CHALMERS

Image segmentation and convolutional neural networks as tools for indoor scene understanding

Final report

Author:
Adam Liberda
19900822-4579
liberda@student.chalmers.se

Author:
Adam Lilja
19941125-3538
adamlil@student.chalmers.se

Author:
Björn Langborn
19940913-0870
langborn@student.chalmers.se

Author:
Jakob Lindström
19940701-1296
jaklinds@student.chalmers.se

Examiner:
Fredrik Kahl

Supervisor:
Måns Larsson

Bachelor's Thesis at Signals and systems - SSYX02-16-31
May 17, 2016

Abstract

Computer vision using deep learning is a rapidly increasing computer science field based on a set of algorithms giving machines the ability to learn from examples. In general, computer vision is considered a central aid in automation of processes where humans historically have been in control but machines could do it more efficiently. The purpose of this project is to create a program being able to pixel-wise classify images containing common indoor objects, a method known as semantic segmentation.

In order to support the result a thorough theoretical background to convolutional neural networks (CNNs) in general is provided. The main task is to create a program taking an image of an indoor environment as input, and giving what objects are detected and where in the picture they are as outputs. To do this a vast and diverse data set needs to be set, an efficient and accurate network structure built and trained, and an algorithm using the output from the CNN in order to spatially locate objects created. The final convolutional neural network and semantic segmentation developed by this project understands an indoor scene with an accuracy of mean Jaccard index of 0.3939. The spatial localisation is fully functional as long as the CNN's segmentation is sufficiently well done.

The network was trained for roughly a week which is considered enough for a CNN of this kind, but yet failed to provide desired results. Nevertheless, this study concludes that a convolutional neural network can be learnt via deep learning methods to perform semantic segmentation. The key factors for not achieving wished for results are concluded as being the architecture of the network and its parameters, as well as the data set used to train the CNN.

Sammandrag

Datorseende och mer specifikt med hjälp av djupinlärning är ett snabbt växande fält inom datorvetenskap som baserat på en mängd algoritmer ger maskiner förmågan att lära sig utifrån exempel. Generellt sett är datorseende ett centralt hjälpmedel för att automatisera processer som historiskt sett varit kontrollerade av människor men där maskiner har möjlighet att utföra dessa mer effektivt. Syftet med detta projekt är att skapa en programvara som pixelvis kan klassificera bilder på inomhusmiljöer.

För att bygga en grund för resultatet tillhandahålls en genomgående teoretisk bakgrund av neurala faltningsnätverk (CNN). Huvuduppgiften består i att producera ett program som tar en bild på en inomhusmiljö som indata, och som genererar utdata i form av en specifikation på vilka objekt som finns i bilden samt var i bilden de befinner sig. För att åstadkomma detta krävs en stor och varierad datamängd, utveckling samt träning av en effektiv och precis nätverksstruktur och skapandet av en algoritm som tar utdata från nätverket och spatiellt lokaliserar objekten i denna. Det slutliga neurala faltningsnätverket utvecklat i denna rapport klassificerar objekt med en precision på 0.3939 mätt i Jaccard index. Den spatiella lokaliseringsalgoritmen är fullt fungerande så länge indatan är tillräckligt välsegmenterad.

Nätverket tränades i ungefär en vecka, vilket anses vara tillräckligt för ett CNN av detta slag, men trots det uppnåddes inte önskat resultat. Likväl dras slutsatsen att ett neuralt faltningsnätverk kan tränas med hjälp av djupinlärning för att utföra semantisk segmentering. De viktigaste orsakerna bakom varför programvaran producerade otillräckliga resultat anses vara nätverksarkitekturen och dess parametrar, samt datamängden som användes för att träna nätverket.

Contents

1 Introduction **1**
1.1 Purpose and Scope . . . 2
1.2 Problems . . . 2
1.3 Limitations . . . 2
1.4 Outline . . . 3

2 Theoretical background **3**
2.1 Convolutional Neural Networks . . . 3
2.2 Convolutional Neural Networks for image applications . . . 3
2.2.1 Whole-image classification . . . 4
2.2.2 Semantic segmentation . . . 4
2.3 Data and preprocessing . . . 4
2.3.1 Splitting data . . . 5
2.3.2 Preprocessing . . . 5
2.4 Basic operations of a Convolutional Neural Network . . . 6
2.4.1 Forward propagation . . . 7
2.4.2 Loss function . . . 7
2.4.3 Backward propagation . . . 7
2.5 The layers of a Convolutional Neural Network . . . 8
2.5.1 Convolutional layer . . . 8
2.5.2 Rectified Linear Unit . . . 10
2.5.3 Pooling layer . . . 10
2.5.4 Fully Connected layer . . . 10
2.5.5 Softmax Loss . . . 11
2.5.6 Batch Normalization layer . . . 11
2.5.7 Deconvolutional layer . . . 11
2.6 Training a Convolutional Neural Network . . . 12
2.7 Challenges and common solutions when training . . . 13
2.7.1 Overfitting . . . 13
2.7.2 Data augmentation . . . 13
2.7.3 Dropout . . . 13
2.7.4 Batch normalization . . . 14
2.8 Evaluating a Convolutional Neural Network . . . 14
2.8.1 Evaluating semantic segmentation . . . 14

3 Method **15**
3.1 Choice and usage of the data set . . . 15
3.1.1 Reformatting the data set images . . . 16
3.1.2 Reformatting the data set labels . . . 16
3.2 Matlab implementation details . . . 17
3.2.1 Choice of MatConvNet-wrapper . . . 17
3.2.2 GPU drivers for faster computations . . . 17
3.3 Experiments - network architectures . . . 18
3.4 Spatial localisation of objects . . . 19
3.5 Evaluating the obtained network . . . 19
3.5.1 Implementation of Jaccard indices . . . 19
3.5.2 Harsh or non-harsh evaluation . . . 20

3.5.3 Uncertain predictions and thresholding . . . 21

4 Results **21**

4.1 Final network structure . . . 21

4.1.1 Network training statistics . . . 22

4.1.2 Network performance in Jaccard indices . . . 22

4.1.3 Example segmentations . . . 24

4.2 Spatial localisation of objects . . . 26

5 Discussion **28**

5.1 Experiments . . . 28

5.2 Final network structure . . . 29

5.3 Obtained results . . . 30

5.3.1 Ground truth . . . 30

5.3.2 Jaccard indices . . . 32

5.3.3 Obtained segmentations . . . 33

5.3.4 Downscaling labels, and the procedure's consequences for the network . . 34

5.4 Spatial location of objects . . . 35

5.5 General problems . . . 35

6 Conclusion **36**

References **38**

Appendix A Final structure **39**

Appendix B Experiments **42**

Appendix C Chosing Classes Algorithm **44**

Appendix D Reformat Label Values Algorithm **47**

Appendix E Spatial localisation algorithm **49**

1 Introduction

This report describes the process of learning a convolutional neural network (CNN) to perform semantic segmentation for common objects in indoor environments via deep learning. What a CNN for this purpose basically does is to attempt pixel-wise predictions of the objects contained in an image. Example of objects are chairs, floor, desks and monitors. These are also referred to as the network's classes. Object's not learnt by the network ought to be classified as background. As the report being heavy in theory, the terminology and concepts will be explained along the way.

To put this study in context, the study falls under the research category of computer vision. Computer vision is a subarea in computer science focusing on computers' understanding of the world through image analysis and can be divided into fields such as object classification or identification, optical character recognition, pose estimation, face detection or recognition, and semantic segmentation. One highly relevant method of achieving these tasks can be deep learning. For images specifically, one performs deep learning by providing a program with a lot of data and visual contents that contain examples and counter-examples of a task or concept that the program is to learn.

Depending on the program's purpose, a great variety of outputs can be acquired from the computer. Common outputs that can be obtained after a program has analyzed an image is numerical or symbolic labels, or a probability distribution over the identified objects, characters or faces. Whilst the network's performance in the case of images is commonly evaluated via test images and algorithms, humans act as the ultimate judge deciding whether an input is classified correctly or not. Therefore these kinds of studies in computer vision naturally end up in algorithms simulating the human brains' comprehension of visual impressions.

The objective can vary from automating a straightforward but time consuming processes, to exceeding the human brain in more complex tasks. In some application the latter performance is already reached, as He et al. [1] did in the competition ImageNet Classification Challenge. Their neural network managed to classify objects ranging from animals to sports images with higher precision (4.94% error classification) than the average human (5.1%)[2]. In other cases, as in MS COCO Captioning Challenge, where pictures are to be described in words humans are still superior to the machines in image understanding [3]. The technique is relatively new and still expanding in a prodigious manner. One up-to-date application is the artificial intelligence in autonomous cars, for example the vision based lane tracking in Fletcher et al. [4]. Here computer vision is used to detect and track the boundaries and lane markings of a road.

In general, computer vision is considered a central aid in automation of processes where humans historically have been in control but machines will do it more efficiently. These processes can also for instance be of the kind that interaction between human and machine depend on the machine understanding its surrounding. Applications range from complex medical tasks such as robots with diagnostic ability to simpler tasks like helping humans lift heavy items. Countless computer vision applications are very relevant since there are tasks humans might not be able, willing, need nor ought to perform. To point out a difficulty of achieving well-performing computer vision programs via deep learning, one usually needs a lot of data in order for the program to learn concepts that can be generalized. In order to learn a program for computer vision tasks, one generally does not only need a lot of images but also images with sufficient diversity.

Variation in images can be in sense of the object's angle, size or deformation. Also it scopes if the object is obscured or if the illumination impinge the visual impression. This heterogeneity naturally results in the required need of large data sets of images. As a reference the frequently occurring data set CIFAR-10 consists of 60,000 images for training and evaluation of 10 different objects.

One field in computer vision is, as earlier mentioned, semantic segmentation. This is the process of splitting an image into different segments which describe objects' shapes and contents. More specifically semantic segmentation assigns each pixel in an image with a label, where each label may represent an object or just background. Taking the example with autonomous cars once again, semantic segmentation can be used to classify whether there is a road, sign or person in the circumference and in turn make the car act accordingly. Henceforth this paper focuses on indoor semantic segmentation and what can be achieved using its output information.

1.1 Purpose and Scope

Through usage of a convolutional neural network for semantic segmentation the purpose of this project is to determine not only what objects there are in an image and where they are located pixel-wise, but also to comment on the location of each and every object. About twenty five, by the authors set, different common indoor objects are going to be pixel-wise classified.Lamps, chairs and garbage bins are examples of objects in the spectra. The program's ability to perform semantic segmentation will be evaluated and compared to ground truth, which is defined as the correct segmentation of an image. The similarity between the segmentation and ground truth will be measured using Jaccard indices defined in (16). The goal is achieving an average value of $J \geq 0.6$ for image-wise segmentation.

1.2 Problems

To fulfill these goals there will appear problems and trade-offs. These can be split into two main problems regarding the network structure and the spatial localisation respectively. The network structure will stand for the greater part of the concerns by needing much mental power to construct the architecture leading to great efficiency and accuracy. Questions like 'which network structure is most efficient?', 'which and how many layers are needed?', 'is it worth having image preprocessing?' and 'what hyperparameters are optimal?' obviously need answers. The main challenge regarding the spatial localisation will be developing a fast and efficient algorithm. In order to solve the two main problems other smaller obstacles presumably appear along the way. About the network structure those include the data set and questions such as 'how much data is needed?' and 'are the images of satisfactory diverse?' arise. They also include software issues like 'what programming language is preferable?', 'which toolbox is to be used?' and 'what wrapper will lead to the least amount of trouble?'.

1.3 Limitations

The main task is to create a program taking a picture of an indoor environment as input, and giving what objects are detected and where in the picture they are as outputs. The number of possible objects are restricted to approximately 25 common indoor items. The network's structure is created by the authors of this report and will be representative for the short period of time, 2 study periods, disposed for the project. The resources regarding computational powers are also limited.

1.4 Outline

Some theoretical background is needed and the following chapter treats just that. Beginning with the basics as what a convolutional neural network is and its possible applications. Specifications about data set needed as well as what one ought to do with it before usage is here discussed. Continuing by depicting important types of layers and ending up in how a CNN is trained, and what challenges and common solutions there are. Later, when more knowledge has been gathered, the method for development of this projects CNN is described. The method section is divided into 5 parts concerning the data set, Matlab implementations, experiments, spatial localisation of objects and evaluation of the obtained network respectively. In the result section the obtained program and its structure will be accounted for together with achieved performance and accuracy. Answers to the problem statements in 1.2 not treated in chapter 3 will successively be so in the result chapter as well.

2 Theoretical background

The following section intends to provide some theoretical background concerning convolutional neural networks and how they work. It starts with explaining what a convolutional neural network is and describes its application in object classification and semantic segmentation. Furthermore, the basic foundational operations, as well as different layers used in a network of this kind, are clarified. It also covers theory regarding preprocessing of the input data, how to train a network and various challenges and how to deal with these. As a final part this chapter describes how to evaluate a CNN.

2.1 Convolutional Neural Networks

A convolutional neural network (CNN) is a tool used in data processing for various tasks, amongst other image processing. In image processing, tasks can vary from object classification and object detection to face recognition and semantic segmentation. When used for the latter, the CNN takes an image as input data and produces a segmentation of the image, where each class is represented by a different numerical value and color. The classes are defined by the user and refer to the different objects in the image, for example floors, chairs, cats and desks. A CNN can, mathematically, be described as the following function

$$f(\mathbf{x}) = f_N(f_{N-1}(...f_2(f_1(\mathbf{x}; \mathbf{w}_1); \mathbf{w}_2)...; \mathbf{w}_{N-1}); \mathbf{w}_N). \tag{1}$$

The functions $f_i, i = 1, \ldots, N$ are more commonly referred to as computational blocks or layers in this context. A function $f_i, i = 1, \ldots, N$ takes an input $\mathbf{x}_i$ that, depending on layer type, operates differently on the data resulting in an output $\mathbf{x_{i+1}}$. Every layer has some layer specific parameters that affect the data output, but some layers also possess learnable weights. For convenience and mathematical brevity $\mathbf{w}_i$ includes either just layer parameters or both in equation (1). Which layers have learnable weights is discussed in section 2.5. The network also contain other eligible parameters, which along with the layer specific parameters are referred to as hyperparameters. An example of a hyperparameter can be the choice of learning rate. Together all these settings govern the overall training process of the network.

2.2 Convolutional Neural Networks for image applications

Whilst CNNs are applicable in many research fields, it is probably most often seen in the context of image applications, using visual information to learn a specific task. Two prominent examples are to be discussed here, the task of giving an entire image a single label and semantic segmentation. A visual example of the differences can be seen in Figure 1.

