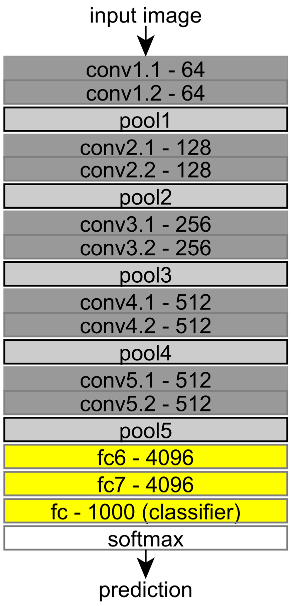


Figure 4.5: An illustration of the "VGG16" net-architecture [5]. The network consists of convolutional ("conv"), pooling, ReLU and fully-connected ("fc") layer. Each layer has a name and a number, which indicates the number of channels. Please note that (although not explictly shown) each "conv"-layer is actually followed by a ReLU-layer.

This network is known as **"VGG16"**, as it was developed by Oxford's "Visual Geometry Group" and has 16 weight layers. Figure 4.5 illustrates the topology of the network.

Weight layers include the convolution ("conv") and fully-connected layers ("FC"). All-in-all this net consists of 138 million trainable parameters.
"VGG16" is organized in groups of convolution layers, each followed by a max-pooling layer. Max-pooling is performed within a 2x2 window, with stride 2. As "VGG16" includes five pooling layers this leads to an overall downsampling factor of 32. Every "conv" layer goes along with a non-linearity (more precisely a ReLU) and has a very small receptive field of 3x3. As several "conv"-layers form a group, the effective size of the receptive field increases. Three successive layers cover together a 7x7 region. Compared with a single layer covering a 7x7 field these three layers together have approximately 45% less parameters, but include more non-linearities. Besides the reduced memory footprint, less parameters are advantageous as it regularizes the network. In addition, using more non-linear transformations makes the decision function more discriminative [5]. As a result, the VGG-team achieved remarkable results in the ILSVRC-2014 competition: Second place w.r.t. the classification score and first place w.r.t. the localization score [70].
The data set of ILSVRC-2014 includes a total of 150k images, where 100k images serve for training. There are 1000 object categories.

The pretrained "VGG16" network was "convolutionalized" by Long et al. - i.e. the fully-connected layers were replace by convolutions with a kernel size of 1x1. Such a "fully-convolutional" layer results in a linear transformation of the input channels. Rather than producing one dimensional predictions, this leads to two-dimensional predictions i.e. the segmentation maps. As already described in section 2.2.3, the upsampling of the segmentation maps is realized by a skip-architecture. The model was finetuned on 1464 fully annotated images (PASCAL VOC 2012 challenge) [64].

As introduced in section 2.3 Zheng et al. improved the spatial accuracy by appending a graphical model (more precisely a CRF). The finetuning was carried out with an enlarged PASCAL VOC dataset. Together with a subset of MS COCO [68] they ended up with a huge set consisting of more than 77k images.

This summary has shown the history of the base model and its evolution over time - from classification to fine-grained segmentation. This is where my work built upon. All network variants, which I will introduce in the next sections, were optimized with SGD and momentum. For the same reasons as Zheng et al. I use one image per batch. The memory limits of the GPU, the overall net size and the usage of large input images (768 x 432) enforced this. In order to handle this extremely small batch size Zheng et al. use a small learning rate (10^-13 for the unnormalized loss) and a high momentum of 0.99. This means that a relatively high fraction of the previous weight update is added to the current one (compare to section 2.2.1). Thus, congruent weight updates are amplified, successive updates with different directions are smoothed out. In other words, the learning "history" has a higher impact on the current learning. As momentum potentially increases the step size, one must reduce the learning rate accordingly.

Although this approach worked well, I've used an alternative approach. Instead of using an extraordinary high momentum, I've deployed gradient accumulation over 10 iterations. This approach tended to more stable training curves. However, the pros and cons of both settings were not examined in detail.

Regularization techniques were applied like described in the original "VGG16" net [64]. This means a dropout operation of ratio 0.5 was performed after the fully-convolutional layers ("FC") and the weight decay was parametrized by an L_2 penalty multiplier of $5 \cdot 10^{-4}$.

4.4 Preliminary Experiments

Based on the previously described base network and settings I've conducted several preliminary experiments. I elaborated on the following three main aspects:

- facilitating the training process
  (especially in the critical initialization phase)
- improving spatial accuracy
- fusing color and depth information

I present the most important results in the corresponding sections. A summary of all results can be found in the appendix (see table A.2).

Based on the experimental results I have designed and trained the final network, which I present in section 4.5.

4.4.1 Initialization Phase

The success of training deep neural networks is very dependent on the weight initialization and selection of hyperparameters. Due to the "vanishing gradient" problem it's hard to train a large net from scratch. Therefore it's a common practice to train large networks in stages. This means that developers start the training with a shallow network and increase its size gradually. For example the "VGG16" network was trained in this way [5].

Building a deep neural network can be a costly process. For this reason it's very helpful to initialize from a pretrained model. This process is called "finetuning". Although, I make use of this technique a lot of difficulties remained. Due to the skip architecture the network intermediately splits into three streams and reconnects again. This DAG (directed acyclic graph) structure complicates the training procedure and stalls the learning. Hence, the training is likely to get stuck in a bad local optima. I have confirmed that in experiments (see figure 4.6). Particularly in the case of very unbalanced training examples, the network will initially focus on the majority class, which is indeed an unfavorable local minimum. I have already made arrangements (e.g. class balancing), but the distributions have not been fully adjusted (see table 4.3). Many factors influence whether the optimizer will find a way out of an unfavorable minimum. Major factors are the learning rate, momentum, batch size as well as the quality and distribution of the training samples.

Due to hardware limitations, a very small batch size had to be chosen, more specifically only one image per batch. This extreme condition requires a very small learning rate and a high momentum or alternatively the usage of gradient accumulation. Both techniques have some kind of smoothing effect on the learning process, which promotes a holistic result. In turn this makes it difficult to escape from a local minimum. An ideal configuration of all parameters is difficult or sometimes even impossible to find. At least the search is very time consuming.

Long et al. solve this issue again by stage-wise training [23]. In this way, the amount of untrained parameters is kept small. The reduced complexity facilitates the learning process, which is especially crucial for the first iterations.

Although this is a practicable approach, it's desirable to train a network at once. Ideally, one would like to reduce the manual effort to a minimum. Because of that I've searched for ways to support the learning process, especially for the first iterations.

Finally I came up with two solutions for training a branched FCN network at once. The first solution is dependent on the class definitions of the source and target network, whereas the second can be used universally.

Extensive Weight Transfer

Large models are often seen as black boxes, they are hard to understand and to visualize. On the other hand several works have shown that the early layers tend to capture very general features. Those are well-suited for the reuse in other models.
An effective approach to tackle new deep learning problems is therefore to transfer the learning from one model to another. Commonly all weights of the feature extraction part are transferend and a new classifier is learned on top of it. This works well even for very distant tasks [17]. If the dataset is large enough, it's possible to finetune the entire model with a low risk of overfitting. Usually the benefit of transfering weights is still measurable - even after many learning iterations [17].
For this work, the transfer of learning is even indispensable. The amount of training data is insufficient to train a robust deep network from scratch. Moreover my experiments have shown that the transfer of pre-trained classifier weights can help to train a skip architecture from scratch. In addition, this leads to good segmentation results within few iterations.
In general, it's not possible to transfer the weights from one classifier to another, as usually the number and ordering of the outputs (number of classes) differ. Thus, the transfer is not supported by the standard routines of Caffe. Fortunately, one can directly manipulate the model parameters through the Python interface. This way I've selectively copied the classifier weights of identical or similar classes.
Since a model, trained on the PASCAL VOC2012 dataset, was used for this purpose, it was possible to directly take over the weights corresponding to "person" and "void". For other classes, not contained in the dataset, I built pairs of visually similar objects. For example a "palette" shares to some extent the appearance of a "table". Both objects usually have a wooden texture and consist of horizontal and vertical elements. Weights associated with the "forklift" class were initialized by the "car" class. Weights corresponding to "ground" and "marker" were initialized from random variables generated by a xavier filler with a standard deviation of 0.1 [12].
Figure 4.6 shows the learning curve for a model partially initialized from a pretrained model compared to the previously described "full" initialization.
If the classifier weights are solely initialized from random numbers the learning process stalls after few iterations. The values were generated by a so-called "xavier" initializer with a standard deviation of 0.1 [12]. A constant value of 0 initializes the bias values. Other combinations were also tested, such as the more recent "msra" initializer [13], as well as several alterations. But the result was the same (the learning stalled).

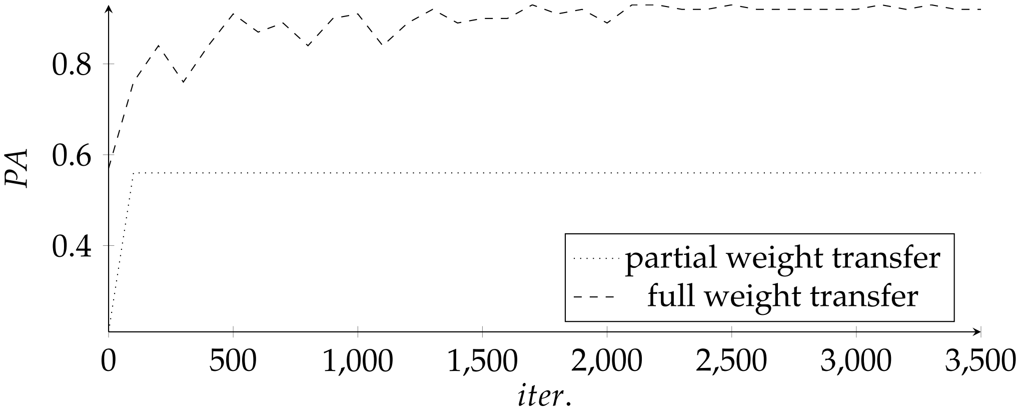


Figure 4.6: The dotted line shows the learning curve for 3 randomly initialized classification layers. The training gets stuck in an inconvenient local minimum. In contrast, the second variant shows a very steep increase (dashed line). This variant is called "full weight transfer", as all weight layers, including the classifiers, are initialized from pre-trained weights. The classifier weights are chosen according to equality or similarity of object classes. Due to better initial conditions the training is facilitated. This is especially important for the critical stage at the beginning.