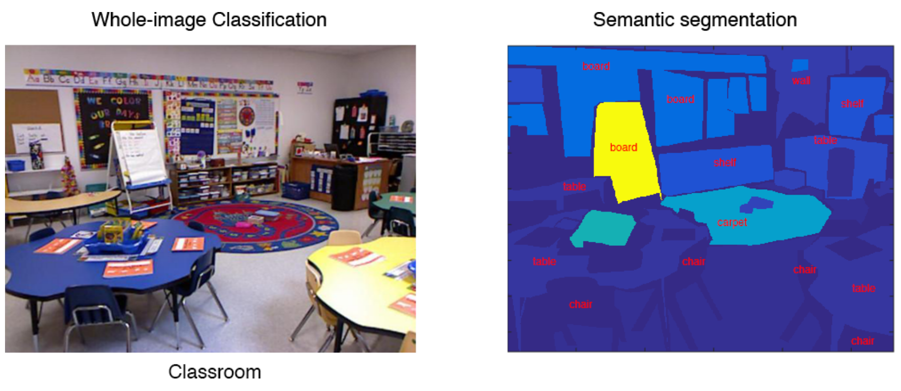


Figure 1: *Left: Whole-image classification, resulting in a single label for the whole image. Right: Example semantic segmentation, labeling each pixels and regions with their respective object class.*

2.2.1 Whole-image classification

In order to avoid confusion with terminology, it is emphasized that this subsection concerns the task of classifying an entire image with a single output label to describe the image's content. An early example of this type of a CNN is the LeNet-5 network, for recognizing letters and words [5]. What characterizes these networks is that they use an entire image's worth of information for giving a single output label describing the image. This label can either be a single letter, an entire sentence, or similar. Note that, when using this technique no spatial information is preserved from the input image to the network output. This means that in most cases one can not recreate the original image solely given the output. For instance, given an image with a chair in the top right corner the whole-image classification will, hopefully, give the output "chair". But only given "chair" as an input it's impossible to recreate the spatial information and thus the knowledge about the chair's location is lost.

2.2.2 Semantic segmentation

Semantic segmentation concerns the task of pixel- or region-wise classifying an image with class labels. The output is thereby multidimensional, commonly with format $W \times H$ as in image width and height. The image is divided into segments containing the spatial positions of each classified object. A prominent recent example of semantic segmentation is the Fully Convolutional Neural Network (FCN)[6] where whole-image classification networks like the above mentioned LeNet are converted into networks for semantic segmentation. This is done by slightly modifying the pre-trained classification networks combined with implementation of layers for upsampling the otherwise non-spatial output via operational layers known as deconvolutional.

2.3 Data and preprocessing

The procedure of getting a CNN being able to perform a desired task, for example semantic segmentation, consists of a couple of steps. The first step is to train the network with a set of data, a training set. During the training, a validation set is used to check how well the CNN is being trained. The validation set and the training set consists of different images. Lastly, when the network has been fully trained, a test set determines how well the CNN is able to perform its task. More about this in section 2.3.1.

Merely using the original images in the training is possible, but in order to facilitate a faster learning process one usually do some sort of pre-processing on the images. The two most common methods of pre-processing, mean subtraction and image normalization, are discussed later on in section 2.3.2.

2.3.1 Splitting data

When a well-defined network architecture has been created, the main task is to begin the practical considerations of how to split available data into a train-, a validation- and a test set. The training images and segmentations will directly alter the various network layer's filters and biases via the train method of back propagation (see section 2.4.3). This data set will thereby determine what object features trigger the network into classifying pixels to a certain class or category. Thus, a lot of training data with a wide variety of characteristic features under various conditions is desirable for every given object.

The validation data is also actively part of the training process. It possesses the functionality of validating whether or not alterations made to layer filters and biases after each training epoch actually improves object classification for data other than what's in the training set. Validation data thereby measures the ability of a network to generalize learnt features from training into correctly classifying new data. Poor validation statistics whilst learning gives strong indication that the model will not generalize well and perform satisfactory on other data than the train set. Validation data thereby alters the training process so that only training with good validation statistics should ever be allowed to reach the final stage of network evaluation.

The test data is only used once a network with satisfactory training characteristics has been obtained, meaning a well-performing model for both train and validation data. Test data serves to evaluate performance of the network, on images that has had no part in the training process.

An issue that can go critically wrong in splitting data is if one does not shuffle and randomize the available data. If multiple images containing the same object (or the same environment or likewise) are in the same set of data, the lack of diversity might become a problem. If said scenario occurs for multiple images in the train set, the network might train and learn very specific features for just this one instance of an object. In worst case these features do not generalize well into other instances of the object or setting. Shuffling the data set is a preventive measure to avoid this.

Another practical issue with splitting data is how much data one ought to allocate to each set. As a general rule of thumb, one usually splits available data into 80% train and validation sets and 20% in the test set. A common split, in percentage units, of train and validation sets is 60%, 20%. These numbers are in no way fixed or mathematically proven to yield optimal result. What one considers when splitting data is merely that each set has sufficient data to somewhat prove its purpose.

2.3.2 Preprocessing

Any good data set will be diverse in order to include all possible data features under various conditions as previously stated. Whilst this kind of data is desirable for producing a generalizable network, this also introduces computational issues. Learning from far too diverse data, the layer weights to be adapted can have difficulty converging upon values that can satisfy all present diversities and extremes in the data. Two significant pre-processing methods can be applied to handle these issues. Common practice is to apply them in the order of presentation.

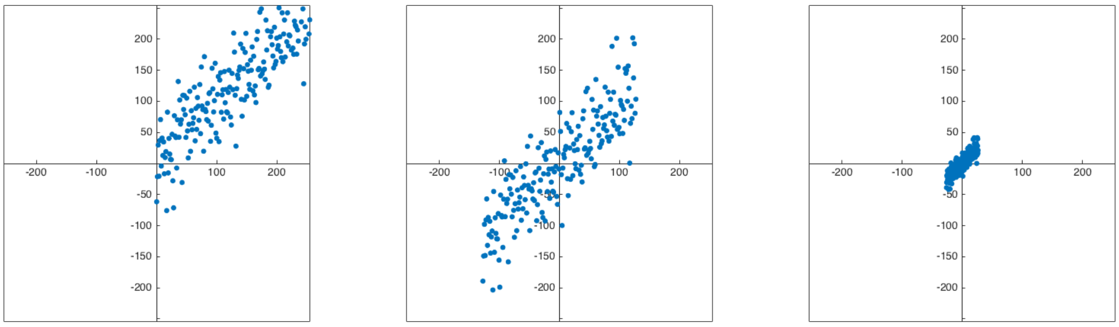


Figure 2: *This figure shows a geometric representation of the preprocessing stages. On the* ***left*** *is the original data, the* ***middle*** *image shows the data after mean subtraction and on the* ***right*** *is the result of normalization.*

Mean subtraction is a commonly used technique when preprocessing data. A geometric interpretation of the operation is zero centering the data. The way this is implemented depends on the given data set. In the case of the data being images, in other words multidimensional matrices, the preprocessing is applied by removing the mean of every color channel from all matrices. Mean subtraction can be implemented in a few slightly different ways. The mean subtraction used in this paper is presented in equation (2) and subtracts the mean individually for every color channel in all images. Here, a_{ij} denotes a pixel in a color channel with coordinates i, j and after mean subtraction, it's expressed as a'_{ij}. Also H denotes the height and W the width of the color channel.

$$a'_{ij} = a_{ij} - \frac{1}{HW} \sum_{h=1}^{H} \sum_{w=1}^{W} a_{hw} \tag{2}$$

Normalization serves to rescale data into a common range of possible numerical values. In the case of images that already are in range [0, 255], this is not strictly necessary, but can improve performance in some cases. There are in practice two ways of normalizing an image: division with the standard deviation of every color channel or rescaling the values to lie within the unit scale $[-1, 1]$.

2.4 Basic operations of a Convolutional Neural Network

A CNN uses both forward and backward propagation in order to train the network. In order to obtain an output, forward propagation is used. This output is compared with the true value to calculate the error via a loss function. As the error ought to be as small as possible a minimization problem has arisen. So, to minimize the error, backward propagation is used to calculate the error derivatives which in turn are used by the stochastic gradient descent to alter the weights. All this is more thoroughly described in the following subsections.

2.4.1 Forward propagation

Forward propagation, or forward pass, is the "flow of information" for a network from input layer to output layer. When running an image through a CNN, that is, doing a forward pass, the CNN produces a score for each particular class in that image. In other words, there is a score function that maps the image pixels to class scores. An example is shown in Figure 3 with the function $\bar{f}(\bar{x}, \bar{y}, \bar{z}) = (\bar{x} - \bar{y}) \cdot \bar{z}$. By splitting this function into two, the following expressions are obtained: $\bar{q} = \bar{x} - \bar{y}$ and $\bar{f} = \bar{q} \cdot \bar{z}$. Let's say input values are $\bar{x} = 5$, $\bar{y} = 1$ and $\bar{z} = -3$, then the output becomes $\bar{f} = -12$ when making a forward propagation. See the green bold numbers. To check whether the network is correctly classifying the objects in the image, a correlation between the score and the ground truth labels is needed. Ground truths are the correct, usually hand-made, classifications of the data that the network strives to replicate when classifying images. The correlation between the score and the ground truth is called a loss function and describes the deviation between the score and the ground truth.

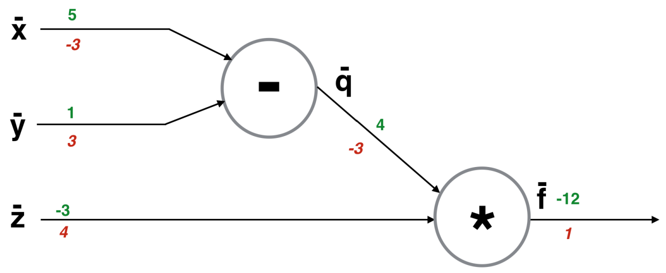


Figure 3: *Example of forward and back propagation with inputs variables $\bar{x}$, $\bar{y}$, and $\bar{z}$ and functions $\bar{q} = \bar{x} - \bar{y}$ and $\bar{f} = \bar{q} \cdot \bar{z}$. The green and bold ones are the values from input to output (forward pass). The red and Italic values is the result of a back propagation and shows the gradients from the output back to the input.*

2.4.2 Loss function

A loss function is used to give a quantitative expression of how near the network's prediction is to the ground truth. A high value of loss is most commonly interpreted as a bad prediction and a low value as a good prediction. The loss function is only part of the network at training time and is located at the end of the network. The learning algorithm later uses the loss value to update the weights of the network. There are many different ways of implementing the loss functions for different data but one of the most common one is by the use of a softmax layer, see section 2.5.5. In the softmax layer, the loss function 'logarithmic loss' is used and is defined by equation (3). Input to the function are the scores given from forward propagation X and the correct categorical labels c.

$$\ell(X, c) = \log(X(c)) \tag{3}$$

2.4.3 Backward propagation

Backward propagation, or backprop, is applying the chain rule several times in order to compute the error's gradients with regards to the weights. The gradient is used to update the weights in direction of the steepest descent in order to make the loss as small as possible. Common practice is using some gradient method for the minimization, for example stochastic gradient descent (SGD). Backward propagation calculates the gradient of the loss with respect to a certain combination of weights $\mathbf{w}$. Let's take an example of a small CNN consisting of one layer; $z = f(x, w)$ where x and w are inputs. When running this CNN a value of z will be obtained. Let's denote the loss L. The gradient of L with respect to z, $\frac{\partial L}{\partial z}$, is known. We want to calculate $\frac{\partial L}{\partial x}$ and $\frac{\partial L}{\partial w}$ in order to investigate how the obtained loss depend on the input and its parameters. Since the function $f(x, w)$ is known and differentiable, this can be done through the chain rule: $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w}$. Once this has been calculated the weights can be updated in the direction of the gradient and thus tweaked to perform better next iteration. Equation (4) shows how SGD updates the weights. Note that η denotes the learning rate, or step size, and sets how far in the direction of the gradient the weights will be tweaked in each iteration.

$$w^n = w^{n-1} - \eta \frac{\partial L}{\partial w} \tag{4}$$

Taking a numeric example, the same as in 2.4.1, the derivatives can be calculated and are shown in red italic font in Figure 3. Starting by the output it's readily apparent that $\frac{d}{dz}\bar{f} = \bar{q} = \bar{x} - \bar{z} = 5 - 1 = 4$. From $\bar{q}$ to the input $\bar{x}$ and $\bar{y}$ the derivatives are $\frac{d}{dx}\bar{q} = \frac{d}{dx}(\bar{x} - \bar{y}) = 1$ and $\frac{d}{dy}\bar{q} = \frac{d}{dy}(\bar{x} - \bar{y}) = -1$ respectively. In turn the derivatives of the loss with respect to $\bar{x}$ and $\bar{y}$ becomes $\frac{d}{dx}\bar{f} = -3$ and $\frac{d}{dy}\bar{f} = 3$ respectively. When having a more complex CNN than the one shown in equation 1, the same principle is applied, but in greater scale.

2.5 The layers of a Convolutional Neural Network

A CNN consists of different layers. Each layer process particular features of the original image. What's in common for all these layers is that they're all taking a 3D volume as input and producing a 3D volume as output using a differentiable function. In this section the most commonly used ones are presented and explained. Henceforth it's assumed the input volume has dimensions $W_1 \times H_1 \times D_1$.

2.5.1 Convolutional layer

The convolutional layer (CONV) is the foundational building block of a CNN. It is made up of an arbitrary number of filters K, with a spatial area denoted as F. Every filter convolves, or "slides", horizontally across the spatial dimension of the input and computes a dot product between the input pixel data and the neurons in the filter, as shown in Figure 4. This results in a 2D activation map for that particular filter. Note that every filter perform its operation through the whole depth of the image, that is along the whole dimension D_1 of the input. This implies that the depth of the filters is always equal to the depth of the input data. The amount of weights for each filter is calculated as a product of the width, height and depth of that filter. The activation map for each filter is stacked depth-wise to produce the output of the convolutional layer: $W_2 \times H_2 \times D_2$, where $D_2 = K$. The filters are often relatively small, for example 3×3, and are learnable. This means that the network can learn the filters to activate for a specific feature in the image.

The spatial dimension of the output depends on a couple of factors; the input dimension, the filter size, the stride and the zero-padding. The two first have been covered. The stride is the rate with which the filter slides across the input data. If the stride is S, the filter will slide S amount of steps between every calculation of the dot product. Let's take an example. If the input data has dimensions $8 \times 8 \times 3$ and we use a filter of dimension $4 \times 4 \times 3$ with $S = 2$, the

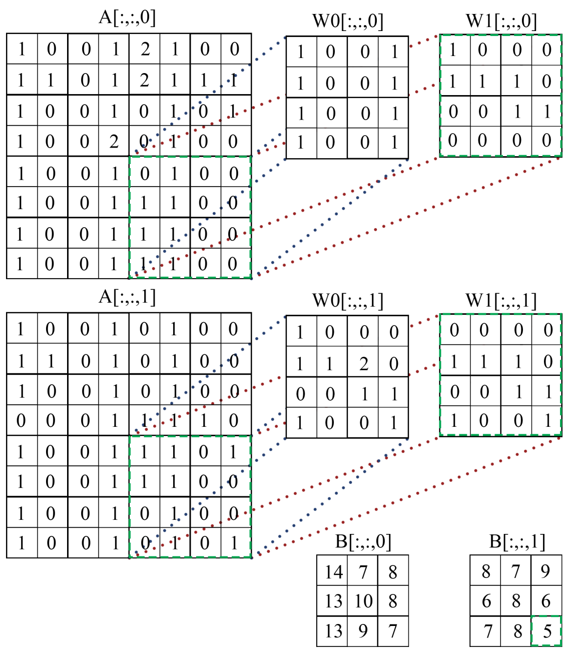


Figure 4: *In this example A is the input to a conv layer having the dimension 8x8x2. There are two filters $W0$ and $W1$ with stride $S = 2$, both having the dimension 4x4x2. B is the output of the conv operation. The input area and filters involved in the calculation of the output value are shown with green dashed lines.*

filter will be able to slide two steps horizontally before vertically dropping down S steps and doing it all over again. Hence, the width W_2 will be equal to 3, and so will the height H_2. If we would change the stride to 3, a problem would arise, since the filter would not be able to completely "cover" the input data. One handy way to solve this, is to use zero-padding. This is a process where P pixels with value 0 are added along the outer border of the input data like a frame. By using a zero-padding of 1 in the example above, the input dimension would change to $10 \times 10 \times 3$ and with $S = 3$, the resulting spatial output dimension would be 3×3. The technique of using zero-padding is also very handy if there's a desire that the input and output should have the same spatial dimension. Let's take another example to illustrate this. If the input data has dimensions $W_1 = 5$, $H_1 = 5$ and $D_1 = 3$ and the filter has a receptive field $F = 3$ with $S = 1$, the output area will be 3×3. If we add a zero-padding of 1, the output will instead be 5×5, the same as the input area. The formulas to calculate the output width W_2 and height H_2 are thus given by equations (5) and (6) respectively.

$$W_2 = \frac{(W_1 - F + 2P)}{S} + 1 \tag{5}$$

$$H_2 = \frac{(H_1 - F + 2P)}{S} + 1 \tag{6}$$

As covered earlier, the output depth is given by

$$D_2 = K.$$

The four parameters K, F, S and P are hyperparameters.

With a filter ω with field size F the input to the layer x_{ij}^{ℓ} is shown in equation (7). Here the contributions from previous layer cells are summed up. The output y_{ij}^{ℓ} after the convolutional layer's non-linearity is applied is shown in equation (8).

$$x_{ij}^{\ell} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} \omega_{ab} y_{(i+a)(j+b)}^{\ell-1} \tag{7}$$

$$y_{ij}^{\ell} = \sigma(x_{ij}^{\ell}) \tag{8}$$

2.5.2 Rectified Linear Unit

Rectified Linear Unit (ReLU) is a unit that implements a rectifier, which in turn is an activation function defined as $f(x) = \max(0, x)$. That is, it takes a neuron x as input and sets the value to zero if x is negative and does nothing otherwise. ReLU works elementwise, and hence leaves the dimensions unchanged.

2.5.3 Pooling layer

A pooling layer (POOL) is a layer that performs downsampling, that is, it uses a certain kind of operation to change the size of the volume along the first and second dimension, the spatial ones. The most common type of pooling is MAX pooling, which could typically work by sliding a filter of dimensions 2×2 along the 3D volume and picking the maximum number of the four numbers within the filter. The filter moves with a specified stride, i.e. the amount of pixels it moves across with each step.

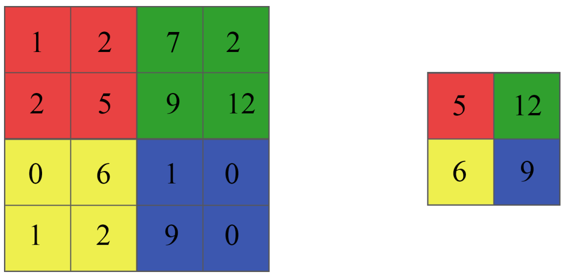


Figure 5: *The left picture is the input to a max pooling layer with filter size of 2×2 and a stride of 2. To the right the output is shown, each colored section is the maximum value of each 2×2 sub-block of the same color in the input.*

2.5.4 Fully Connected layer

The fully connected layer (FC) is a special case of a CONV layer. The layer is connected to all the activations of the previous layer hence the name fully connected layer. The layer is commonly used in the later part of the network to store high level abstraction information, therefore needing information from all the neurons from the previous layers. Just like the conv layer it has weights and biases for the learning process. The main difference is the size of the filters, because since no spatial information is needed at this stage, the filters are only of size 1×1.

2.5.5 Softmax Loss

Softmax loss (SOFTMAXLOSS) is one of the most common implementations of a loss function for image data and CNNs. It combines the softmax function with the logarithmic loss function defined in section 2.4.2, and is preferable in most cases due to its probabilistic interpretation. In comparison to another popular loss function, Multiclass Support Vector Machine Loss (SVM), the softmax loss classifier has a normalized output and is easier to use and interpret. The resulting expression combining the loss- and softmax functions is shown in equation (9) where x is the output vector with probabilities from the network, c is the class for which the loss is calculated and C is the total number of classes.

$$\ell(x,c) = -log\frac{e^{x_c}}{\sum_{k=1}^{C} e^{x_k}} \tag{9}$$

2.5.6 Batch Normalization layer

The batch normalization layer (BATCHNORM) is commonly used before the ReLU layer and speeds up the training process of the network. The layer applies normalization to a whole batch of data and thereby lowers the internal covariate shift of the data. The technique behind batch normalization will be explained later in section 2.7.4. Equation (10) implements the batch normalization layer, where y_{ijkt} is the normalized data batch having dimensions $H \times W \times K \times T$ where $H \times W$ is feature size, K is feature channels and T is batch size. The input x_{ijkt} has the same dimensions as the output y_{ijkt}. The weights and biases are represented by w_k and b_k, both being tensors of size K. Also ϵ is a constant added for numerical stability. The mean value μ_k of every feature channel of x is calculated by (11) and $\sigma^2{}_k$ is the variance of every feature channel of x calculated in (12).

$$y_{ijkt} = \omega_k \frac{x_{ijkt} - \mu_k}{\sqrt{\sigma^2{}_k + \epsilon}} + b_k \tag{10}$$

$$\mu_k = \frac{1}{HWT}\sum_{i=1}^{H}\sum_{j=1}^{W}\sum_{t=1}^{T} x_{ijkt} \tag{11}$$

$$\sigma^2{}_k = \frac{1}{HWT}\sum_{i=1}^{H}\sum_{j=1}^{W}\sum_{t=1}^{T} (x_{ijkt} - \mu_k)^2 \tag{12}$$

2.5.7 Deconvolutional layer

The deconvolution layer is the reverse of a convolution layer. As a conceptual parable, one can view the deconvolutional layer as a stamp that applies learnt filters onto the output feature map with the input feature map as weights. A visualization of the concept can be seen in Figure 6. The padding of a convolutional layer here acts as a crop of the output and the filter stride as an upscaling factor. The overlapping parts of the output are being added together.
A more formal description of the operation is presented in equation (13). Where x is the input feature map with dimension $H \times W \times D$, f is the filter with dimension $H' \times W' \times D \times D''$ and y is the output feature map with dimension $H'' \times W'' \times D''$.

$$y_{i''j''d''} = \sum_{d'=1}^{D} \sum_{i'=0}^{q(H',S_h)} \sum_{j'=0}^{q(W',S_w)} f_{1+S_h i' + m(i''+P,S_h),1+S_w j' + m(j''+P,S_w),d'',d'} \times x_{1-i'+q(i''+P,S_h),1-j'+q(j''+P,S_w),d'} \tag{13}$$

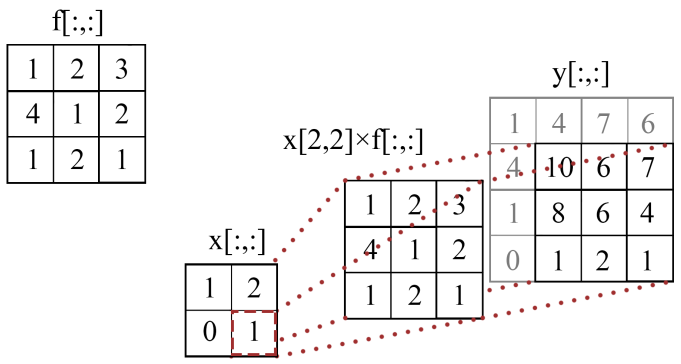


Figure 6: *In this example x is a 2×2 input and f a learnt filter of size 3×3. Upsampling is set to one resulting in the output size 4×4 of y. The crop is set to zero, resulting in keeping all of the features in y. The last upsample step is shown in red, element $x[2,2]$ is multiplied as a scalar with f, then added to the marked position of output y.*

Here S_h and S_w are the upscaling factors for height and width, also P is the crop. The function $m(k, S)$ is implemented by

$$m(k, S) = (k - 1) \bmod S\,, \tag{14}$$

whereas $q(k, n)$ is described by

$$q(k, n) = \left\lfloor \frac{k-1}{S} \right\rfloor . \tag{15}$$

2.6 Training a Convolutional Neural Network

Training a CNN is equivalent to adjusting the layers' weights towards classifying the training data correctly. The basic flow of the training consist of a couple of fundamental operations already mentioned in 2.4. Firstly a batch of data is sampled. This batch is then forward propagated through the network in order to obtain a classification and a loss. Using the loss a back propagation is made leading to a calculation of weight gradients and different parameters are updated through the SGD. Repeating this process leads, hopefully, to weights making the CNN converge towards a low loss resulting in a correct classification of the input. Each iteration of aforementioned process on the whole training set is called an epoch.

The input data to a CNN is partly made up out of two structures: all the images in the training and validation sets and corresponding labels for each pixel in these images. The images are used in the forward propagation of the network and their labels are implemented when calculating the loss. All this data is provided in matrices with dimension $W \times H \times D$, where W and H is the size of the image, width and height respectively. D denotes the depth of the image and in RGB pictures this is equal to 3, one for each color. In the label matrix, each number usually corresponds to a specific class, for example 1 is equal to a floor, 2 is equal to a table and so on. This data is referred to as the ground truth. One part of the input data is used for training and the other part is used for validation. In section 2.8 the reasons for this is discussed, but generally it's because it helps to detect whether the loss converge towards zero or not.

To summarize: the first step in the training process is a forward pass and loss calculation. The next step, back propagation, calculates the gradient of the loss with respect to the input weights.

As a final step the SGD updates the weights in order to perform better next iteration. This process of three steps is then repeated all over again until a decent loss has been reached.

2.7 Challenges and common solutions when training

Some challenges are recurrent for everyone dealing with CNNs and here overfitting, data augmentation, dropout and batch normalization are discussed. These are the most common tricks to overcome pit falls and hence the most important.

2.7.1 Overfitting

Overfitting is a common problem related to training a CNN. The problem is characterized by having a low training error at the same time as the validation error remains high (see section 2.8 for training and validation error). In practical terms this means that the network accurately can categorize the training data but the features learned are not generalizing well to the concepts of the whole data set and therefore the network cannot categorize data outside of the training set with great accuracy. The reason to why the network overfit the data depends on a number of different things. The amount and diversity of data is crucial to minimize overfitting. Having a data set of just 10 data samples would yield a network that predicts the training set very well but would most likely not perform well on other data. By increasing the amount of data, generally the performance of the network increases as well. The complexity of the network can also be a reason for overfitting. When the number of parameters in the network is increased the network is able to take more complexity of the data into account but if the complexity does not increase i.e. the amount of data is the same, the risk for overfitting increases instead. It is not always possible to increase the amount of data for different reasons. Instead data can be created artificially, this method often referred to as data augmentation is very useful and will be explained in 2.7.2. Another standard method for handling overfitting when training networks with many parameters is dropout. The main concept is to drop connections in the network randomly during training to prevent the network from co-adapting in greater extent. A more in-depth explanation of dropout can be found in section 2.7.3.

2.7.2 Data augmentation

Data augmentation describes a widely used method for creating artificial data from the original data set. The main idea behind the technique is to alter the original data randomly in such a way that the new data is slightly different but still represents the same concept. In the case of the data being images, by for example randomly altering the rotation of one image, more images can be generated still promulgating to follow the concept of the original image. When it comes to images there are many different ways to achieve this. The most common methods of augmenting image data are rotating, resizing, flipping and translating the data. Another way of generating more image data is random occlusion. By randomly covering random sized patches of the images more image data can be generated.

2.7.3 Dropout

Dropout is used to regularize neural networks and reduce overfitting. When networks with many parameters train with small data sets the risk for overfitting increase. This results in a network that predicts the training data well but does not generalize well. This due to taking random sample noise from training data into account instead of learning the underlying concepts of the data. The dropout technique propose a way to regularize the network and prevent it from building up relationships on the data noise. By dropping connections between different layers with a certain probability during training, the network becomes less reliant on all of the different parts of it and therefore more diverse. It can also be viewed as training many thinner networks with shared weights in parallel and sampling them at test time. See Srivasta et al. [7] for more extensive details.

2.7.4 Batch normalization

Batch normalization is a technique for normalization at each layer of the CNN. The batch normalization should be applied before the activation function layer. It was developed to handle the so called internal covariate shift i.e. the shift in the distribution of the input data in between layers of the network. This is a problem specifically in deep networks because of their many parameters. The input of a layer depends on all previous parameters. With the many parameters of deep networks even small changes of the weights get amplified in deeper layers. This has earlier been handled by using small learning rates and being careful when initializing the weights of the network. With batch normalization the internal covariate shift is reduced allowing for higher learning rates and higher tolerance in the initialization of the weights. It has been shown by, amongst others Ioffe et al. [8], to be very useful because it also has regularization properties reducing the need for dropout in deep network architectures.

2.8 Evaluating a Convolutional Neural Network

In order to compare CNNs and to know whether an improvement has been made or not, some standardized measurement of success ought to be mentioned and defined. A test is performed after each epoch of training and an example of the loss is plotted in Figure 7. In this example only 23 epochs have past and the loss is still large. The training graph (the blue one, commonly below the red validation graph) shows the error running the CNN with the same images as it has been trained with. The validation graph on the other hand tests the CNN with images that haven't affected the weights and has therefore often higher error. In the top-1 error graph, only the top class (the one having the highest probability of being correct) is compared with the target label (ground truth). In the top-5 error the comparison between the top five most likely predictions and the ground truth is made. The top-1 and top-5 error can give a guidance whether the network is having problems to distinguish between the different classes, for instance if the top-1 error remains high when the top-5 error is much lower. The objective is the max log-probability average among training cases of the correct label or in other words the output of each loss layer. Looking at the graphs is the easiest way of monitoring the network's progress and detecting glitches or failures, some of which were mentioned in 2.7.