By transferring weights from equal or similar classes the network is capable to achieve good evaluation scores within few iterations (see figure 4.6). This is due to the better starting conditions at the initial phase. Thus, the network is already beyond the "critical point".

Normalization

I've already discussed some major problems of training deep neural networks. The success of training a deep neural network is very dependent on weight initialization and hyper-parameter selection. Now, I want to discuss further details. What makes the training of a deep neural network so fragile? One reason for this is the so-called "covariate shift" [11]. Covariate shift describes the process when the input distribution of a layer changes. This forces the layer parameters to adapt accordingly. As the input of each layer is affected by all previous layers this becomes problematic for very deep networks. Small changes are amplified throughout the network. Consequently the covariate-shift is more likely to happen. That is why small learning are absolutely necessary. But small learning rates cause a slow training process and may even prevent the escape of suboptimal local minima. In order to avoid this effect Ioffe and Szegedy proposed to normalize all layer inputs such that the input distributions are kept in bounds [11].

Similar problems occur when different layer outputs are fused - e.g. for multimodal fusion.
Liu et al. report that different layer outputs are often very dissimilar in scale [71]. This creates problems when those outputs are merged.

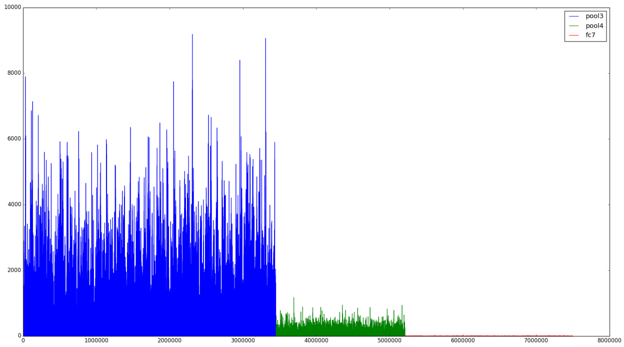


Figure 4.7: For spatial refinement three classifiers are trained, each based on features of different strides. By visualizing the corresponding activations it becomes clear that the scales do drastically differ. Apparently fusing the three classifications inhibits the learning process.

In their work "ParseNet: Looking Wider to See Better" they present a special net architecture where feature vectors are fused by concatenation. They found that different layers frequently produce features of different scales. Naturally features of higher scales are emphasized. Apparently the network is not capable to balance this issue while training. I observed the same problem.
For this reason Liu et al. include a normalization step before fusing features of different scales. They use the well known L_2 normalization. All values are normalized across channels:

$$||x||_2 = \sqrt{\sum_{i=1}^{d} x_i^2} \tag{4.9}$$

where:

x is the input vector
d is the vector length, i.e. the number of channels

However normalizing the feature vector to 1 is cumbersome. Normalizing to 1 leads to very small values which in turn slows down the learning. Therefore it's helpful to multiply all values by a factor s_i. This scaling parameter is initialized by a constant value (e.g. 10 or 20) and then learned for each channel separately. Thus, each value x_i is processed as follows:

$$x_i^* = s_i \cdot \frac{x_i}{||x||_2} \tag{4.10}$$

Inspired by their work I've conducted experiments regarding my own network, especially the skip architecture. Figure 4.7 clearly shows that the activations for "pool3", "pool4" and "fc7" (see figure 4.5) heavily differ in scale. This makes it difficult to learn the skip-architecture from scratch.

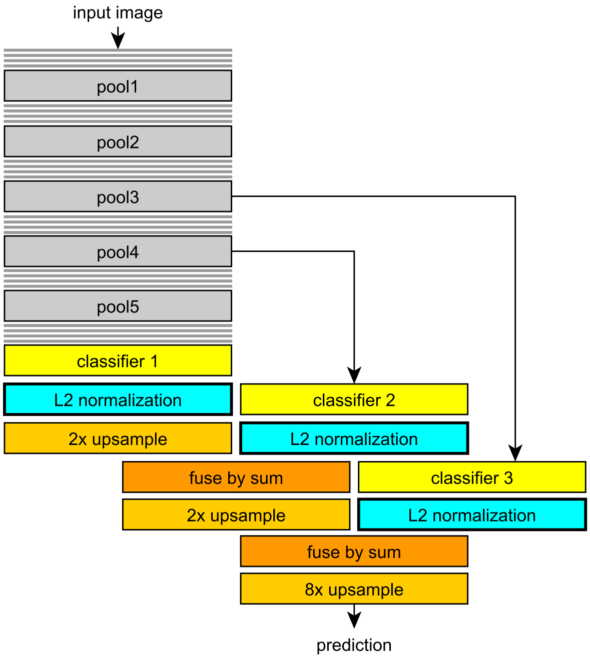


Figure 4.8: Modified skip-architecture: By introducing a normalization step before fusing the scores it's possible to train the skip-architecture at once.

Leveraging from their findings I've applied this normalization technique before fusing classification scores of different strides (see figure 4.8). In contrast to them I normalize the scores and not the feature vectors. I've initialized the scale factors of the first classifier (on top of $fc7$) significantly higher than [71] (100 vs. 10). This emphasizes the influence of the coarsest classifier in the beginning. The influences of the finer predictions can adapt gradually. My experiments have shown that this technique facilitates the learning process and leads to better convergence for training a skip architecture. This effect can be best seen in figure 4.9.

Comparison

To allow a comparison of the proposed methods, figure 4.9 shows all learning curves. In the following I refer "full weight transfer" as method (a) and L$_2$ normalization" as (b).

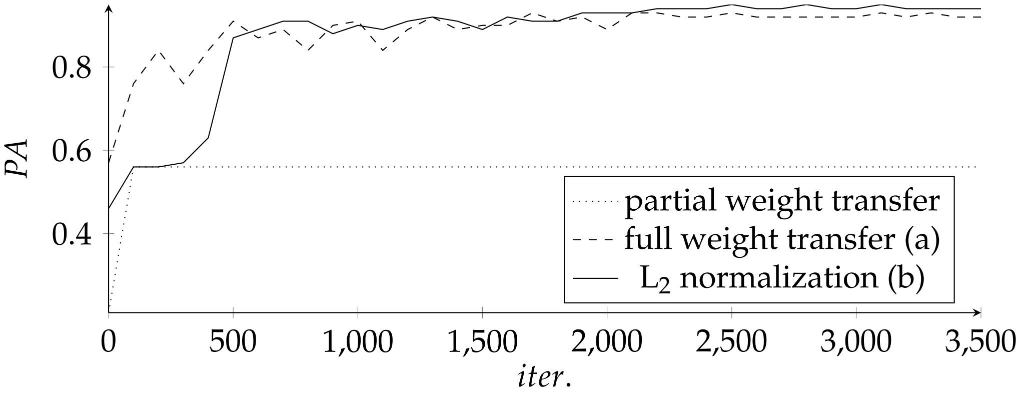


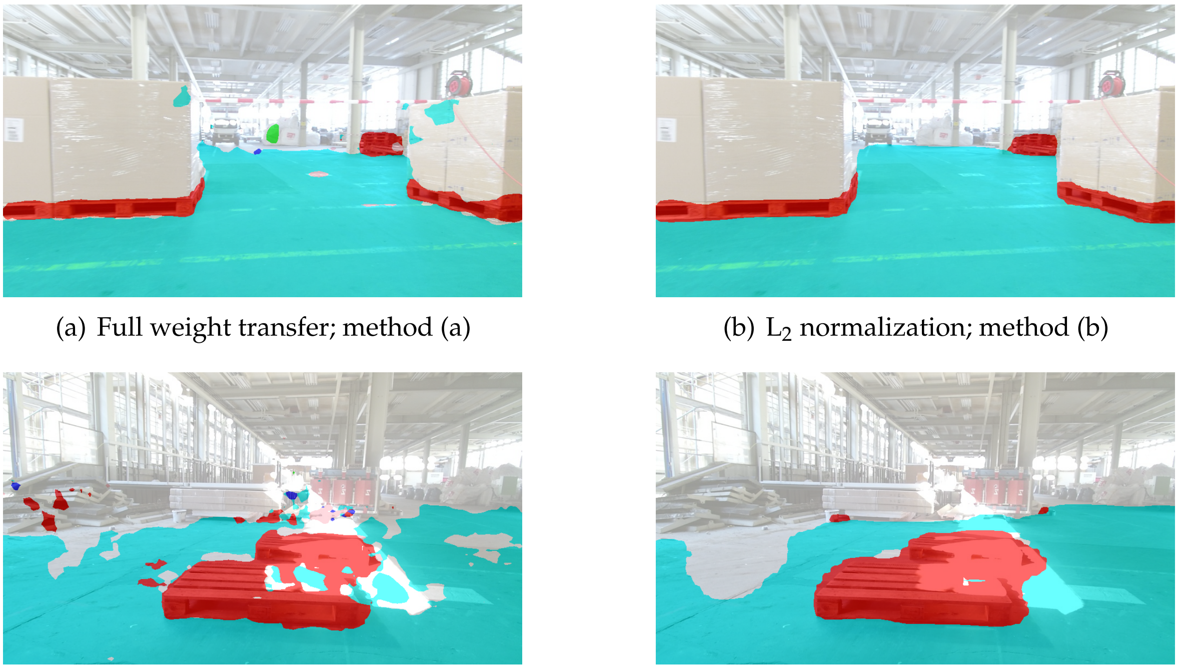
Figure 4.9: Comparison of three variants of training a skip-architecture. The dotted and dashed lines show the learning curves for the previously described variants - "partial" and "full weight transfer". These are shown in order to enable a better comparison. L$_2$ normalization allows to train the entire skip architecture from scratch and even outperforms the second variant on the long run.