2.8.1 Evaluating semantic segmentation

In contrary to the aforementioned way of comparing structures, Jaccard index is calculated after the training is done using the part of the data set still untouched. This is a more universal measure of the performance of a fully trained network and is defined as the size of the intersection divided by the union of the sample set size, see equation (16). In this equation M_1 is the output of the network and M_2 the ground truth. There are many ways of implementing the Jaccard index and the one used in this report is elaborated in section 3.5.1.

$$J(M_1, M_2) = \frac{|M_1 \cap M_2|}{|M_1 \cup M_2|}\,. \tag{16}$$

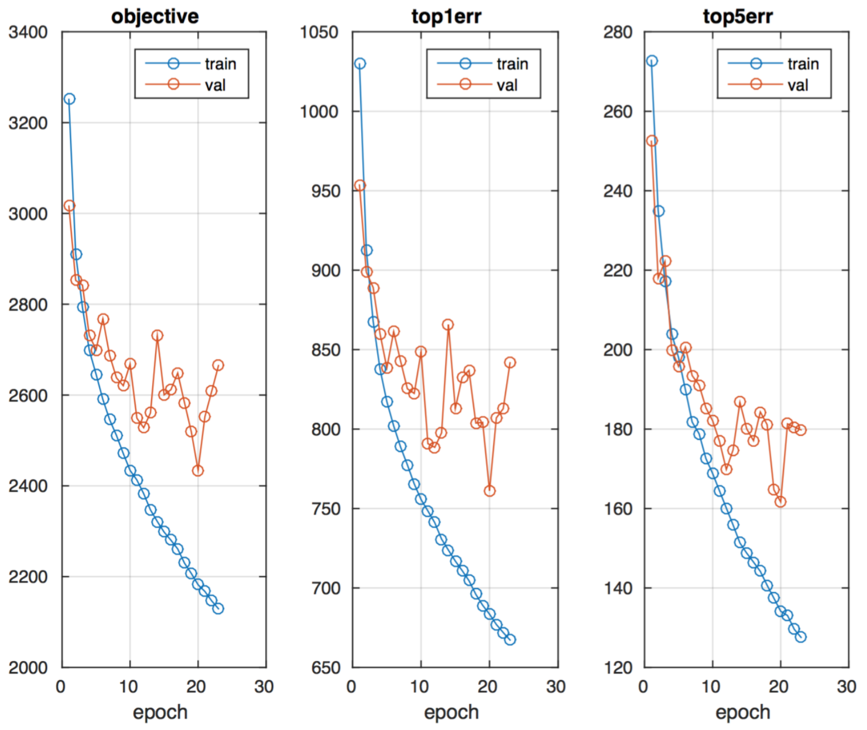


Figure 7: *Example of how the objective function and errors vary with epochs past. Successful training will display a, preferably monotonous, decrease for all data points as epochs pass. The objective gives a measure of how well obtained segmentation correlates with ground truth. The top1err and top5err graphs show the most likely segmentation predictions for the top one and top five classes respectively, as described in section 2.8.*

3 Method

The procedure of this project can be divided into two stages: gaining knowledge about deep learning and developing a CNN for indoor scene understanding. First off, project members did literature research on merely the basic concepts of deep learning. As concepts became familiar, tutorials on whole-image classifications were attempted. Knowledge from these tutorials were then extended to writing a letter recognition network for the MNist data. The deep learning techniques were then applied in learning semantic segmentation, starting with binary segmentation.With binary segmentation understood, the next phase of the project begun by starting to work with the chosen data set and multi-class segmentation. The method for obtaining final network architecture is outlined in the following sub sections. Firstly the data set is treated. Secondly the software used and how they were set up are accounted for. Also how experiments were conducted in order to achieve knowledge about different architectures is here specified. Furthermore the method of how to obtain an algorithm performing spatial localisation is described and as a final part how the network evaluation was done is presented.

3.1 Choice and usage of the data set

For the final program, the vast data set SUN [9] was used. This data set includes images from three other papers by N. Silberman et al. [10], A. Janoch et al. [11] and J. Xiao et al. [12].

Table 1: *Chosen classes of indoor objects to learn. The general idea for choosing these classes is that they represent common objects that are either large and easy to orient by, or somewhat smaller objects that carry significance in indoor environments. The number before each object represents the label given to each pixel.*

1: Chairs	10: Printer	19: Person
2: Floor	11: Fax machine	20: Monitor
3: Garbage bin	12: Coffee machine	21: Shelves
4: Refrigerator	13: Sofa	22: Cabinet
5: Wall	14: Lamp	23: Door
6: Laptop	15: Bed	24: Table
7: Computer	16: Bench	25: Small Containers
8: Keyboard	17: Stairs	0: Background
9: Window	18: Piano	

Since this data set includes over 800 various objects, which extends far beyond this study's goal, images had do be extracted and reformatted. This had to be done in accordance with the study's chosen classes and specific task. The chosen classes are some frequently seen indoor objects that may be of interest in detecting and are specified in Table 1. The number before each object name is the class number corresponding with a class. After reformatting the data set, it consisted of approximately 10,000 images.

3.1.1 Reformatting the data set images

The images containing at least one of the chosen objects had to be extracted from the SUN data set. This was done by looping through the data set folder structure, and for each image comparing the list of classes contained in an image with the classes chosen for this study. Not only images, but also corresponding ground truth label were then saved into a new file structure where they later had to be further processed. See Appendix C for the entire algorithm and its help functions.

To get manageable data sizes to work with, images were rescaled to 200×200 pixel size format. As motivated in section 2.3, these image were also pre-processed with mean subtraction. With both aforementioned operations made on the images, they were saved as new image files. The cost in drive space was weighed as far less precious than the substantial time saved for having these images saved and loaded into training rather than repeatedly being processed in run-time.

3.1.2 Reformatting the data set labels

With images in a satisfactory format, labels were next up for processing. First off, the net structure reduced images with size 200×200 to 50×50. Ground truth labels therefore had to be resized into 50×50 format. As a preliminary note of caution, this procedure is technically not in line with the concept of segmentation where an image with dimensions $H \times W \times D$ ought to be segmented into an image of dimension $H \times W$. This is further discussed in section 5.3.4. The algorithm created for reformatting labels in this project can be seen in Appendix D.

The original ground truth labels were to be given new numeric class values, in the range 1 to 25, and to be resized into format 50×50. The numeric alteration is both necessary for the functionality of computing loss with the MatConvNet softmax layer [13] but also desirable for convenience. A different approach for the labels than when rescaling images had to be taken. When downscaling images, interpolating RGB numeric values in the images is not a problem.

With fixed integer labels however, altering these numeric values with Matlab's default image resizing would introduce intermediate decimal numbers and would distort the segmentation to a useless state. Resizing labels was instead done via extracting a segmentation matrix for each separate class value in the segmentation. These segmentation matrices, specific for a single class, were given as binary matrices of zeros and ones. The information contained in these matrices were, in a binary state, resized to desired 50×50 format, and thereafter given a new numeric label in the range 1-25. These separate matrices were finally combined into a new ground truth. If two labels were assigned the same position (which very rarely happened) in the combined matrix, the lowest got preference. An alternative way, discovered later on, is downscaling labels via nearest neighbor method. All label images were also saved as separate files, in the same folder as the corresponding image. More specifics on downscaling and a segmentation comparison of the two methods can be found in Appendix D.

3.2 Matlab implementation details

The only software used in this project was the CNN toolbox MatConvNet, for Matlab, as per recommendation from the project supervisor. Toolbox MatConvNet requires a lot of computational power. Using a GPU instead of a CPU for processing is highly preferable as it can increase computational speed significantly. Also, two types of CNN wrappers are available in MatConvNet, training networks either with ordinary Matlab syntax or in an object oriented environment. Respective wrapper's are called SimpleNN and DagNN.

3.2.1 Choice of MatConvNet-wrapper

SimpleNN is a the somewhat easier wrapper to use and understand. Networks are created as structures, containing various sub-structures such as network layers and meta data. These sub-structures contain various network information lower in the variable hierarchy. Numerics, such as convolutional layer parameters, are stored in cells. All numerics must be stored in "single" data format. This prerequisite pays off in considerable faster computations in the various computation blocks. The SimpleNN-wrapper allows only for straight forward- and backward passes of data through the network.

In DagNN, networks are instantiated as objects with certain properties. One adds layers to the object after network instantiation. A fundamental difference from the SimpleNN-wrapper is that one specifies two indices for layer data input and output. This allows for customized passes of data between various layers, as desired. One can also share parameters between layers in the DagNN-wrapper, by specifying "fanin" and "fanout" for layers and variables. As a final note regarding the wrappers, a thing to keep in mind is that they both use the same core computational blocks.

Having no need for other data flow than straight forward- and backpasses, SimpleNN was used for the project's final network. This was the preferable choice, partly for the simplicity in usage but also because the DagNN tends to be slower in computation for small networks like this [14].

3.2.2 GPU drivers for faster computations

Where CNN computations and learning can be done with CPU processing, computing on a modern GPU has the potential to speed up the learning a lot. In this project, Nvidia graphic cards along with Cuda and CudNN software are utilised for extreme improvements in performance. A reference value is that computational speed can improve up to 50 times with GPU processing, on appropriate hardware and drivers [15]. Worth mentioning is that a GPU with Cuda capability larger than 3.0 is needed for utilization of CudNN. Cuda capability is a score based on supported functions by different GPU architectures [16]. Methodwise, GPU:s alter the training process in such a way that images are stored onto the GPU:s RAM memory and there internally processed.

3.3 Experiments - network architectures

Experimentally, to obtain the final network structure, three network types were initially tested training on merely a single image. That image was used as both the sole image of the train- and validation set. This was followed up by test training on a small data subset of 100 images, with data set split of 70/20/10 for respective train/val/test sets. The network design to achieve best performance in these trials was then chosen for the later large scale training. This general method, obviously not in line with the concept of deep learning, was used to obtain an initial understanding of how the network output would come to look like without waiting several days or even weeks for results. The network architectures were designed as in Appendix B and are outlined in this section.

ConvRedTo4th was the design to be chosen for final implementation. The network follows a pretty common pattern of having layer sequence [Conv− >ReLU− >Pool], but has additional batch normalization layers implemented in between the convolutional and ReLU layers. It's a simple network, implemented with straight forward- and back passes, but common for the reason that it performs well on most data. The network relies on early convolutional layers learning spatial information from the input image, wheras pooling and forward passes emphasize image features for later convolutional layers to learn from. Early convolutional layers span a wide receptive field in order to extract as much spatial information as possible. In return, fewer filters are used in the earliest layers, as fewer features are immediately available for processing. As the image is reduced in spatial dimensions, but also in unnecessary "image noise" - irrelevant image information -, later convolutional layers use a slightly smaller receptive field but instead stack more filters. More filters enable the learning of more features. After each convolutional layer, a layer of batch normalization is implemented, in order to normalize and center output data and to thereby facilitate a faster learning process. Implementing the batch normalization before the non-linearity ReLU layer ensures more effective and stable activations, as further described in the original Batch Normalization report [8].

Deconv_light had basically the same network architecture as ConvRedTo4th but with a deconvolution layer up toward the end functioning as an interpolation filter upsampling the segmentation. Test runs of this network showed decent statistics up to a point, but stagnated at an unsatisfactory level of segmentation despite continued training. The only significant addition of this net beyond the **ConvRedTo4th**-net was an interpolation-filter at the end. Since this can be implemented after a satisfactory training of the core network in ConvRedTo4th with minimal training, this network was discarded.

Deconv_128 was an attempt to create a network with multiple deconvolutional layers. The design was intended to replicate the approximate structure of Noh et al.'s deconvolutional network for semantic segmentation [17] but was implemented with insufficient understanding of how this would be implemented in the MatConvNet-framework. Where the deconvolutional layers are meant to in a way mirror the previous convolutional layers, and with unpooling done with information from previous pooling layers, this is not automatically connected when initializing a new network structure. How the connections would be made in MatConvNet was not understood from reading the documentation. Case study of a network for semantic segmentation, H. Ros' tutorial session 5 [18], did not provide clarity in this instance. The data flow there is straight feed forward, and no other information read from the network dagNN-object or training function provided further insight. Thus, we did not find how a deconvolutional network is supposed to be implemented in MatConvNet.

3.4 Spatial localisation of objects

To make further use of the results obtained from the CNN, an add-on making a spatial localisation of the objects in the image was created. The algorithm uses the segmentation matrix, given as output from the CNN, to determine where in the image specific objects are located. Only objects that cover a certain minimum area are being taken into consideration. The threshold differs for each item, since the classes vary from large tables to small vases. When the program is specifying in what region objects can be found, the image is divided into 9 equally large regions. For each pixel in every object, the x- and y-coordinates are obtained. The algorithm then checks if the x-coordinates lie in the left, middle or right parts of the image, or over several of these regions. The same is done for the y-coordinates, where the areas are defined as top, center and bottom. The program takes into consideration over how many regions the object spans. If an item ranges over two regions x- and y-wise, two regions x-wise and one region y-wise, or vice versa the algorithm checks if these areas are corners and will specify if that is the case. In a similar manner, if an object ranges over the whole span horizontally or vertically, this will be specified in the output. Besides giving an output in the form of a sentence specifying the object and its localisation, the algorithm also prints the class name on the specific object in the segmentation. To find the correct location of where to insert this information, the median value of the x- and y-coordinates for the object is extracted and specified as the position of the text label. The entire code for the algorithm can be found in Appendix E.

3.5 Evaluating the obtained network

This section outlines measures and techniques one can use for evaluating the performance of a convolutional neural network designed for semantic segmentation. Implementation of Jaccard indices and averaging these over the test set is discussed, followed by a short mention of how one might approach uncertainty in the pixel predictions.

3.5.1 Implementation of Jaccard indices

The network performance is evaluated with Jaccard indices for both average image-wise accuracy and average class-wise accuracy. These are mathematically defined via equation (16), but implemented in Matlab as following. Image-wise segmentation is implemented as

$$J_{im,i} = \frac{\text{sum}(\text{sum}(S_i == L_i\ ,2),\ 1)}{50 \times 50}\,. \tag{17}$$

Here S_i is the segmented image matrix, L_i is the ground truth (label) matrix, i denotes image number (say image 5 in a test set of 2000 images) and 50×50 denotes the number of pixels in a label matrix (which is the union of S_i and L_i). Statement $S_i == L_i$ compares the matrices S_i and L_i element-wise and returns a binary matrix. Summing over all 1:s in the binary matrix, one obtains pixels classified the same in ground truth as in the segmentation. One obtains the average image wise accuracy through

$$J_{imAv} = \frac{\sum_i J_{im,i}}{N_I}\,, \tag{18}$$

where N_I denotes the number of test images used.

Class-wise accuracy, for image i, is implemented as

$$\left.\begin{aligned} S_{1,l} = &\ (L_{i,l} == l) \\ S_{2,l} = &\ (S_{i,l} == l) \end{aligned}\right\} \quad \begin{aligned} &\text{union}_{i,l} = \text{sum}(\text{sum}((S_{1,l} + S_{2,l}) > 0,\ 2),\ 1), \\ &\implies J_{cl,i,l} = \text{sum}(\text{sum}((S_{1,l}.*S_{2,l},\ 2),\ 1)/\text{union}_{i,l} \\ &l = 0, 1, 2... \end{aligned} \tag{19}$$

Here l denotes a specific class, say number $l = 1$ which is a chair. $S_{1,1}$ and $S_{2,1}$ thereby contains binary matrices of only chairs for example, taken from ground truth and the obtained segmentation respectively. The union of these binary matrices is obtained by adding the matrices element-wise. This addition gives 2:s for common regions, whereupon statement $(S_1 + S_2) > 0$ returns a binary matrix of the union. For each class then, Jaccard indices $J_{cl,i,l}$ are calculated via element-wise matrix multiplication and division by their numeric union. Intersection $S_1 \cap S_2$ can not be calculated as in equation (18) since comparison $S_1 == S_2$ would here compare the binary matrices 0:s also, that are not part of the intersection between segmentation class prediction and ground truth. After class-wise Jaccard indices $J_{cl,i,l}$ have been obtained, one can utilise a few different methods of averaging over all images. Two ways are specified in the following section.

3.5.2 Harsh or non-harsh evaluation

When evaluating average image wise accuracy, one divides the Jaccard indices $J_{im,i}$ by the number of test images as in equation (18), approximately 2000 in this study. When evaluating average class-wise accuracy however, one can approach the averaging in other ways. The Jaccard index for a single image when calculating class-wise accuracy J_{clAv} is seen in equation (19). But when to decide what images that contain the object, one can choose differently. The defined "harsh approach" is as

$$J_{clAv,l} = \frac{\sum_i J_{cl,i,l}}{N_{L,l} + N_{CS,l}}\,, \tag{20}$$

where $N_{L,l}$ refer to the number of ground truth images with class l in them, and $N_{CS,l}$ refer to the number of segmentation images with the class l in them. If a pixel is contained in either the segmentation or the ground truth to an image, that image is counted when averaging. This means that if an image does not contain, for example, a chair but the segmentation classifies a single pixel in the image to be a chair anyways then this image is counted. Not only does the incorrect pixel classification decrease J_{cl} then as part of the intersection in equation (19), but the image count also increases. This means that added J_{clAv} for various images will be divided by a larger image count, thus decreasing overall average further.

The "non-harsh" approach instead only counts images with ground truth containing a class when dividing by number of images containing the class. In form of an equation means

$$J_{clAv,l} = \frac{\sum_i J_{cl,i,l}}{N_L}\,. \tag{21}$$

The non-harsh approach counts only the portion of pixels correctly classified. The technically correct approach is to use the harsh one, but it can be interesting to measure "stray pixels" in segmentations of other images by comparing the obtained $J_{clAv,l}$:s with the two differing methods.

3.5.3 Uncertain predictions and thresholding

When a pixel is classified to a certain category, it classifies based off of the best class prediction. The network will not classify each image pixel with full confidence, which will show in that numeric predictions for a pixels might have similar value for different classes. Based off of this idea, one might implement thresholding to demand a certain confidence in a pixel predictions. One simple way of doing it is to compare the best prediction with the second best prediction. This is used for this study, where the exact thresholding deems that a good prediction has

$$\text{best prediction} > \text{threshold} \cdot \text{second best prediction}\,. \tag{22}$$

The exact numeric value of threshold parameter used, empirically discovered to yield the greatest increase in J_{imAv}, was about 1,2. Pixel classifications that fail to pass this threshold are classified as background (0:s).

4 Results

The results of this study are divided into what network structure was obtained, how well that network performs in various measures and evaluation of the spatial localisation algorithm.

Table 2: *Final structure of CNN, RedTo4thSize. The net has 19 layers, with convolutional-, batch normalization-, rectifying linear unit-, max pooling- and softmax layers. The network takes an input image of size* $200 \times 200 \times 3$ *and produces a segmentation of size* $50 \times 50 \times 26$ *containing class scores for the 25 classes plus a zero classification for background pixels. Field size sets the quadratic spatial extent of a layer's operation, for example a spatial extent of* 7×7 *for the first convolutional layer. The number of filters (#filters) determine how many filters a layer contains. Both stride and padding have quadratic dimensions, and operate as specified by the theory in section 2.5.1. The network has 2,4 million parameters.*

#	Layer type	Img size	Field size	#Filters	Stride	Pad
1	Conv	200	7	60	1	3
2	Batch norm	200	NA	60	NA	NA
3	Relu	200	NA	NA	NA	NA
4	Conv	200	5	120	1	2
5	Batch norm	200	NA	120	NA	NA
6	Relu	200	NA	NA	NA	NA
7	Maxpool	100	2	NA	2	0
8	Conv	100	5	200	1	2
9	Batch norm	100	NA	200	NA	NA
10	Relu	100	NA	NA	NA	NA
11	Conv	100	5	300	1	2
12	Batchnorm	100	NA	300	1	NA
13	Relu	100	NA	NA	1	NA
14	Maxpool	50	2	NA	2	0
15	Conv	50	1	400	1	0
16	Batch norm	50	NA	400	NA	NA
17	Relu	50	NA	NA	NA	NA
18	Conv	50	1	26	1	0
19	Softmax	50	NA	NA	NA	NA

4.1 Final network structure

The structure yielding the best results is shown in Table 2, with corresponding Matlab code to generate the structure included in Appendix A. The net has 19 layers, with convolutional-, batch normalization-, rectifying linear unit-, max pooling- and softmax layers. It reduces an original image of size $200 \times 200 \times 3$ to an output segmentation with size $50 \times 50 \times 26$ containing class scores for the 25 classes plus a zero classification for background pixels. Field size sets the quadratic spatial extent of a layer's operation, for example a spatial extent of 7×7 for the first convolutional layer. The number of filters (#filters) determine how many filters a layer contains. Both stride and padding have quadratic dimensions, and operate as specified by the theory in section 2.5.1. The network was trained with a learning rate of $2 \cdot 10^{-5}$. The network has 2,4 million parameters and was trained for roughly a week.

4.1.1 Network training statistics

The resulting graph of the network's training process can be seen in Figure 8. There one clearly notes an undesired increase of the objective and top errors for the validation set after 72 epochs whereas statistics for the training set show desired monotonous decrease. The network performing

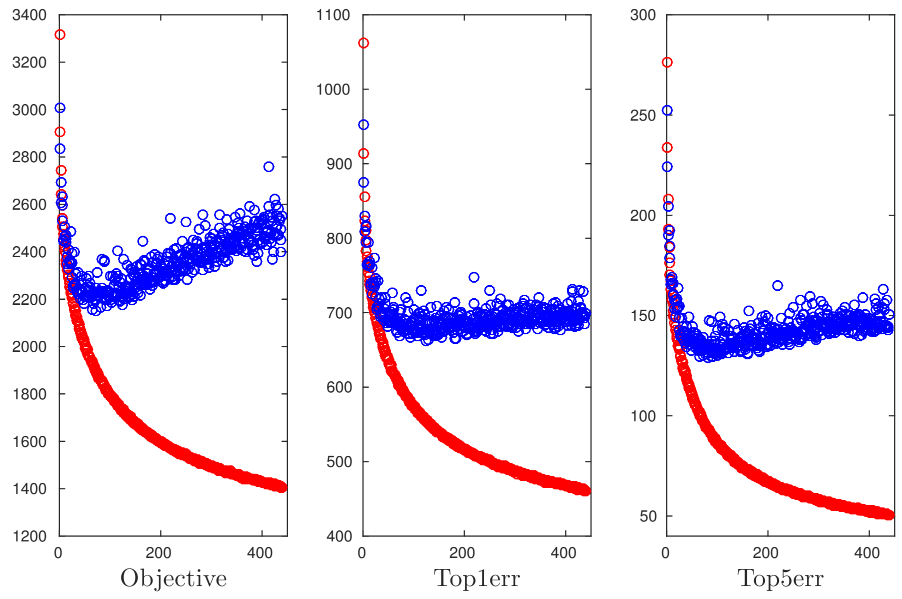


Figure 8: *Statistics for the training of the network. Red data points belong to training set data, blue data points belong to the validation set data. The figures show how objective, top 1 error and top 5 error as described in section 2.8 varies as epochs pass.*

best on the validation data is thereby obtained after 72 epochs. Monotonous decrease in training statistics give that the network performing best on the training data is the latest net, at 439 epochs. These two networks are compared and evaluated in the following sections, with notations net$_{BT}$ and net$_{BV}$ where BT stands for "Best Train" performance and BV for "Best Validation" performance. Where one usually evaluates only the network with best validation statistics, this report takes an unconventional approach of evaluating both aforementioned to further assess the failure in training.

4.1.2 Network performance in Jaccard indices

The networks achieve average image-wise Jaccard indices of $J_{imAv} = 0.3957$ and $J_{imAv} = 0.3939$, for net$_{BT}$ and net$_{BV}$ respectively. Class-wise segmentation gives Jaccard indices according to Figure 9 and Figure 10. Note that the class numbers are shown in Table 1.

Implementing a threshold of 1.2, in accordance with equation (22), gives an increase in average image-wise Jaccard indices to $J_{imAv} = 0.4488$ and $J_{imAv} = 0.4510$ for net$_{BT}$ and net$_{BV}$ respectively. However, looking at the class-wise segmentation Jaccard indices one notes that this increase in average image-wise performance comes much from the 0-classification improving at expense of other object classifications.

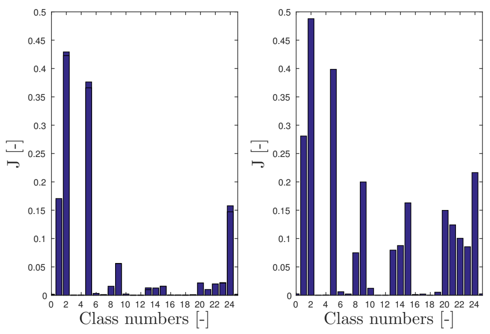


Figure 9: *class-wise segmentation statistics of net$_{BV}$, measured in Jaccard index $J_{clAv,l}$. To the left, one sees the $J_{clAv,l}$ values for harsh evaluation. To the right, the $J_{clAv,l}$ values are given for non-harsh evaluation.*

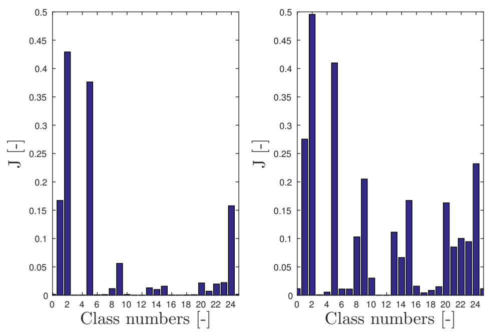


Figure 10: *class-wise segmentation statistics of net$_{BT}$, measured in Jaccard index $J_{clAv,l}$. To the left, one sees the $J_{clAv,l}$ values for harsh evaluation. To the right, the $J_{clAv,l}$ values are given for non-harsh evaluation.*

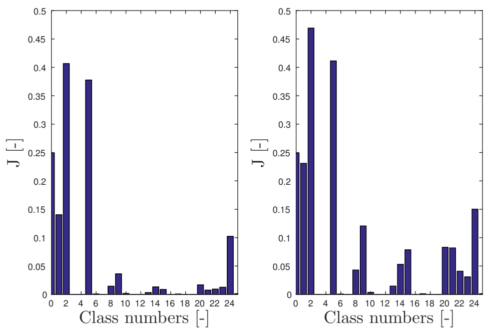


Figure 11: *class-wise segmentation statistics with threshold of net $_{BV}$, measured in Jaccard index $J_{clAv,l}$. To the left, one sees the $J_{clAv,l}$ values for harsh evaluation. To the right, the $J_{clAv,l}$ values are given for non-harsh evaluation. Note that the 0-classification accuracy has increased drastically compared to the corresponding class-wise segmentations without threshold, seen in Figure 9. Importantly, do also note that the increase in 0-classification accuracy is followed by a decrease in other object classification accuracies.*

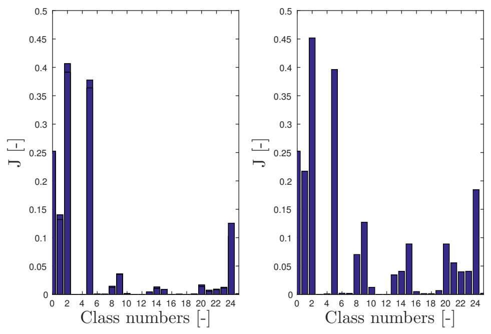


Figure 12: *class-wise segmentation statistics with threshold of net$_{BT}$, measured in Jaccard index $J_{clAv,l}$. To the left, one sees the $J_{clAv,l}$ values for harsh evaluation. To the right, the $J_{clAv,l}$ values are given for non-harsh evaluation. Note that the 0-classification accuracy has increased drastically compared to the corresponding class-wise segmentations without threshold, seen in Figure 10. Importantly, do also note that the increase in 0-classification accuracy is followed by a decrease in other object classification accuracies.*

4.1.3 Example segmentations

Where Jaccard indices give a good quantitative measure of how well the network performs on average, some qualitative assessment in viewing actual segmentations might provide better understanding of the results. This section is dedicated to explaining a few obtained segmentations.

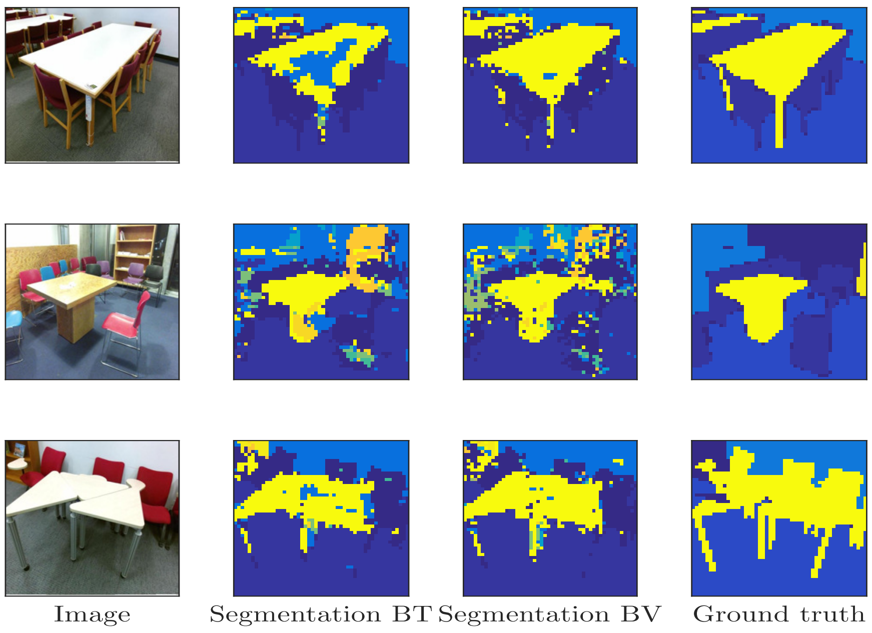


Figure 13: *Example segmentations of obtained network showing somewhat good results. The left column show the original images, resized to dimension* 200×200*. The left centered column show the segmentations generated from running the network with best train (BT) statistics. The right centered column show the segmentations generated from running the network with best validation (BV) statistics. The right column show the corresponding ground truths to the images. Looking at the first row, one sees significantly better accuracy in the network with better validation statistics. This despite the fact that the best trained network has run for about four times as long. Looking at the second row, the appearance of orange in both segmentations indicate the presence of object "shelves" which can clearly be seen in the original image. The ground truth however does not contain any shelves. The networks do still deduce however, based from learnt information and labels from other images, that the shelves are there. Looking at the last row, another image is segmented more accurately than the ground truth. A conversion error, of unknown sort, labels the all furniture to being tables. Once again, the network still classify the chairs to correctly being chairs despite corresponding ground truth.*

Somewhat good segmentations are found in Figure 13, showing qualitatively the differences between net$_{BV}$:s and net$_{BT}$:s results. These segmentations were chosen based off of having high image-wise Jaccard indices. Note that all images consist mainly of chairs, tables, floor and tables which are the top classes measured in Figure 9 and Figure 10. Also note that the segmentations on the second and third row actually overperform relative to ground truth, as ground truth was faulty for these images.