Both methods are enabling the training from scratch of the entire skip architecture. L$_2$ (b) leads to steady convergence and outperforms other methods w.r.t. PA in the long run. However, the previous discussion on the PA metric in section 4.2.1 should be considered. This metric shows a general trend, but does not allow a distinct assessment of segmentation accuracy. For this reason the slightly better performance is not a significant advantage. However, the great advantage of using (b) compared to the method (a) is its universal applicability.
In order to compare both models more accurately further error metrics are presented in table 4.4.

Table 4.4: Final results averaged over classes for the proposed methods.

	mPA	mIU	mFP
(a) full weight-transfer	0.798	0.700	0.021
(b) normalization	0.753	0.689	0.018

Here the result is indeed different. Method (a) achieves higher values for both mPA and mIU compared to (b). However, the visual results of method (b) look more consistent (figure 4.10). There are less misclassified "speckles". This is as well confirmed by the false positive rates (mFP).
Due to its universal applicability and the more consistent results I decided to deploy method (b) in the final model (see section 4.5).



(a) Full weight transfer; method (a)

(b) L_2 normalization; method (b)

(c) Full weight transfer; method (a)

(d) L_2 normalization; method (b)

Figure 4.10: Visual comparison of both presented methods (a) and (b). The segmentation result of method (a) shown on the left side is less consistent i.e. includes speckled labelings. Whereas method (b) produces contiguous labeled areas. Apart from the visual consistency, method (a) achieves better results. The small marker visible in the first row, is detected correctly on the left side, whereas on the right side not.

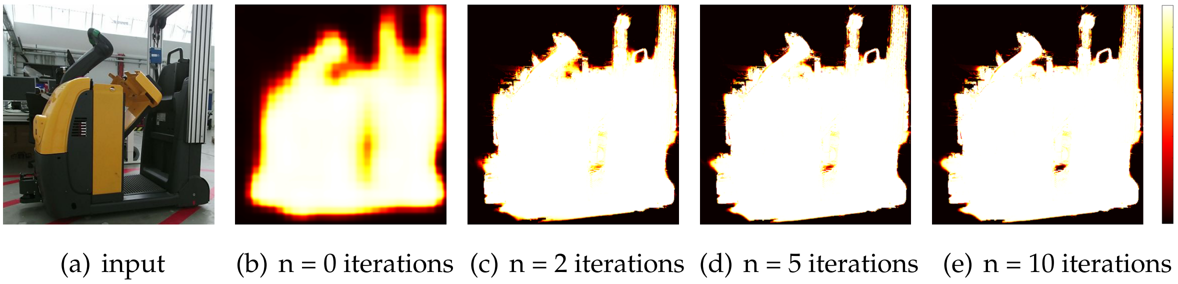
4.4.2 Spatial Accuracy Refinement

Deep neural networks, like "VGG16", are very powerful in making robust classifications, but its "convolutionalized" counterpart shows weaknesses when it comes to spatial accuracy. This section will first recap the reasons and ways to attenuate this weakness. Then I will analyze the impact of two selected techniques for recovering fine details. The results are compared both visually and quantitatively.

The "VGG16" and also other networks are designed to solve high-level vision tasks. High-level tasks, such as image classification, desire a meaningful and general representation of the image as a whole. There is no need to retain spatial information. A desirable property of such a CNN is the invariance w.r.t. geometric transformations. Weight sharing and max-pooling contribute to this property. Usually, pooling goes hand-in-hand with down-sampling ("striding"), which dumps spatial information.
The "VGG16" net for example includes 5 max-pooling layers of stride 2, which leads to a 32 times downsized image representation. Although the image dimensions of in- and output can be aligned by interpolation, the spatial accuracy can not be restored this way (see figure 4.12(a)).
What options are available to solve this problem? One simple implication is to reduce the grade of down-sampling. Long et al. made an attempt to change the output stride of pool5 from 2 to 1, but weren't able to achieve results comparable to their final solution [23]. My experiments have shown similar difficulties. Likewise a shortened network, where all layers between pool4 and pool5 were omitted, didn't lead to competitive results.
This is why I've applied the same **skip-architecture** as proposed by Long et al.. In addition to the classifier based on the 32s-input (32s = stride 32) of fc7, additional classifiers are trained on top of pool4 and pool3. First, the 32s- and 16s outputs are fused, by combining both scores by an elementwise sum (see figure 4.12(b)). Since the 32s output is two times smaller than the 16s output it must be upsampled accordingly. This is done by a deconvolution layer with a kernel stride of 2 and the double kernel size (i.e. 4). The weights of the deconvolution layer are filled by a bilinear filler and are fixed during training. This is repeated in the same manner for the fusion of stride 8 (see figure 4.12(b)). Finally the segmentation map is upsampled by another deconvolution layer by factor 8 (i.e. stride 8). Accordingly this layer has a kernel size of 16. The kernel weights are initialized from a bilinear filler as well, but are adoptable during training.
As stated by Long et al. the fusion of earlier layers (e.g. pool2) does not further improve the segmentation accuracy. Figure 4.13 shows that in fact the improvements tend to saturate.

Another approach includes the use of a graphical model. With the help of the original input image it's possible to refine the coarse segmentation maps by applying a **CRF**. The 8s-output serves for initialization of the unary potentials (see section 2.3). The potentials are refined by maximizing the local consistency of appearance. In this case the color vector was used in order to represent the local appearance. The CRF

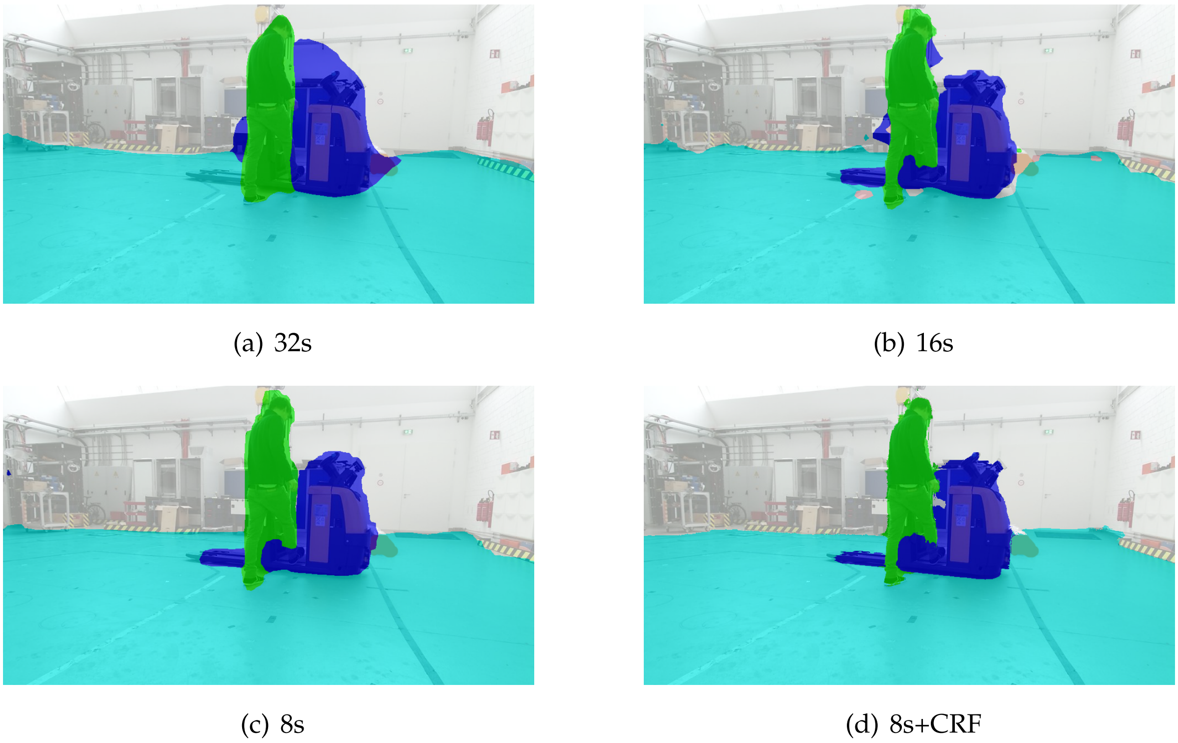
implementation I've used in this work is publicly available [28].
All CRF hyperparameters were set to the same values as used by Chen et al.. This decision is based on a parameter search on the validation test-set w.r.t. the PA metric. I've used "Spearmint with Caffe" (see section 3.2, [54]) for bayesian parameter optimization. The optimization process led to very similar parameters with only slightly better accuracies (0.2%). Since Chen et al. used a significantly larger validation-set I found their findings more trustworthy. For that reason and the marginal improvements I've decided to take over the same hyperparameters.
Both the spatial and bilateral weights are initialized from constant values. Based on the validation errors I use 3 and 4 respectively (instead of 3 and 5 [28]). The diagonal of the compatibility matrix is initialized by -1. The entries of spatial, bilateral and compatibility matrix are then optimized by back-propagation. As recommended I've used only 5 iterations for training, but 10 iterations for inference.
Figure 4.11 shows the iterative improvement of the probability maps.



(a) input (b) n = 0 iterations (c) n = 2 iterations (d) n = 5 iterations (e) n = 10 iterations

Figure 4.11: Visualization of the CRF convergence. The subsequent plots show the evolution of probability maps after n iterations. Black means 0%, whereas white stands for 100% confidence. (a) test image; (b)-(e): probability maps for class "forklift" after n CRF iterations

The visual comparison of the example images (a)-(c) shows how the skip-architecture recovers fine details (see figure 4.11). The forklift is a good example to see the evolution. In (a) the forklift is labeled very coarsely - like someone used a huge brush to paint it. As a result thin parts like the fork are missed and the object boundaries are blurred. This introduces many misclassifications. In (b) the level of detail is improved significantly - best seen at the forks. The fusion of the 8s-scores (image (c)) removes some smaller faulty regions and leads to slightly better results at object boundaries. Image (d) shows the strengths of a CRF. Based on local similarity it adjusts the labeling. A correct but coarse initialization can be refined, which is especially visible at the object boundaries. Nevertheless, the possibilities have not been fully exploited, as you can see at the tip of the fork or the right foot.