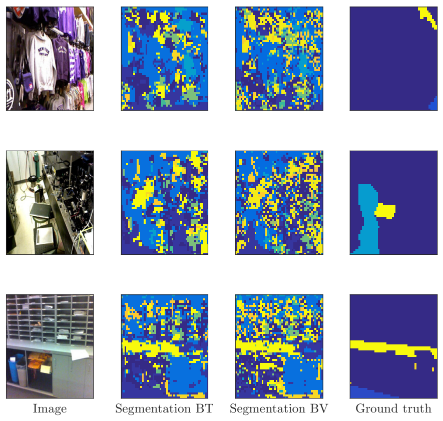


Figure 14: *Example segmentations of obtained network showing, subjectively assessed, diffult images that have very little object information in the ground truth segmentation. For example, object "clothes" is not included in this study's list of object which means that those pixels are given a label classification of 0 in ground truth. The left column show the original images, resized to dimension 200 × 200. The left centered column show the segmentations generated from running the network with best train (BT) statistics. The right centered column show the segmentations generated from running the network with best validation (BV) statistics. The right column show the corresponding ground truths to the images.*

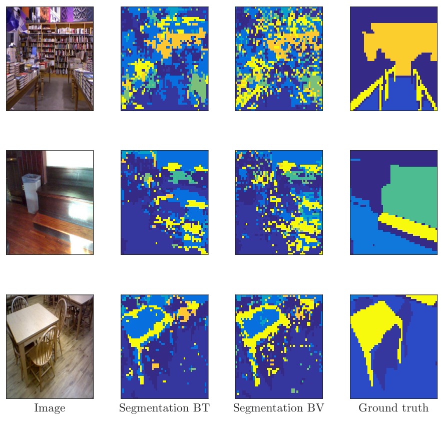


Figure 15: *Example segmentations of obtained network showing, subjectively assessed, difficult images. The left column show the original images, resized to dimension 200 × 200. The left centered column show the segmentations generated from running the network with best train (BT) statistics. The right centered column show the segmentations generated from running the network with best validation (BV) statistics. The right column show the corresponding ground truths to the images.*

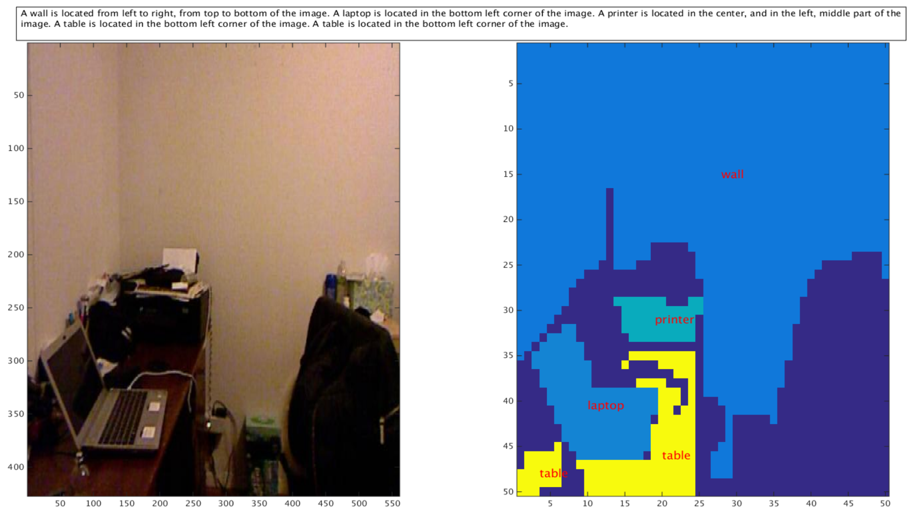
Figure 14 shows network segmentations where the ground truth was correct, but consisting mostly of background. As seen across all segmentations, for both net$_{BT}$ and net$_{BV}$, features in the background are instead interpreted to being known objects. Wheras segmentations on the first and second row show incredibly poor performance in all measures, the third row segmentations do actually segment the known parts of the image quite well.

Network segmentations of, subjectively assessed, difficult images but with sufficient ground truths are displayed in Figure 15. The same tendency of incorrectly classifying background to being known objects is shown here. The segmentations in Figure 15 could however be deemed to show more spatial coherence than in Figure 14. Even though the individual pixel classifications are inaccurate, the general spatial features of the image are somewhat preserved.

Looking at Figure 15:s first row, the difficulty of images with a lot of visual clutter, as well as how one ought to shape ground truths in these environments is highlighted. The network obviously fail to distinguish object instances in the image. The second row show, subjectively spoken, quite an unusual sofa (not bench, as one might expect) in a weirdly lit wooden room. Even though the ground truth perfectly labels the image, the network clearly does not learn to segment the image's objects well. The last row show an extremely color-wise homogeneous setting, with the network for example being unable to distinguish between the wooden table centre belonging to the table and not being a wall. Some chair edges are interpreted as shelves, as indicated by the orange pixels in this instance.

4.2 Spatial localisation of objects

When running the spatial location-algorithm for a couple of different input images, example of results obtained are shown in Figure 16, 17 and 18. The program prints what kind of different items that exist in the image and their corresponding localisation. It also produces red text labels for each object taken into consideration by the algorithm and displays these on the specific object in the segmentation. The spatial localisation is correct as long as the input to the algorithm is legitimate.



A wall is located from left to right, from top to bottom of the image. A laptop is located in the bottom left corner of the image. A printer is located in the center, and in the left, middle part of the image. A table is located in the bottom left corner of the image. A table is located in the bottom left corner of the image.

Figure 16: *This figure shows the results of the algorithm being implemented on a ground truth matrix and its corresponding image. The spatial localisation algorithm produces the text in the top of the figure as well as labels the objects in the ground truth.*

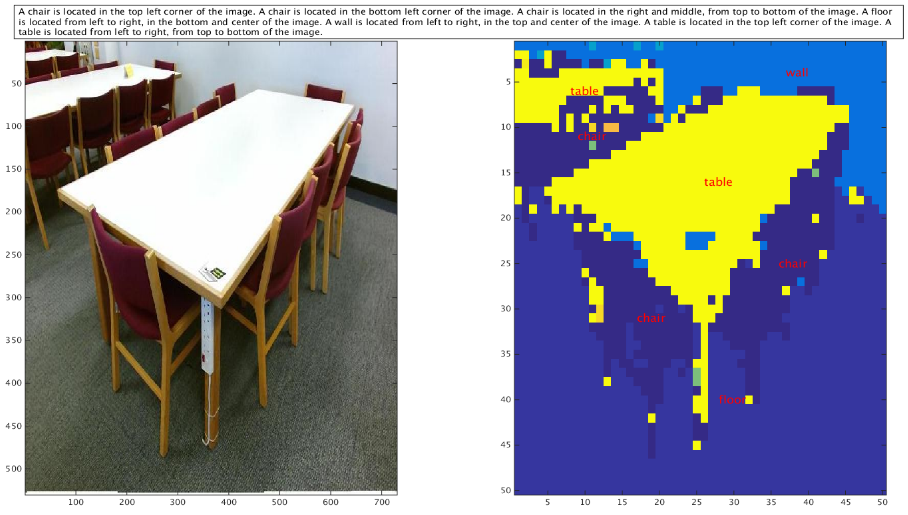

A chair is located in the top left corner of the image. A chair is located in the bottom left corner of the image. A chair is located in the right and middle, from top to bottom of the image. A floor is located from left to right, in the bottom and center of the image. A wall is located from left to right, in the top and center of the image. A table is located in the top left corner of the image. A table is located from left to right, from top to bottom of the image.

Figure 17: *This figure shows the results of the algorithm being run on a good segmentation produced by the CNN. The original image is shown to the left and the corresponding segmentation on the right. The spatial localisation algorithm produces the text in the top of the figure as well as labels the objects in the segmentation.*

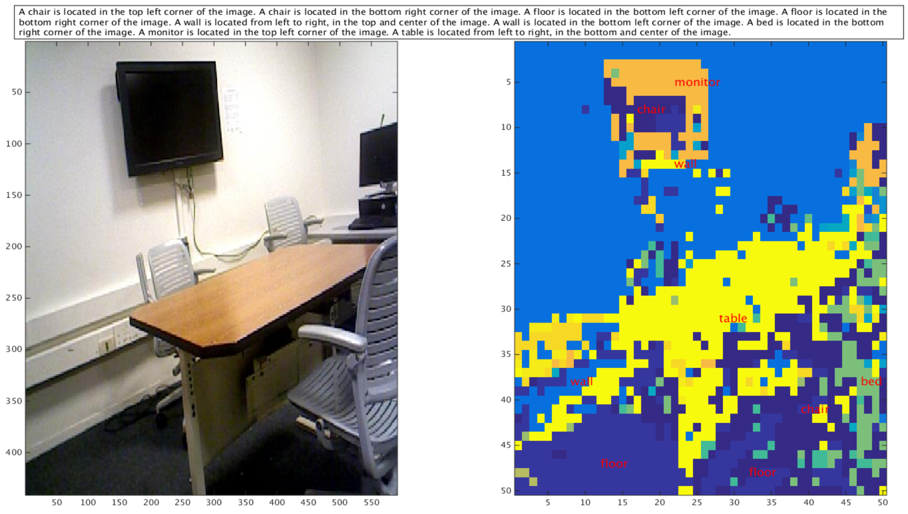

A chair is located in the top left corner of the image. A chair is located in the bottom right corner of the image. A floor is located in the bottom left corner of the image. A floor is located in the bottom right corner of the image. A wall is located from left to right, in the top and center of the image. A wall is located in the bottom left corner of the image. A bed is located in the bottom right corner of the image. A monitor is located in the top left corner of the image. A table is located from left to right, in the bottom and center of the image.

Figure 18: *This figure shows the results of the algorithm being tested with a bad segmentation produced by the CNN. The original image is shown to the left and the corresponding segmentation on the right. The spatial localisation algorithm produces the text in the top of the figure as well as labels the objects in the segmentation.*

5 Discussion

The results obtained and presented in this study does not satisfy the project's initial goal. A functional, but underwhelming, convolutional neural network for semantic segmentation was obtained with the general intended methodology. Firstly the experiments for how the final network structure was obtained will be reviewed and then the final network structure, the architecture of the CNN, will be discussed. Subsequently the CNN's obtained results are discussed. Some alterations were made to the procedure of acquiring a functional data set for learning that, for example, infringed upon the concept of ground truth. That will be explained and elaborated upon.

Result measures and their respective implication will be reviewed. Furthermore, analysis of how one might proceed for greater success in similar studies is performed.

5.1 Experiments

The experiments conducted, described in section 3.3, give an overview of the result one can expect from a network. The used method does not however give any information on whether or not the network design will actually work when trained on a larger data set. A network with few filters in the convolutional layers might excel when evaluated with this method, due to all filters being able to learn necessary features in a specific image. With far more images introduced, and far more features to learn, the few number of filters might then prove insufficient in learning necessary features. Similarly, a net with many filters in the convolutional layers might excel when evaluated in this process, but might fail in practice due to phenomena like overfitting. A more complex network is sometimes necessary for handling more complex data. But a complex network might also over complicate and overly analyze data. For example, let's say the first convolutional layer of a network has three filters and the input image has merely three interesting features. Then the network is well suited for learning that image. But in the case where a convolutional layer with ten filters tries to learn features in the same image, seven of the filters will attempt to analyze other features in the image that are not necessarily relevant in characterizing the object. Thus, the more complex network will overfit the data and not generalize well into practical application.

One can conclude that whilst the method gave information and insight into the training process, large-scale experiments should have been conducted earlier on in the project. The small scale experiments did swiftly conclude that the **Deconv_128** was poor in design. The network design **ConvRedTo4th** that was obtained after the small scale experiments did however have a lot of filters in each convolutional layer, as seen in Table 3. Compared to other segmentation networks, say FCN [6], especially the first convolutional layer had far too many filters. This translates into the network having a lot of parameters. With that many parameters, training took several days to conclude that the network overfit the data. Two attempts of down sizing the network design were done, both reducing the number of convolutional filters in the convolutional layers. One attempt also reduced the field size 5×5 of a later convolutional layer to be 3×3. Both attempts led to networks that overfitted as well. If more time had been dedicated toward large scale training the method drop out, as described in section 2.7.3, could for example have been utilized for potentially reducing overfitting.

5.2 Final network structure

The result of the training in form of loss will be discussed with focus on the network structure, design choices that where made and what possible improvements that could have been implemented.

The obtained loss of the final network was relatively high, around an objective value of 1400 for net$_{BT}$ on training data and around 2200 for net$_{BV}$ on validation data. The graph in Figure 8 is showing clear signs of data overfitting, as the training loss continues to fall and the validation loss is increasing. Training loss exhibit slower convergence towards zero as epochs pass, and looking towards later epochs one might suspect that the network might not improve on the training data much more than current performance. The gradient based learning method seem to reach a local minimum, where parameter updates no longer yield significant improvement in minimizing the loss function. Because of the limited amount of parameters in the network, each update to the parameters during training will eventually just counteract earlier parameter updates, resulting in the stagnation around 1400 for training data.

The loss could possibly be improved by having a more complex network, but it is hard to make a proposal to the exact change in the network structure as several more experiments would be needed. A small increase in the amount of filters could be made overall since the early layers tend to generalize to a standard set of filters. One way to analyze whether they are correct or not could be to comparing them with layers from other professional semantic segmentation networks. If the network structure is compared to the structure from Long et al.[6], looking at the amount of parameters, the difference is significant. In the proposed structure (ConvRedTo4thSize) there are 2,4 Million parameters (see Table 2) and in the network from Long et al. there are roughly 55 times as many parameters. By further studying the differences among the distribution of the filters, to increase the amount of filters in the later layers can seem to be a good choice. Although the comparison fails to be completely fair due to the design choice of not having 1×1 layers in the proposed network.

Besides changing the network structure, data augmentation could have been used to increase the amount of training data and thereby prevent overfitting. To get a measure of how much one might expect this to improve the network, whole-image classifications net can show up to a 10% increase in performance according to Hays [19]. But because the classification problem at hand is semantic segmentation and not whole-image classification as in the example, the actual increase in performance might be lower. Due to time constraints the implemented data augmentation code was never used in the final training.