(a) 32s

(b) 16s

(c) 8s

(d) 8s+CRF

Figure 4.12: (a) - (c) Comparison of fusing different strides. (d) Additional refinements by a CRF.

By measuring the accuracy at object boundaries this effect can be confirmed quantitatively. The mIU shows a clear upward tendency (see figure 4.13). The mPA value drops at the end (8s+CRF), however this can be explained by the weaknesses of the metric (see section 4.2.1).

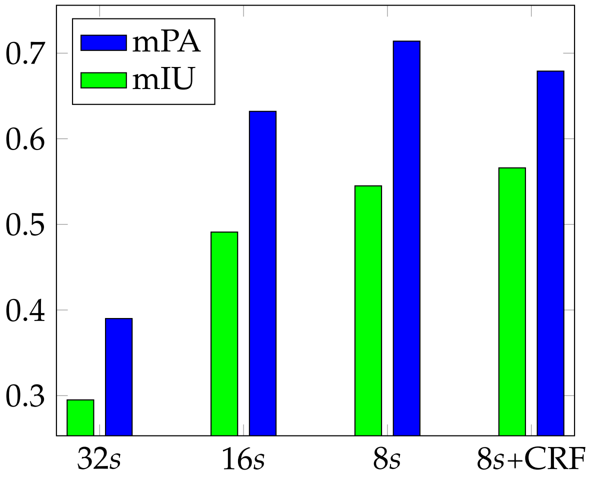


Figure 4.13: Accuracy at object boundaries (trimap of 20px).

4.4.3 Multimodal Fusion

In section 2.4 I've given a general motivation for fusing multimodal inputs, especially concerning color and depth information. While it seems obvious that our system could benefit from depth data, the technical implementation is not that trivial.
First of all the input format has to be conform with the net-structure. More restrictions emerge when the model should be finetuned from an existing RGB model. In that case the three channel input has to be maintained. With the requirement to maintain the original RGB input, this leads to a two-stream architecture - one for color and one for depth. In order to fulfill the net specification one has to spread the depth information over all three channels (compare to section 2.4.1).
At some point both streams need to be fused. According to the literature there seems to be no general rule of thumb [23][35][71]. Both "early fusion" and "late fusion" (described in section 2.4.2) approaches are capable to achieve good results [71]. For that reason I've applied both. In the following I will describe different "early fusion" and "late fusion" approaches and present the evaluation results.

Late Fusion

RGB-HHA net
The first attempt followed the proceeding described by Long et al. [23]. First both networks were trained separately until the evaluation error (mPA) began to converge. Then both networks were fused lately after classification by an elementwise sum.
Late fusion has the advantage that arbitrary networks can be fused since the classification maps are guaranteed to be of equal sizes. This is not always the case for intermediate outputs, such as feature maps. Hence, different network architectures are interchangeable. This work compares two combinations:

- "VGG16-VGG16": A "VGG16" based net was used for both modalities (likewise Long et al. [23])
- "VGG16-Alex": A "VGG16" based net was used for color and "AlexNet" [4] for depth.

"AlexNet" is a more light-weight net architecture with only 60k versus 130k parameters. The reduced time and space complexity is of course a desirable property for both training and deployment stage.
Unfortunately both configurations led to worse results than the single stream network (RGB only). Moreover the "VGG16" outperforms "AlexNet" significantly:

Table 4.5: Late Fusion of RGB and Depth-Streams: VGG16-VGG16 vs. VGG16-Alex

	VGG16-VGG16	VGG16-Alex
mPA	0.774	0.488
mIU	0.677	0.341

One possible explanation is that the depth-stream performs noticeable worse than the RGB-stream. This may be due the little amount of training data including depth. In addition the late fusion approach allows little space for adaptations. The RGB results aren't emphasized accordingly.

Early Fusion

RGB-HHA net

The first attempt for early fusion followed roughly the instructions of Eitel et al. [35]. The feature vectors of fc7 (both VGG16) are concatenated such that a single classification layer can be trained on top. In contrast to [35] I keep fc6 and fc7 in their original configuration. Thus, the number of channels is twice as large (8192).
The classifier on top of the concatenated features is followed by the original skip architecture (color-based) as described by Long et al. [23]. All weights were initialized by pretrained weights (see section 4.4.1).
The performance of this net is even worse than the "late fusion" variant "VGG16-VGG16". It achieves a mean intersection over union of only 0.623 (see table 4.6).
One possible explanation is that the features vectors are of different scale. Without a normalization step this leads to a poor performance as some features dominate over others. One could argue that the weights might adopt accordingly during training, but this was disproved [71]. In addition, this would demand to enable learning in previous layers (i.e. set the local multipliers for the learning rate to values greater

than zero). This is inappropriate because it would promote overfitting. Moreover this would blow up the memory consumption making it impracticable to train in "GPU mode".

Because of these reasons I've decided to add a normalization step before concatenation as described by Liu et al. [71]. Similar to the normalization described in section 4.4.1 a L_2 normalization with additional scaling parameter (initialized by a constant value of 10) was applied. This normalization technique was developed for exactly the same purpose and has shown promising results [71].

Contrary to my expectations this led to even worse results (see table 4.6). One idea to facilitate the learning would be to initialize the scaling parameters in an ascending manner. Hence, higher initial weights for the coarsest and smaller for the finest level. In this way, the training process is encouraged for gradual refinement.

Table 4.6: Comparison of early fusion w and w/o L_2 normalization.

	w/o	w L2-normalization
mPA	0.726	0.536
mIU	0.623	0.435

RGB-H net

The poor results of the previous attempts led me to another more light-weight approach. Under consideration that it might be a good idea to stay as close as possible to the well-performing RGB architecture the idea evolved to use the height as an additional feature for classification. Figure 4.14 illustrates the main principle.

A grayscale image, which represents the height over ground (H), serves as the sole depth input. Instead of extracting high level features the overall form is preserved. For practical reasons the height image is downsized according to the RGB-input by average pooling. In this way the H-feature and the color features can be concatenated at all stages of the skip architecture (not illustrated in figure 4.14).

Table 4.7 shows that the additional H-feature slightly improves the segmentation accuracy. One would expect that especially errors, nearby the ceiling, can be prevented. Figure 4.15 partially supports this presumption. Apparently, the influence of the H-feature is not strong enough to suppress the entire region.

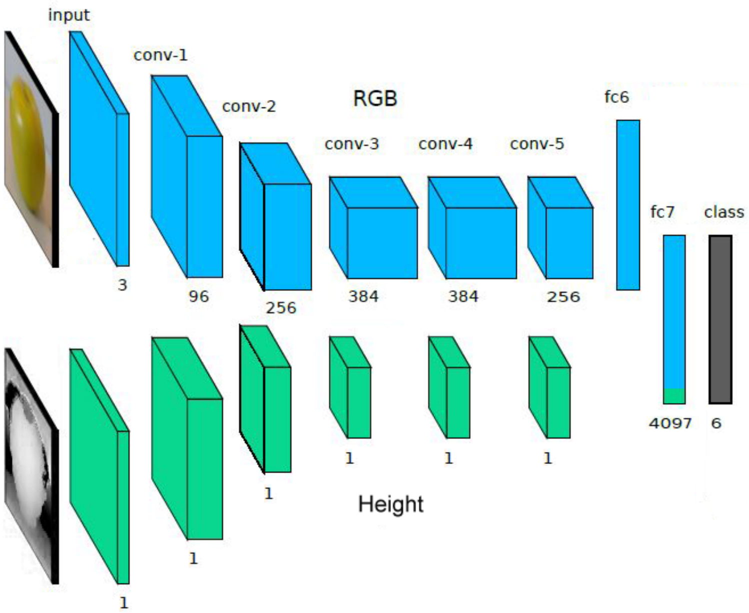


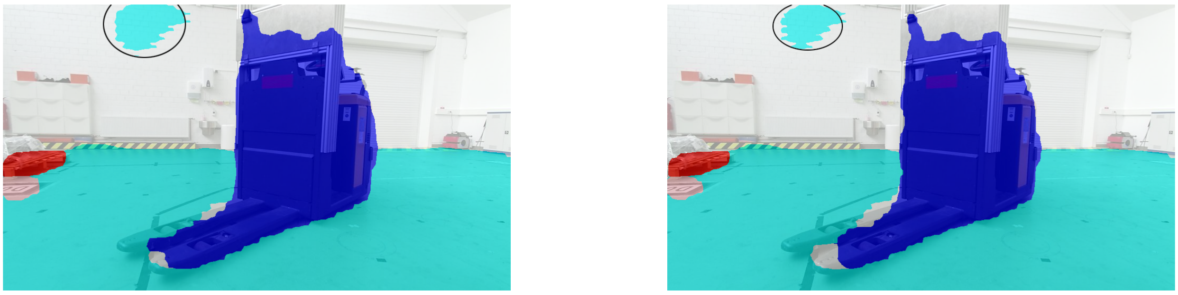
Figure 4.14: Illustration of the RGB-H network. By concatenating 4096 RGB- and one height feature it's possible to learn linear combinations of features obtained from both modes. The pixel-wise classification is based on the concatenated feature vector.

Table 4.7: Class-mean accuracies for the fully convolutional network (RGB-H input).

	RGB net	RGB-H net
mPA	0.753	0.772
mIU	0.689	0.690

One explanation is that the RGB-features dominate over the H-feature (4096 vs. 1). The regularization term (compare to equation 2.4) prevents that a single feature gets a much higher (or smaller) weight than others. In order to increase the influence of the height, one could lower the weight decay for the classification layer, but this would also affect all other weights. This would increase the risk of overfitting.

Another idea, is to duplicate the H-feature. In this way the weight decay remains untouched. One drawback is that this method introduces more parameters. In addition, it's not clear how many times the value has to be duplicated. This should be determined experimentally. In this study, no further experiments were conducted.



(a) RGB net, variant (b)

(b) RGB-H net

Figure 4.15: A visual comparison of the RGB- (left) and RGB-H (right) net outputs shows that the segmentation results are quite similar. On the right the incorrectly labeled region ("floor" instead of "void") nearby the ceiling is slightly smaller (marked by a black oval).