Another countermeasure that could have been tried is dropout. The reason for not using this was time constraints in combination with the utilization of batch normalization. Because of batch normalization's generalizing properties, which lowers the need for dropout, but also allows for higher learning rates, it was favored over dropout. Even if dropout would have been used, training time would have increased due to its characteristics described in section 2.7.3.

In addition, weight decay is also an important hyperparameter affecting the network's tendency to overfit [20]. A standard setting was used, but the parameter could have been evaluated more thoroughly for further improvements.

As a last improvement an ensemble of different networks, utilizing the different proposed changes to hyperparameters, could have been used to generalize the results further. This usually results in a performance increase of a few percents [21]. To conclude, the training could have been made more thoroughly but because of constraints in both computing power and time the choice to implement all possible improvements could not be made.

5.3 Obtained results

The result would clearly have been superior if a well performing pre-trained network were used and applied on the study's data set. The reasons for the project's network inability to reach the desired goal of a mean Jaccard index of $J \geq 0.6$ might depend on a number of different factors including the ground truth of the data set, how the Jaccard index is calculated or the rough downscaling of labels. These aspects are all discussed below.

5.3.1 Ground truth

The data set is very likely to affect the result. In some cases, as show in Figure 14, the ground truth contains a very limited amount of information. One reason is that the original data set in fact have a restricted quantity of information for these particular images. Another reason could be the reformation and manipulation of the data set. The network of this report takes 25 different classes into consideration while the original data set includes over 800 classes. An image containing only one of the 25 chosen classes will still be part of the data set used in the network and all other classes, that aren't of any interest, are set to background. This makes the 'background' spectra even wider than it already is since objects not among the 25 are also classed as background. When looking at Figure 19 one can, for instance, see that class number 1, chairs, occurs in 25,000 occasions in the data set, while background is seen more than three times as often. An example of this is shown in the top right image of Figure 14. The ground truth in this example only includes a shelf but the image is still part of the used data set. The network tries to identify the other objects in the corresponding image as one of the chosen classes. This could lead to a faulty learning process since the ground truth and the segmentation are compared when calculating the loss and it will seem like the network is far off from a correct classification even though that's not necessary the case. Features from objects not chosen, for example a dishwasher, will have ground truth background but the CNN may classify it as an object part of the data set, for example a coffee machine since they have many features in common. The weights favoring features equivalent for coffee machines and dishwashers will then be degraded and thus making it harder for the CNN to detect a coffee machine in the next iteration. Noteworthy is also that some of the chosen classes are pooled, for example 'semi large container', where classes from the original data set have been combined into one wider category. Albeit to a small extent this also widens the plausible features for these objects which in turn reduces the precision.

There is also an impending likelihood that an incorrect alteration of the weights is made when the ground truths are poorly segmented or even lack segmentation. An example of this phenomenon is illustrated in the middle row of Figure 13. Here the ground truth only specifies some objects in the image, the table and the floor, even though it includes objects that are part of the chosen classes, such as the shelf in the background. When the network tries to classify the shelf it does so correctly but since the ground truth is inconclusive the loss will be high and the weights tweaked in the wrong direction. To solve this problem one could skip the images with poor segmentation ground truths even though it's far from certain this would improve the results as a less numerous data set has it's drawbacks as well. Lack of diversity to mention one.

A final note regarding the data set is the impact of human error. As seen in Figure 13, the ground truths there are not actually correct. For the ground truth on the second row, the

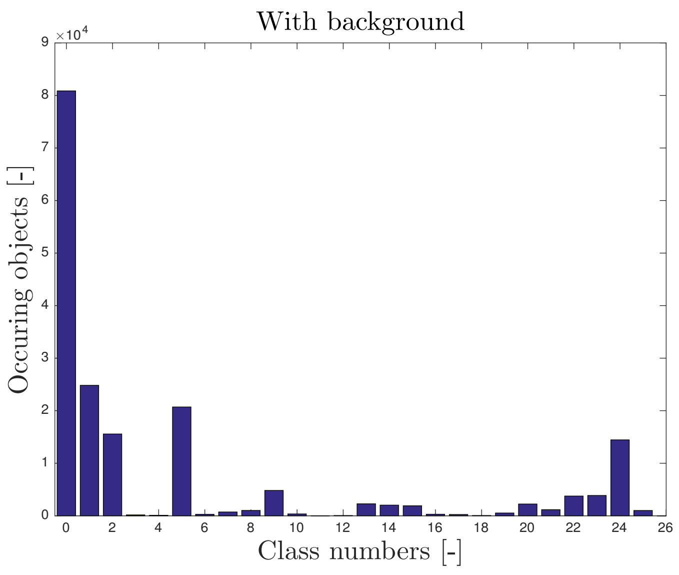
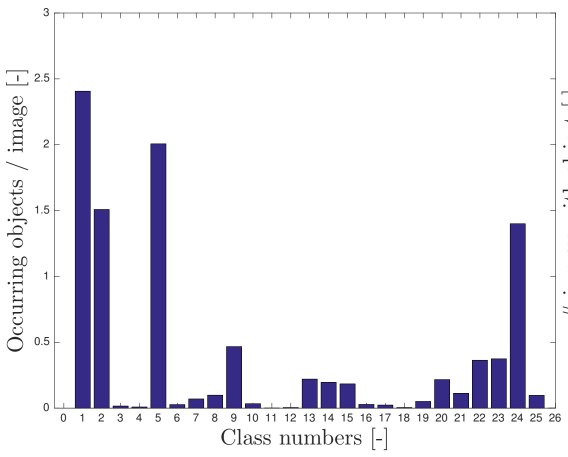


Figure 19: *Number of occasions the data set contains a certain object at least once. So if, for instance, two lamps are seen in one image, that will increase the count by two. The most frequently occurring objects are the background, chairs, floor, walls and tables at respective class numbers 0, 1, 2, 5 and 24. Note the extremely low occurrence of objects like "garbage bin", "refrigerator" and "piano" at respective class numbers 3, 4 and 18.*



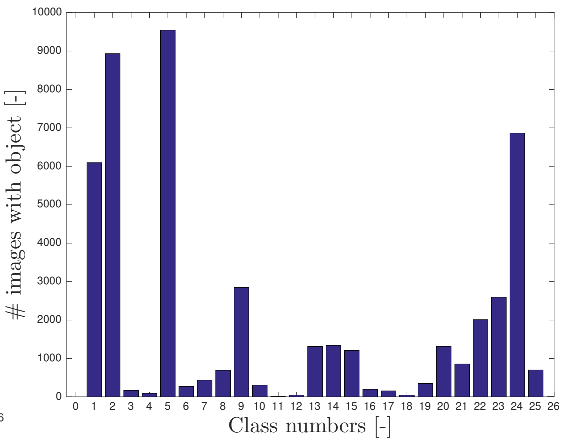


Figure 20: *Number of objects occurring, on average, per image in the data set for each class. Most frequently occurring objects per image are chairs, followed by wall segments at respective class numbers 1 and 5. There are also objects with extremely low occurrence of objects like "garbage bin", "refrigerator" and "piano" at respective class numbers 3, 4 and 18. Note that here background is excluded from this graph.*

Figure 21: *Number of images in the data set that contains a certain object at least once. The most frequently occurring objects are chairs, floor, walls and tables at respective class numbers 1, 2, 5 and 24. There are also objects with extremely low occurrence like "garbage bin", "refrigerator" and "piano" at respective class numbers 3, 4 and 18. Note that background is excluded from this graph.*

function that converts the original data set's numeric label values into this study's class values fail to bring along the shelf. The object category ”shelves” in this study was defined using multiple different class names from the original data set. The converting algorithm takes into account all shelves labeled as ”shelves”, ”bookshelf”, ”bookrack” amongst others, but keyword ”shelf” was not included. Hence, the algorithm labeled that shelf as background. For the ground truth on the third row, the original ground truth was incorrectly labeled. The chairs were labeled as ”desks”, which turned them into tables after conversion due to desk being a subcategory of table in our definition of the object.

5.3.2 Jaccard indices

The variation of the Jaccard indices between different classes vary from 0 to 0.5, as seen in Figure 9 and 10. The classes with high values are chairs (#1), floors (#2) and walls (#5), whereas less common or difficult objects such as computers (#7) and people (#19) have a very low value close to zero. The reason behind this result is probably the characteristic features of chairs, floors and walls, but also the quantitative distribution of objects in the data set, as seen in Figure 20 and Figure 21. The mentioned kind of objects are very common in the original data set. People, on the other hand, are seldom occurred in the data set leading to the CNN's inability to train them. Thus leading to a poor classification and segmentation which in turn generates a low Jaccard index. To avoid this, and as an attempt to reduce the variation in the class-wise Jaccard indices, a data set with a uniform distribution between the different classes should have been used. Alternatively a couple of data sets should have been chosen and mixed.

Let's take a look at the differences between the harsh and non-harsh evaluation in Figure 9 and 10. The Jaccard indices for the non-harsh evaluation are apparently better and do also have a better spread. When doing the harsh evaluation, all pixels that have been incorrectly classified for the analyzed class are taken into consideration. A lot of the segmentations made on images where ground truth contained a incorrect or a small amount of information include many clusters of objects that didn't exist in the ground truth. This is discussed in section 5.3.1, and seen in Figure 14. Obviously, this fact results in a decrease of the Jaccard indices for the harsh evaluation. To prevent this, the study should have used ground truths with a certain minimum amount of information and removed the images with insufficient ground truth data. This would result in a decrease of misleading segmentations and a better value for the harsh evaluation, which technically is the correct way of measuring Jaccardi indices, compared to the modified version of non-harsh evaluation.

Furthermore, by implementing a threshold, and recalculating the Jaccard indices, different conclusions can be drawn. In Figure 11 and 12 the values of the different classes are shown after applying a threshold to remove uncertainties regarding the segmentated pixels. When comparing these graphs with the ones where no threshold was used in Figure 9 and 10, one can see that the values of the classes with a high Jaccard index remain unaltered. The class-wise spread has decreased, where values of the classes with low Jaccard indices further drop off while the 0-classification accuracy has increased. A lot of these classes have thus been classified with a low certainty and the network's two top predictions have had a similar score, leading to them being reclassified as 0. Hence, the network has not fully been able to classify these objects with certainty. This fact further aligns with the assumption made regarding the lack of a uniform distributed data set.

Another deduction that can be made is that the Jaccard index of the network with the best training statistics is higher than the one calculated from the network with the best validation statistics. This is fairly uncommon and could have something to do with the test set. If the test set contains images that are similar to the ones in the training set a better result is obtained since the network would have learned the specific features of these kind of images. Note that the data set was shuffled in an attempt to prevent this from occurring. The training algorithm also re-shuffles the order of each image batch before use.

5.3.3 Obtained segmentations

Obtained segmentations, in this report shown in Figure 13, Figure 14 and Figure 15 are discussed in the following paragraphs.

The somewhat good segmentations, Figure 13, are all of visually clear and un-obscured chairs, tables, floor, wall and shelves. Whilst the general color of some object's is similar to background and other objects, object features are generally quite distinct in these images. This, along with the fact that a lot of, for example chairs, are visually similar in the data set, are probably the two main factors that merits good results. Regions that show very few features, and in monotonous colors, are quite often seen to be difficult to predict in these images. For example, looking at the first row and net$_{BT}$:s segmentation the table center is interpreted as a wall.

Another key point to mention concerning Figure 13 is that the segmentations on rows two and three overperform relative to the faulty corresponding ground truths. Whereas a backward pass of the network using those faulty ground truths would drive the network into qualifying for example the right chair on row three incorrectly, there are enough correct ground truths in the training set containing chairs to ensure that chairs are correctly predicted for most pixels.

The segmentations obtained from images with ground truths that contained mostly background, Figure 14, all show too great of a tendency to classify background into a certain object category. Speculatively, this might be because there are a lot of images that have corresponding ground truths consisting of predominantly background. The various backgrounds, ranging from clothes to documents, have a lot of varying features. This probably leads to the classification of background being unable to converge upon common background features. This leads to that the pixel prediction for category background will always be low. This speculative argument is supported by the 0-column in Figure 9 and Figure 10 having a very low $J_{clAv,l}$ value.

Images with sufficient corresponding ground truths for the study's defined objects are shown in Figure 15. The images are still classified poorly, for speculatively different reasons. The first row's image show a book store with dense information contents, for example with plentiful of edge features and varying colors. Where the corresponding segmentations do display similar spatial semblance to ground truths, pixel-wise predictions show poor accuracy. In the case of the book store, one might assume it could be because of conflicting visual impressions. Specifically, where books in the book shelves are given ground truth labels as part of the category "shelves" the book lying on tables are labeled as background. Whereas this might be unavoidable when specifying ground truth due to bookshelves naturally containing books, segmentation networks can have understandable difficulty in distinguishing the difference.

The second row's image in Figure 15 show a sofa, fooling at least the currently writing author. Where said author views a sofa as being visually quite different, this image exemplifies that segmentation networks also might have difficulty distinguishing between visually very different object instances within the same object category. Alternatively, it exemplifies that ground truths are also subject to human perception where different person's might interpret an object differently.

Figure 15:s third row show an extremely homogeneous setting, where one might have difficulty assessing a local region of the image without full image context. For example, seeing the difference between just the table's center and a part of a wooden wall or floor would prove difficult even for most humans. The same could be said for the region around the chair legs in the back of the image, that the net$_{BT}$ classify as shelves.

In the instance of a homogeneous setting, having convolutional filters with large spatial extent (say a field size of 7×7) does seem preferable due to the larger region that is utilized in classifying each pixel. More spatial context in each pixel classification seems likely to improve pixel predictions in settings like this, which makes the use of a wider field size for this and other segmentation networks seem justified.

5.3.4 Downscaling labels, and the procedure's consequences for the network

When using semantic segmentation to obtain a pixel map of object classes and their spatial locations, one expects a pixel map of the same dimensions as the original image. Learning segmentation with a network that obtains a segmentation with a fourth of the initial spatial extent (1/16 of the image spatial area), as done in this report, is thereby wrong. One can not obtain all relevant spatial information from a downsized segmentation. Also, in the approximate procedure of downscaling ground truths one does not only loose spatial information but there is also a chance for stray pixels ending up not quite where the object is and thus providing a false or semi-false pixel classifications.

The reason why the network was designed for downscaled segmentations was in order to retrieve more and better feature information, whilst not drastically increasing the flow of data in the network. Max pooling retrieves the largest numeric values as the spatial dimension of the data is reduced. The convolutional layers in turn learn to activate upon certain input features and alter their numeric value, so that the downscaled output after pooling preserves the most vital features of the image. These features are then utilized for qualified pixel predictions.

One can learn features without downscaling the data matrices, but since feature extraction demand stacks of often hundreds of convolutional filters to learn activation upon hundreds of features the amount of data that is forward passed will quickly escalate. For example: With an original image of size $200 \times 200 \times 3$, forward pass without max pooling would yield output data of size $200 \times 200 \times 400$ after **ConvRedTo4th**:s fully connected layer (layer #15) compared to the otherwise given size $50 \times 50 \times 400$. Removing max pooling gives 6,4 billion times as many numeric values from in output.