4.5 Final Approach

Based on preliminary experiments I've designed and trained the final model. In order to achieve a high segmentation accuracy I used a network similar to Zheng et. al. Thus, consisting of a FCN and an CRF part.
In contrast to them, I used (as described in 4.4.1) a L_2 normalization to facilitate the training process. Furthermore the classification is based on color- and height-images. The color features are extracted by the "VGG16" based part, but the height image is merely downsized and then directly used. Both modes are fused by concatenation in an early manner. This principle is illustrated in figure 4.14.
The training was divided into three stages. First I've trained the network based on RGB input only. In this way it was possible to leverage from all available RGB-images (see section 4.1).
The next step was to manually transfer the pretrained weights to the RGB-H network. This was necessary, as additional parameters are needed and therefore the mapping between source and target network was ambiguous. The classifier was finetuned on color and height images.
Finally, the net was extended by the CRF and again finetuned. Please note that this increases the accuracy, but also increases the runtime. In my setting, ten CRF iteration approximately increases the runtime by 690ms (960x540 input image). However, by changing the network definition the CRF can be turned off easily. Depending on the requirements, this can be changed anytime.

Table 4.8 shows the final results w.r.t. various metrics. The average runtime for one forward pass is approximately 1170ms (input: 960x540 RGB-H-image). This test was

Table 4.9: Per-class results for the final fully convolutional network (RGB-H input).

			Trimap 20px	
class	mPA	IU	mPA	IU
ground	0.962	0.930	-	-
pallet	0.866	0.688	0.832	0.597
person	0.884	0.803	0.841	0.725
forklift	0.870	0.773	0.763	0.623
marker	0.229	0.098	0.230	0.073
void	0.950	0.902	-	-

conducted on a high-end GPU (see section 3.2). Though, the CRF was executed on the CPU. The implementation for GPU execution is a possible optimization step for future works .

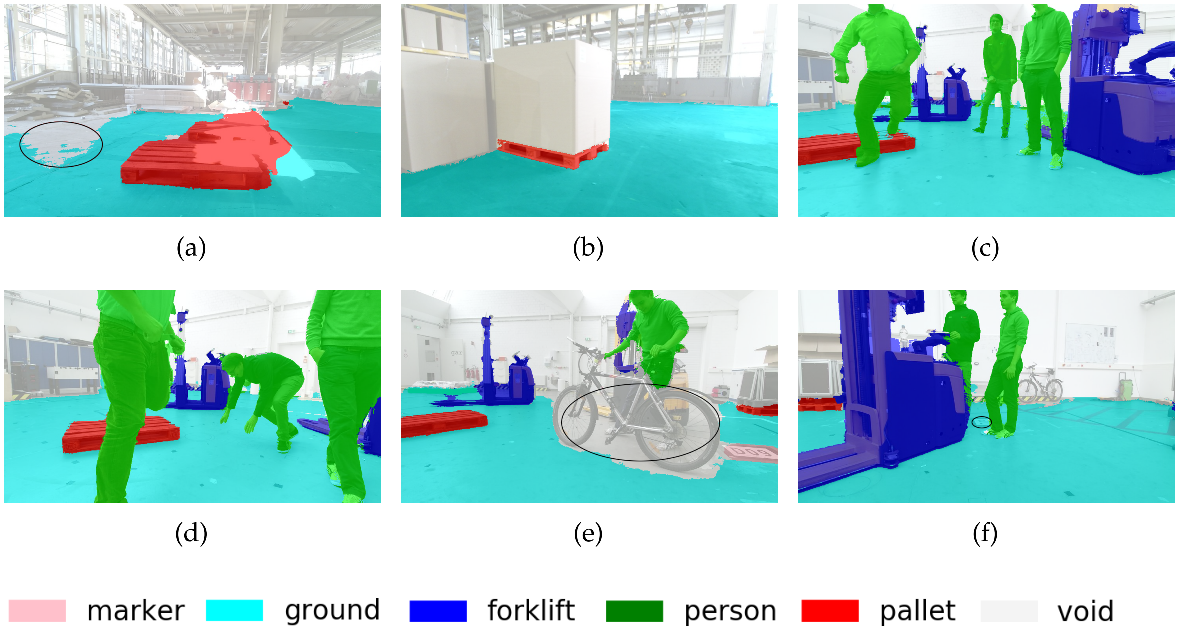
Table 4.8: Class-mean accuracies for the final fully convolutional network with connected CRF (RGB-H input).

		Trimap 20px	
mPA	mIU	mPA	mIU
0.793	0.699	0.667	0.504

My model obtains a (class average) mIU of 0.699. Compared to the results of Long et al. (0.622) and Zheng et al. (0.720) on Pascal VOC 2012 this is a reasonable value. With significantly more training data (incl. MS COCO) Zheng et al. even reach a score of 0.747. As the models are trained on different data sets and object categories, the results are not directly comparable. However, it allows a rough judgment and plausibility check.

Particularly revealing are the class-specific values. These are shown in the appendix A.11. It is striking that the marker class achieved very poor results. By excluding this value the model would obtain a significantly higher mIU value of 0.819. This result is not surprising. Despite class balancing the "marker" class is extremely underrepresented. Furthermore the appearance is often suboptimal - small and distorted. This is due to the wide-angle lens, the low camera position and the relatively small size of the marker. A higher camera position and orientation towards the ground would improve the conditions for the marker perception. However, other objects, such as forklifts or persons, would likely be cropped.

By visualizing the test results (shown in figures A.1 and A.2) it's possible to analyze the strength and weaknesses of the model. One can see that the segmentation results are generally very robust.
Overall there are few misclassified regions. Inaccuracies frequently occur when objects are occluded (best seen in figure A.15). As you can see in figure A.11 some areas of the floor are labeled incorrectly. This phenomenon appeared with the inclusion of the height and especially at image borders. Usually there is a lack of depth information in these regions. As these regions become larger the inpainting method is not able to reconstruct the missing information. However, the advantage of height inclusion is an increased robustness for other classes. There occur less wrong speckles (for example, objects nearby the ceiling). In my opinion, this advantage outweighs the disadvantages for the ground class.



(a) (b) (c)

(d) (e) (f)

Figure 4.16: Exemplary segmentation results of the final net configuration. Problematic regions have been marked with a black-rimmed ellipse. Further segmentation results can be found in the appendix A.2.

Despite several precautions to avoid overfitting (e.g. regularization, data augmentation), it's not clear if the model can perform equally well in other warehouses. The tests I've carried out can not prove such a strong statement. Without a doubt the use case is very challenging and requires a robust model. However, compared to other use cases, e.g. self-driving cars, the conditions are more manageable. Especially indoors, the lighting conditions are controllable. Moreover the appearance of objects belonging to the same class are very similar (e.g. certain type of palette or forklift). Moreover, it is conceivable to deliver customized solutions by finetuning the neural network on additional location-specific images. This is a general advantage of machine learning solutions - they can be automatically adapted to specific conditions.

Migration to a ROS application

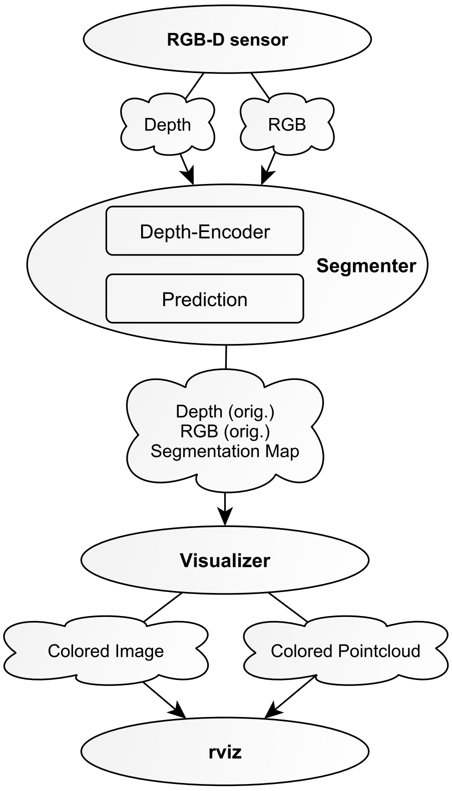


Figure 4.17: A ROS pipeline for semantic segmentation based on RGB-D images. The segmentation is used to prepare a colored image and pointcloud. These serve as an input for rviz. Rviz is a visualization tool for 2D and 3D provided by ROS.

In order to demonstrate the usage of semantic segmentation on a robotic system I've implemented an exemplary ROS application. The communication graph is visualized in figure 4.17. I've embedded the depth encoder and Caffe model in a ROS node (node: "Segmenter"). Based on depth and color input the "Segmenter" node segments

the input into the predefined object categories. Depth is represented by an organized pointcloud, which is provided by a RGB-D camera, such as a Kinect camera. Color is represent by a three channel RGB image.
A custom ROS message type wraps the input and segmentation output. This is advantageous, because for subsequent ROS nodes the synchronization of the messages doesn't have to be performed again. Since the data is passed via smart pointers, no additional costs arise.
The next node colorizes the image and pointcloud for visualization (node: "Visualizer"). Colors are assigned by a predefined look-up table. The colors are added as an overlay to the orginal image. The resulting color components are transfered to the PCL pointcloud. This is possible as the pointcloud is stored in the same order as the image.
Actually, the "Visualizer" node doesn't visualize the output, instead it prepares the segmentation output for visualization. This can be done by rviz, a tool for visualizing 3D and 2D data. The 3D viewer allows to navigate through the 3D model and is also very helpful during the development process. An exemplary screenshot of the visualization result can be seen in figure 4.18. More screenshots and the entire screencast video can be found in the appendix (A.2, A.1).

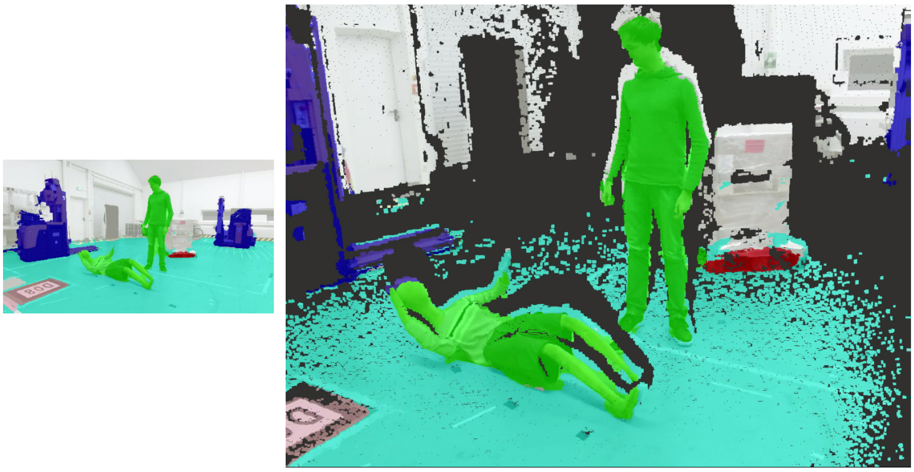


Figure 4.18: A screenshot of rviz showing the segmented image and pointcloud. Even the dummy (not contained in the training data) laying on the ground has been detected correctly. This indicates that the system is invariant with respect to various geometric and perspective transformations.

In the future, the published messages may be consumed not only by one node. Possibly other nodes will join the application graph. These would be responsible, for example, for path planning or for picking up a palette.

5 Future Work and Conclusion

Future Work

In the following I want to share some thoughts and recommendations for future works. Three aspects are mainly important for realizing this feature on a product - the required accuracy and runtime, as well as the size and costs for the deployed hardware. These aspects are of course very case sensitive and interdependent. A comprehensive solution must be found.

Accuracy: Further improvements by simply re-training on more annotated data expectable. This assumption is supported for instance by the results of Zheng et al.. They obtain an improvement of 2.7% w.r.t. mIU by increasing the training set significantly. Of course, the provision of annotated training data is quite costly. It's therefore important to determine the requirements for a certain use case. One should also consider other ways to increase the database. For example:

- Usage of computer generated images:
  - Generation of different views from 3D models.
  - Deployment of generative models for image generation, such as described in [72].
- Usage of weakly annotated images. For example, Bearman et al. work with annotations of one pixel per object [73]. Such annotations can be created much faster. Weak annotations are particularly suitable for pretraining the model.

With more training examples, one should reconsider the presented RGB-HHA networks (see section 4.4.3). These architectures have been successfully applied in other works [23] [35].
There are many other ways to improve the segmentation accuracy. This usually involves more sophisticated techniques and further development.
Currently each image is processed separately. Successive images often contain redundant or similar information. With the inclusion of previous segmentation results, it's possible to improve the runtime, as well as the accuracy. One possibility to enhance

the segmentation quality is the usage of spatio-temporal CRF's, like it is described in the work of Liu et al. [74]. Another graph-based approach to enforce temporally consistent pixels was presented by Couprie et al. [75].

Runtime: This is probably the most critical point. The presented network is far away from a possible real-time usage. First I will present the most obvious and "easy to implement" options.
Although many calculations already benefit from hardware acceleration, there is still a need for further optimization. At present, the CRF can only be executed on the CPU. Parts of the depth-encoding can be easily parallelized as well. Height and disparity are calculated individually for each point. Thus, a parallel execution is straight forward.
The immense number of parameters of the proposed DNN is a key factor for its high segmentation quality. On the other hand it limits the execution speed. One computational bottleneck lies in the first fully convolutional layer. It has 4096 filters of 7x7 spatial size. Chen et al. decrease the spatial size to 4x4 (or 3x3) and reduce the number of channels to 1024 with no significant loss in segmentation accuracy [28]. Probably due to the lack of training images I was not able to achieve equivalent results. For this reason I decided to stay with the pre-trained fully connected layers. However, this approach is still promising for future works.
Other studies also investigated on the reduction of network parameters, in order to reduce the memory footprint, the energy consumption and runtime [76] [77] [78]. While the abundance of parameters seems important to train complex models, only few weights are important for the final performance. This fact can be exploited, in order to thin out the layer connections. In this way, an optimized network with sparse layer connections is obtained. For example Han et al. learn which connections are crucial for the DNN and then prune redundant connections. This is followed by a further finetuning step [76].
Chen et al. follow a different approach. Instead of pruning weights, the memory consumption is reduced by randomly sharing weights. The shared weights are accessed by a low-cost hash function [79].
Another idea would be to fill the temporal gaps between segmentation results with predictions based on appropriate tracking methods. For example, if an object was correctly detected and localized, it can be usually tracked for at least the next subsequent frames. Such tracking systems have been around for some time.

Hardware: If the application affords to cover a close range area, a ToF camera is an

appropriate choice. However, due to their size the Kinect is unsuitable for the usage on a product. The market already holds some suitable industrial cameras for this purpose. Further developments can be expected.
To run the DNN an implementation on a FPGA (Field Programmable Gate Array) might be appropriate. A FPGA has several advantages: low cost, low energy consumption, small physical size and usually a decent speed-up compared to ordinary CPU's. A nice overview is given by Vasilyev and Artem [80].

Conclusion

In the scope of this work a deep neural network for segmentation tasks has been developed. The work was embedded in the context of autonomous forklifts. However, the overall approach and the system is capable for transferal to other use cases. The goal was to provide a detailed understanding of the environment. Despite several difficulties this objective has been achieved. My work described possible pitfalls and how to handle them. Although there was a lack of annotated data the final model achieved competitive results. This was mainly possible through appropriate augmentation techniques and the transferring of learning.
Another interesting aspect was the incorporation of depth. Initially I suspected that depth information will improve the segmentation result significantly. Surprisingly it was hard to surpass the results based on pure color images. The most likely reason is again the lack of annotated data. Although the amount of depth data was insufficient to train a complex model, I was able to design a more light-weight model. This model based on RGB-H input served as a suitable compromise.
Although the depth information is not crucial for the segmentation result, it is still useful for many applications. The combination of depth and semantics is very powerful. Especially autonomous systems can highly benefit from such a knowledge. By knowing the positions of certain objects it's easier to perform specific tasks - such as navigation or palette carriage.

A Appendix

A.1 Segmentation Results

Table A.1: Overview over all tested net configurations. The last colum "Ref." refers to the relevant section.

Net conf.	Input	Base Network	L2 norm.	Fusion	CRF	Ref.
(a)	RGB	VGG16	N	N	N	4.4.1
(b)	RGB	VGG16	Y	N	N	4.4.1
(c)	RGB	VGG16	Y	N	Y	4.4.2
(d)	RGB-HHA	VGG16 (2x)	Y	Late	N	4.4.3
(e)	RGB-HHA	VGG16/Alex	N	Late	N	4.4.3
(f)	RGB-HHA	VGG16 (2x)	N	Early	N	4.4.3
(g)	RGB-HHA	VGG16 (2x)	Y	Early	N	4.4.3
(h)	RGB-H	VGG16	Y	Early	N	4.4.3
(i)	RGB-H	VGG16	Y	Early	Y	4.5

Table A.2: Test results of different net configurations (see legend in table A.1).

				Trimap 20px	
Net conf.	mPA	mIU	fp-rate	mPA	mIU
(a)	0.798	0.700	0.021	0.714	0.545
(b)	0.753	0.689	0.018	0.568	0.491
(c)	0.796	0.703	0.014	0.656	0.512
(d)	0.774	0.677	0.025	0.667	0.529
(e)	0.488	0.341	0.055	0.310	0.114
(f)	0.726	0.614	0.035	0.677	0.486
(g)	0.536	0.435	0.051	0.386	0.353
(h)	0.772	0.690	0.018	0.604	0.490
(i)	0.793	0.699	0.015	0.667	0.504

Table A.3: Per-class results for net configuration (a) (initialization by "full weight transfer").

			Trimap 20px	
class	PA	IU	PA	IU
ground	0.937	0.895	-	-
pallet	0.847	0.639	0.836	0.581
person	0.876	0.787	0.838	0.702
forklift	0.879	0.740	0.791	0.592
marker	0.331	0.276	0.389	0.305
void	0.920	0.864	-	-

Table A.4: Per-class results for net configuration (b) (RGB-input, L2 normalization).

			Trimap 20px	
class	PA	IU	PA	IU
ground	0.959	0.912	-	-
pallet	0.802	0.688	0.731	0.593
person	0.785	0.732	0.697	0.624
forklift	0.808	0.740	0.662	0.570
marker	0.329	0.191	0.328	0.174
void	0.947	0.877	-	-

Table A.5: Per-class results for net configuration (c) (RGB-input, L2 normalization, CRF).

			Trimap 20px	
class	PA	IU	PA	IU
ground	0.959	0.932	-	-
pallet	0.861	0.698	0.829	0.609
person	0.868	0.798	0.819	0.721
forklift	0.855	0.779	0.747	0.639
marker	0.268	0.103	0.230	0.078
void	0.964	0.905	-	-

Table A.6: Per-class results for net configuration (d) (RGB-HHA input, L2 normalization, late fusion).

			Trimap 20px	
class	PA	IU	PA	IU
ground	0.961	0.870	-	-
palett	0.823	0.645	0.781	0.583
person	0.842	0.755	0.769	0.674
forklift	0.876	0.751	0.761	0.610
marker	0.262	0.207	0.358	0.248
void	0.878	0.836	-	-

Table A.7: Per-class results for net configuration (e) (RGB-HHA input, VGG16-AlexNet, late fusion).

			Trimap 20px	
class	PA	IU	PA	IU
ground	0.772	0.737	-	-
pallet	0.249	0.055	0.299	0.052
person	0.540	0.376	0.507	0.356
forklift	0.515	0.092	0.434	0.048
marker	0.000	0.000	0.000	0.000
void	0.853	0.785	-	-

Table A.8: Per-class results for net configuration (f) (RGB-HHA input, early fusion).

			Trimap 20px	
class	PA	IU	PA	IU
ground	0.943	0.808	-	-
pallet	0.553	0.473	0.705	0.555
person	0.896	0.741	0.862	0.643
forklift	0.915	0.684	0.826	0.504
marker	0.222	0.193	0.316	0.241
void	0.828	0.786	-	-

Table A.9: Per-class results for net configuration (g) (RGB-HHA input, early fusion, normalization).

			Trimap 20px	
class	PA	IU	PA	IU
ground	0.935	0.814	-	-
pallet	0.264	0.258	0.294	0.289
person	0.531	0.489	0.540	0.501
forklift	0.491	0.455	0.550	0.490
marker	0.053	0.051	0.159	0.133
void	0.940	0.542	-	-

Table A.10: Per-class results for net configuration (h) (RGB-H input, early fusion, normalization).

			Trimap 20px	
class	PA	IU	PA	IU
ground	0.959	0.912	-	-
pallet	0.802	0.688	0.731	0.593
person	0.785	0.732	0.697	0.624
forklift	0.808	0.740	0.662	0.570
marker	0.329	0.191	0.328	0.174
void	0.947	0.877	-	-

Table A.11: Per-class results for net configuration (i) (RGB-H input, early fusion, normalization, CRF).

			Trimap 20px	
class	PA	IU	PA	IU
ground	0.962	0.930	-	-
pallet	0.866	0.688	0.832	0.597
person	0.884	0.803	0.841	0.725
forklift	0.870	0.773	0.763	0.623
marker	0.229	0.098	0.230	0.073
void	0.950	0.902	-	-

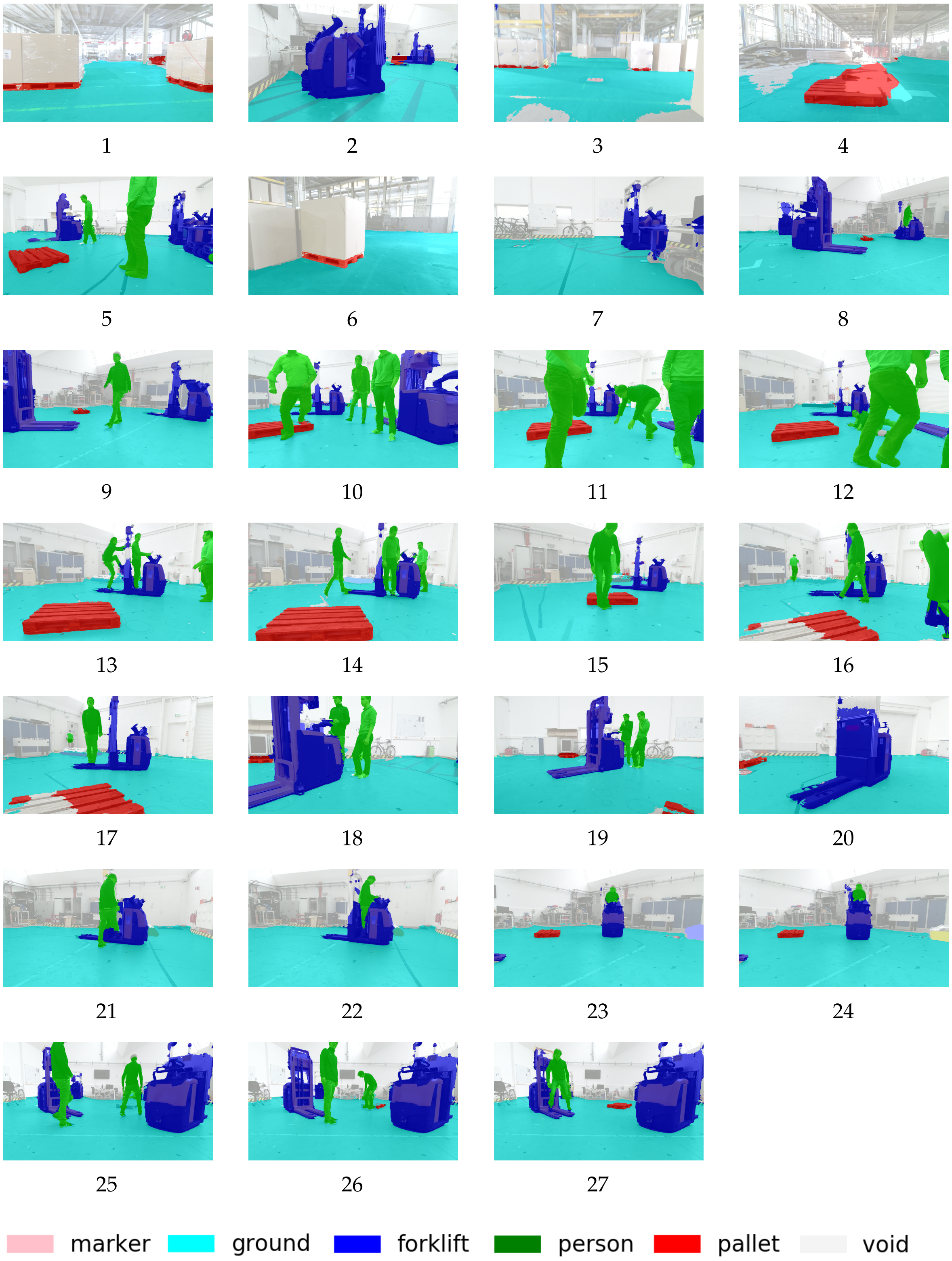


Figure A.1: Segmentation results of the final net configuration (i) (identifiers are explained in table A.1).

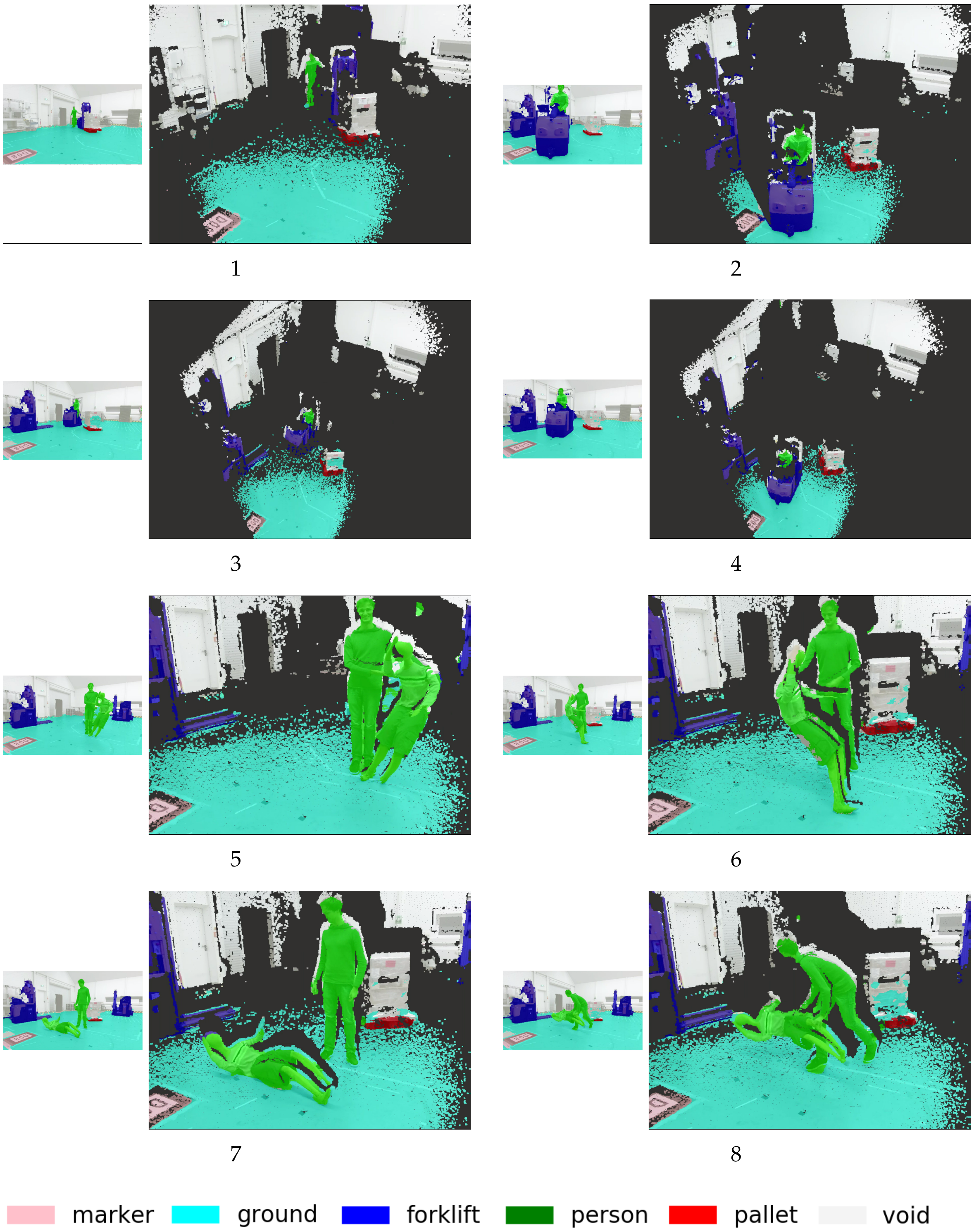


Figure A.2: Segmented images (right) and pointclouds (left) visualized in rviz. These results were obtained from the net with configuration (i) (identifiers are explained in table A.1).

Attached DVD

- Screencast: Visualization of segmentation results in rviz
- Thesis in digital form

Bibliography

[1] D. Lin, C. Kong, S. Fidler, and R. Urtasun, "Generating multi-sentence lingual descriptions of indoor scenes," Mar. 2015.

[2] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and tell: A neural image caption generator," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3156–3164, 2015.

[3] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, pp. 91–110, 2004.

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

[5] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," Sept. 2014.

[6] A. Barron, "Universal approximation bounds for superpositions of a sigmoidal function," *Information Theory, IEEE Transactions on*, vol. 39, pp. 930–945, May 1993.

[7] G. Montavon, G. B. Orr, and K.-R. Müller, eds., *Neural Networks: Tricks of the Trade, Reloaded*, vol. 7700 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2nd edn ed., 2012.

[8] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*, pp. 177–186, Springer, 2010.

[9] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," July 2012.

[10] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[11] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," Feb. 2015.

[12] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS™10). Society for Artificial Intelligence and Statistics*, 2010.

[13] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," Feb. 2015.

[14] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," Nov. 2013.

[15] A. Mahendran and A. Vedaldi, "Understanding deep image representations by inverting them," Dec. 2014.

[16] M. D. Zeiler and R. Fergus, "Stochastic pooling for regularization of deep convolutional neural networks," Jan. 2013.

[17] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?," *Advances in Neural Information Processing Systems 3320-3328. Dec. 2014*, vol. Advances 3320-3328. Dec. 2014, pp. Advances in Neural Information Processing Systems 27,pages 3320–3328 Dec.2014, Nov. 2014.

[18] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," Nov. 2013.

[19] S. Gupta, R. Girshick, P. Arbeláez, and J. Malik, "Learning rich features from rgb-d images for object detection and segmentation," July 2014.

[20] B. Hariharan, P. Arbeláez, R. Girshick, and J. Malik, "Simultaneous detection and segmentation," July 2014.

[21] J. Long, N. Zhang, and T. Darrell, "Do convnets learn correspondence?," Nov. 2014.

[22] C. Farabet, C. Couprie, L. Najman, and Y. LeCun, "Learning hierarchical features for scene labeling," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 35, no. 8, pp. 1915–1929, 2013.

[23] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," *CoRR*, vol. abs/1411.4038, 2014.

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

[25] C. Sutton and A. McCallum, "An introduction to conditional random fields," *Foundations and Trends in Machine Learning*, vol. 4, no. 4, pp. 267–373, 2011.

[26] P. Krähenbühl and V. Koltun, "Efficient inference in fully connected crfs with gaussian edge potentials," in *Advances in Neural Information Processing Systems 24* (J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Weinberger, eds.), pp. 109–117, Curran Associates, Inc., 2011.

[27] A. Adams, J. Baek, and M. A. Davis, "Fast high-dimensional filtering using the permutohedral lattice," vol. 29, no. 2, pp. 753–762, 2010.

[28] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Semantic image segmentation with deep convolutional nets and fully connected crfs," Dec. 2014.

[29] S. Zheng, S. Jayasumana, B. Romera-Paredes, V. Vineet, Z. Su, D. Du, C. Huang, and P. H. S. Torr, "Conditional random fields as recurrent neural networks," Feb. 2015.

[30] R. Rojas, *Neural Networks: A Systematic Introduction*. New York, NY, USA: Springer-Verlag New York, Inc., 1996.

[31] A. G. Schwing and R. Urtasun, "Fully connected deep structured networks," Mar. 2015.

[32] Z. Liu, X. Li, P. Luo, C. C. Loy, and X. Tang, "Semantic image segmentation via deep parsing network," Sept. 2015.

[33] G. Lin, C. Shen, I. Reid, and A. van den Hengel, "Deeply learning the messages in message passing inference," June 2015.

[34] G. Lin, C. Shen, I. Reid, and A. van dan Hengel, "Efficient piecewise training of deep structured models for semantic segmentation," Apr. 2015.

[35] A. Eitel, J. T. Springenberg, L. Spinello, M. Riedmiller, and W. Burgard, "Multimodal deep learning for robust rgb-d object recognition," July 2015.

[36] M. Bertalmio, A. L. Bertozzi, and G. Sapiro, "Navier-stokes, fluid dynamics, and image and video inpainting," in *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, vol. 1, pp. I–355, IEEE, 2001.

[37] B. Ionescu, J. Benois-Pineau, T. Piatrik, and G. Qunot, *Fusion in Computer Vision: Understanding Complex Visual Content*. Springer Publishing Company, Incorporated, 2014.

[38] P. K. Nathan Silberman, Derek Hoiem and R. Fergus, "Indoor segmentation and support inference from rgbd images," in *ECCV*, 2012.

[39] C. G. M. Snoek, M. Worring, and A. W. M. Smeulders, "Early versus late fusion in semantic video analysis," *Proceedings of the 13th annual ACM international conference on Multimedia - MULTIMEDIA™05*, 2005.

[40] I. Lenz, H. Lee, and A. Saxena, "Deep learning for detecting robotic grasps," 2013.

[41] Microsoft Corp., Redmond WA., *Kinect for Xbox 3*.

[42] S. Gokturk, H. Yalcin, and C. Bamji, "A time-of-flight depth sensor - system description, issues and solutions," in *Computer Vision and Pattern Recognition Workshop, 2004. CVPRW '04. Conference on*, pp. 35–35, June 2004.

[43] J. Sell and P. O'Connor, "The xbox one system on a chip and kinect sensor," *IEEE Micro*, vol. 34, pp. 44–53, Mar 2014.

[44] E. Lachat, H. Macher, M.-A. Mittet, T. Landes, and P. Grussenmeyer, "First Experiences with Kinect v2 Sensor for Close Range 3d Modelling," *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, pp. 93–100, Feb. 2015.

[45] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. J. Goodfellow, A. Bergeron, N. Bouchard, and Y. Bengio, "Theano: new features and speed improvements." Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[46] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, no. EPFL-CONF-192376, 2011.

[47] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems (Preliminary White Paper, November 9, 2015)," Nov. 2015.

[48] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," Aug. 2014.

[49] NVIDIA, *caffe - NVIDIA*. `https://github.com/NVIDIA/caffe`, Accessed: 2016-01-06.

[50] NVIDIA, *DIGITS*. `https://github.com/NVIDIA/DIGITS`, Accessed: 2016-01-06.

[51] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," Oct. 2014.

[52] Berkeley, *Model Zoo*. `https://caffe.berkeleyvision.org/model_zoo.html`, Accessed: 2015-12-05.

[53] NVIDIA, *Tesla K-80 Board Specification v05*, January 2015. `http://images.nvidia.com/content/pdf/kepler/Tesla-K80-BoardSpec-07317-001-v05.pdf`, Accessed: 2016-01-06.

[54] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," June 2012.

[55] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[56] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*, (Shanghai, China), May 9-13 2011.

[57] *PCL: Point Cloud Library*. `https://github.com/PointCloudLibrary/pcl`, Accessed: 2016-01-17.

[58] G. Bradski *Dr. Dobb's Journal of Software Tools*, 2000.

[59] *OpenCV: Open Source Computer Vision Library*. `https://github.com/Itseez/opencv`, Accessed: 2016-01-16.

[60] K. Yamaguchi, *js-segment-annotator*. Sendai, 2015. `https://github.com/kyamagu/js-segment-annotator`, Accessed: 2015-12-16.

[61] R. Achanta, A. Shaji, K. Smith, A. Lucchi, P. Fua, and S. Süsstrunk, "SLIC Superpixels Compared to State-of-the-art Superpixel Methods," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 11, pp. 2274 – 2282, 2012. A previous version of this article was published as a EPFL Technical Report in 2010: http://infoscience.epfl.ch/record/149300. Supplementary material can be found at: http://ivrg.epfl.ch/research/superpixels.

[62] P. Fankhauser, M. Bloesch, D. Rodriguez, , R. Kaestner, M. Hutter, and R. Siegwart, "Kinect v2 for mobile robot navigation: Evaluation and modeling," in *IEEE International Conference on Advanced Robotics (ICAR)*, 2015.

[63] R. K. Thiemo Wiedemeyer, *kinect2-ros*, 2015. `https://github.com/ethz-asl/kinect2-ros`, Accessed: 2015-12-27.

[64] W. W. Z. Everingham, Van Gool, "The pascal visual object classes challenge 2012 (voc2012) results." http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[65] J. Xiao, J. Hays, K. Ehinger, A. Oliva, A. Torralba, *et al.*, "Sun database: Large-scale scene recognition from abbey to zoo," in *Computer vision and pattern recognition (CVPR), 2010 IEEE conference on*, pp. 3485–3492, IEEE, 2010.

[66] S. Holzer, R. B. Rusu, M. Dixon, S. Gedikli, and N. Navab, "Adaptive neighborhood selection for real-time surface normal estimation from organized point cloud data using integral images," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pp. 2684–2689, IEEE, 2012.

[67] M. A. Fischler and R. C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Commun. ACM*, vol. 24, pp. 381–395, June 1981.

[68] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, "Microsoft coco: Common objects in context," May 2014.

[69] P. Kohli, L. Ladický, and P. H. Torr, "Robust higher order potentials for enforcing label consistency," *Int. J. Comput. Vision*, vol. 82, pp. 302–324, May 2009.

[70] *Results of ILSVRC-2014.* `http://image-net.org/challenges/LSVRC/2014/results`, Accessed: 2016-01-19.

[71] W. Liu, A. Rabinovich, and A. C. Berg, "Parsenet: Looking wider to see better," June 2015.

[72] A. Dosovitskiy, J. T. Springenberg, M. Tatarchenko, and T. Brox, "Learning to generate chairs, tables and cars with convolutional networks," Nov. 2014.

[73] A. Bearman, O. Russakovsky, V. Ferrari, and L. Fei-Fei, "What's the point: Semantic segmentation with point supervision," June 2015.

[74] B. Liu and X. He, "Multiclass semantic video segmentation with object-level active inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4286–4294, 2015.

[75] C. Couprie, C. Farabet, and Y. LeCun, "Causal graph-based video segmentation," Jan. 2013.

[76] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," June 2015.

[77] S. Srinivas and R. V. Babu, "Data-free parameter pruning for deep neural networks," July 2015.

[78] M. D. Collins and P. Kohli, "Memory bounded deep convolutional networks," Dec. 2014.

[79] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," Apr. 2015.

[80] A. Vasilyev, "Cnn optimizations for embedded systems and fft,"

Declaration of Originality

I hereby declare that this thesis represents my original work and that I have used no other sources except as noted by citations.

Hiermit versichere ich, diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht zu haben.

Stuttgart, March 12, 2016

Nicolai Harich