With downscaling necessary as motivated above, the original intent was to implement deconvolutional layers for upscaling. The network would perform segmentation on images that had corresponding ground truths of the same spatial size. The failures in implementing the deconvolutional layers, along with time restraints and deadlines, did however lead to those layers being removed.

If this training had obtained better segmentation statistics than it did, one might have been interested in pursuing an expansion of the network. An interesting approach would have been to, using the pre-trained **ConvRedTo4th** net, try to add a deconvolutional structure for up sampling after the fully connected layer. This could maybe be done by simply adding deconvolutional layers along with ReLU and batch normalization layers in between, and train the added structure whilst keeping the pre-trained model parameters constant (at least initially when training). One would then strive to up sample the $50 \times 50 \times 400$ output to a final prediction of size $200 \times 200 \times 26$, now using ground truths that have size $200 \times 200 \times 1$.

5.4 Spatial location of objects

The results obtained when running the algorithm for spatial localisation depend on the provided input data. A faulty segmentation obviously produce an incorrect localisation of the objects. As seen in Figure 16 the algorithm works well when run with the ground truth as input but when a inaccurate segmentation is provided the resulting output is incorrect. The error can be decreased by altering the thresholds since the incorrect segmentations often consist of many very small clusters, this can be seen in the top row of Figure 15. By setting a threshold high enough, many of these groups of pixels are ignored. Another problem with many clusters are that the algorithm interprets it as numerous different objects and states the location of all of them, as seen in Figure 18.

As mentioned in section 3.4 the algorithm divides the images in 9 regions and place each object in one or a couple of these. With numerous regions the output will be more specific even though the 9 regions are sufficient for a clear overview. The output from the spatial location algorithm can easily be converted into a useful help for blind people via audio description. However, that is left for another project.

5.5 General problems

Not only the more procedural problems mentioned above arose, also plenty of general problems have been dealt with. To begin with, the learning curve is steep as papers in the area assumes that a good amount of knowledge is already gained. Also, the documentation to MatConvNet sometimes is insufficient.

The usage of three different operating systems led to many time consuming takeoffs as there always were problems getting the MatConvNet to run. For instance compilers such as mex needed to be installed and setup. Also when trying to run networks on GPU:s there were countless of errors needed to be fixed before the training could start. Overall much time have been consumed by these kind of matters.

Another general problem was the lack of computing power as it wasn't until in the end of the project occasional access to an above average computer was given. The irregular possibility to check whether the training was still running sometimes led to the training being interrupted for a while.

There also occurred some practical problems when adapting the used data set to the network structure. The SUN data set [9] consists of a couple of smaller data sets, each with its own varying folder structure and content of files. This created confusion when trying to identify the specific files needed for this project since they all varied in name and in which subfolder they were kept. It also provided difficulties when trying to extract the images and corresponding labels of interest. The first attempts lacked large parts of the original data set and adjustments to the script performing the operations had to be done. After several time consuming alterations and attempts the program succeeded in extracting all data of interest. As discussed in section 5.3, a lot of images with a small amount of ground truth information were included in this data set. This could have been prevented by improving the above mentioned script, for example by checking that the minimum amount of background pixels in the ground truth exceeded a certain threshold.

6 Conclusion

This study's main conclusion is that a convolutional neural network can be learnt via deep learning methods to perform semantic segmentation, but that there are many difficulties along the way. Whereas one can conclude from studies like Noh et al. [17], FCN [6] and many others that the techniques work in practice, rigorous theory to support why it works is still non-existing in some cases. Thus, utilizing deep learning techniques one often refers to methods empirically shown to yield good results rather than established theory. Deviating from common practice, without a clear concept and plan in mind, will therefore often result in poor results.

Adhering to common practice is no guarantee for obtaining satisfying results. Where this study abides most standard recommendations and makes no attempt at anything revolutionizing in the field, project goals are not fulfilled. This of course depends on several factors and difficulties encountered.

A second major conclusion is that the data set of choice, and the specific task one strives to accomplish in semantic segmentation, is a key factor for achieving good results. Not only does one need sufficient amounts of images for learning a objects, but one preferably requires a somewhat homogeneous distribution of object classes in a data set so that the network does not adapt to much to a specific class.

Different object classes can have widely different features making them vary in difficulty to classify. Alternatively, different objects can also have very similar features making them hard to distinguish from one another. Aside from these deductions, many other considerations need to be taken into account when choosing and processing a data set for a task.

A third, also major, conclusion is that the architecture of a convolutional neural network is imperative for achieving any desirable results at all. How one decides to for example stack convolutional filters, and with what field size, is what separates sub-par networks from the best. Also, defining layers in suboptimal ways can lead to a variety of issues like over- and underfitting data. How one defines the layers is also dependent on what the task is and what images the data set consists of. Merely replicating and porting a successful network structure for one task to another might not always accomplish expected results.

Not only does one have to consider how the network is defined, but the training process and it's hyper parameters and methods are also important factors for obtaining desirable results. Set learning methods, such as backpropagation with SGD, and hyperparameters like learning rate will decide the convergence of the loss function and thus the performance of the network.

For a continuation of this study, successful implementation of deconvolutional layers would not only be likely to improve performance but would also ensure that ground truths are correctly given the same spatial size as the corresponding images. For this purpose, if implementation of the network is done in MatConvNet [22] then the DagNN-wrapper is recommended for more flexibility in the data flow.

Other practical recommendations is to thoroughly review one's data set before using it for training. Whilst one might not have the patience to manually check ten thousands of images, viewing sample images and making sure that all algorithms for data set processing are correct is highly recommended.

Regarding practial tips for the network definition, layers like Dropout [7] and Batch normalization [8] are referred to in order to prevent phenomena like overfitting. Also, careful review and comparison of one's network definition to other networks available online is recommended. Crucial points to compare are the convolutional layer's parameters, and the order of the layers by, for example, implementing batch normalizations before activations to ensure optimal usage.

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. `http://arxiv.org/pdf/1502.01852v1.pdf`, Februari 2015. Last visited: 4th of February, 2016.

[2] Russakovsky et al. Imagenet large scale visual recognition challenge. `http://link.springer.com/article/10.1007/s11263-015-0816-y`, December 2015. Last visited: 4th of February, 2016.

[3] Microsoft. Microsoft coco captioning challenge. `http://mscoco.cloudapp.net/dataset/#captions-leaderboard`. Last visited: 4th of February, 2016.

[4] Luke Fletcher et al. Computer vision for vehicle monitoring and control. `http://www.robots.ox.ac.uk/~nema/publications/Fletcher01.pdf`, November 2001. Last visited: 13th of May, 2016.

[5] Y. Bengio Y. LeCun, L. Bottou and P. Haffner . Gradient-based learning applied to document recognition. `http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf`. November 1998.

[6] Trevor Darrell Jonathan Long, Evan Shelhamer. Fully convolutional networks for semantic segmentation. `http://www.cs.berkeley.edu/~jonlong/long_shelhamer_fcn.pdf`. Last visited: 12th of May, 2016.

[7] Nitisha Srivasta et al. Dropout: A simple way to prevent neural networks from overfitting. `https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf`. Last visited: 1st of May, 2016.

[8] C. Szegedy. S. Ioffe. Batch normalization: Accelerating deep network training by reducing internal covariate shift. `http://arxiv.org/abs/1502.03167`. Febuary 2015.

[9] S. Lichtenberg S. Song and J. Xiao. Sun rgb-d: A rgb-d scene understanding benchmark suite. `http://rgbd.cs.princeton.edu/`. Last visited: 17th of May, 2016.

[10] P. Kohli R. Fergus N. Silberman, D. Hoiem. Indoor segmentation and support inference from rgbd images. `http://cs.nyu.edu/~silberman/papers/indoor_seg_support.pdf`. In ECCV, 2012. Last visited: 17th of May, 2016.

[11] Y. Jia J. T. Barron M. Fritz K. Saenko A. Janoch, S. Karayev and T. Darrell. A category-level 3-d object dataset: Putting the kinect to work. `https://scalable.mpi-inf.mpg.de/files/2013/04/B3DO_ICCV_2011.pdf`. In ICCV Workshop on Consumer Depth Cameras for Computer Vision, 2011. Last visited: 17th of May, 2016.

[12] A. Owens J. Xiao and A. Torralba. Sun3d: A database of big spaces reconstructed using sfm and object labels. `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6751312`. In ICCV, 2013. Last visited: 17th of May, 2016.

[13] K. Lenc A. Vedaldi. Matconvnet: vl_nnsoftmaxloss, cnn combined softmax and logistic loss. `http://www.vlfeat.org/matconvnet/mfiles/vl_nnsoftmaxloss/`. Last visited: 10th of May, 2016.

[14] K. Lenc A. Vedaldi. Matconvnet: Convolutional neural networks for matlab. `http://www.vlfeat.org/matconvnet/matconvnet-manual.pdf`. Last visited: 10th of May, 2016.

[15] Nvidia Blog. Jen-Hsun Huang. Accelerating ai with gpus: A new computing model. https://blogs.nvidia.com/blog/2016/01/12/accelerating-ai-artificial-intelligence-gpus/. Last visited: 10th of May, 2016.

[16] Nvidia. Table 13. technical specifications per compute capability. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications__technical-specifications-per-compute-capability. Last visited: 16th of May, 2016.

[17] B. Han H. Noh, S. Hong. Learning deconvolution network for semantic segmentation. http://arxiv.org/abs/1505.04366. Last visited: 9th of May, 2016.

[18] G. Ros. Hands-on dl: session 5. http://www.cvc.uab.es/~gros/index.php/hands-on-deep-learning-with-matconvnet/. Last visited: 8th of May, 2016.

[19] James Hays. Introduction to computer vision. http://www.cc.gatech.edu/~hays/compvision/proj6/. Last visited: 15th of May, 2016.

[20] John A. Hertz Anders Krogh. A simple weight decay can improve generalization. http://papers.nips.cc/paper/563-a-simple-weight-decay-can-improve-generalization.pdf. Last visited: 15th of May, 2016.

[21] Li Fei-Fei Andrej Karpathy, Justin Johnson. Model ensembles. http://cs231n.github.io/neural-networks-3/#ensemble. Last visited: 16th of May, 2016.

[22] The MatConvNet Team. Matconvnet: Cnns for matlab. http://www.vlfeat.org/matconvnet/. Last visited: 17th of May, 2016.

Appendix A Final structure

```
function net = netRedTo4thSize( net )

% Init net for segm AxA -> (A/4)x(A/4)
% for example 128 -> 32

% Create net with empty layers - to be filled
net.layers = {} ;

% Initial small number for weights
f = 1/100;

% and an initial bias b = 0
b=1/100; % 0 default, test

% Conv 256->256
net.layers{end+1} = struct('type', 'conv', ...
                           'filters', f*randn(7,7,3,60,'single'), ...% 170
                           'biases', b*ones(1, 60, 'single'), ... % 40
                           'stride', 1, ...
                           'pad', 3) ;
% Batch normalization
net.layers{end+1} = struct('type', 'bnorm',...
                           'filters', f*randn(1,60,'single'),...
                           'biases', b*ones(1,60,'single'));  % G(k)
```

```
% Relu
net.layers{end+1} = struct('type', 'relu') ;

% 256-> (256-5+4)/1 + 1
net.layers{end+1} = struct('type', 'conv', ...
                           'filters', f*randn(5,5,60,120,'single'), ...
                           'biases', b*ones(1, 120, 'single'), ... % 40
                           'stride', 1, ...
                           'pad', 2) ;
% Batch normalization
net.layers{end+1} = struct('type', 'bnorm',...
                           'filters', f*randn(1,120,'single'),...
                           'biases', b*ones(1,120,'single'));  % G(k)
% Relu
net.layers{end+1} = struct('type', 'relu') ;

% Pool max, 256->128
net.layers{end+1} = struct('type', 'pool', ...
                           'method', 'max', ...
                           'pool', [2 2], ...
                           'stride', 2, ...
                           'pad', 0) ;

% Conv 128->128
net.layers{end+1} = struct('type', 'conv', ...
                           'filters', f*randn(5,5,120,200,'single'), ...
                           'biases', b*ones(1, 200, 'single'), ... % 40
                           'stride', 1, ...
                           'pad', 2) ;
% Batch normalization
net.layers{end+1} = struct('type', 'bnorm',...
                           'filters', f*randn(1,200,'single'),...
                           'biases', b*ones(1,200,'single'));  % G(k)
% Relu
net.layers{end+1} = struct('type', 'relu') ;

% Conv 128->128
net.layers{end+1} = struct('type', 'conv', ...
                           'filters', f*randn(5,5,200,300,'single'), ...
                           'biases', b*ones(1, 300, 'single'), ... % 40
                           'stride', 1, ...
                           'pad', 2) ;
% Batch normalization
net.layers{end+1} = struct('type', 'bnorm',...
                           'filters', f*randn(1,300,'single'),...
                           'biases', b*ones(1,300,'single'));  % G(k)
% Relu
net.layers{end+1} = struct('type', 'relu') ;

% Pool max, 128->64
net.layers{end+1} = struct('type', 'pool', ...
                           'method', 'max', ...
                           'pool', [2 2], ...
                           'stride', 2, ...
                           'pad', 0) ;
% FC layer
% 64-> (64-1+0)/1+1
net.layers{end+1} = struct('type', 'conv', ...
                           'filters', f*randn(1,1,300,400,'single'), ...
                           'biases', b*ones(1, 400, 'single'), ... % 40
                           'stride', 1, ...
                           'pad', 0) ;
```

```
% Batch normalization
net.layers{end+1} = struct('type', 'bnorm',...
                           'filters', f*randn(1,400,'single'),...
                           'biases', b*ones(1,400,'single'));  % G(k)
% Relu
net.layers{end+1} = struct('type', 'relu') ;

% Classifier 64x64x400 -> 64x64x2
net.layers{end+1} = struct('type', 'conv', ...
                           'filters', f*randn(1,1,400,26,'single'), ...
                           'biases', b*ones(1, 26, 'single'), ... % 40
                           'stride', 1, ...
                           'pad', 0) ;
% Softmax
net.layers{end+1} = struct('type','softmaxloss'); % loss'); softmax for dagnn

end
```

Appendix B Experiments

Table 3, Table 4, Table 5 and Table 6 summarise the network architectures of the conducted experiments leading up to the final network design.

#	Layer type	Img size	Field size	#Filters	Stride	Pad
1	Conv	200	7	170	1	3
2	Batch norm	200	NA	NA	NA	NA
3	Relu	200	NA	NA	NA	NA
4	Conv	200	5	250	1	2
5	Batch norm	200	NA	NA	NA	NA
6	Relu	200	NA	NA	NA	NA
7	Maxpool	100	2	NA	2	0
8	Conv	100	5	400	1	2
9	Batch norm	100	NA	NA	NA	NA
10	Relu	100	NA	NA	NA	NA
11	Conv	100	5	600	1	2
12	Batchnorm	100	NA	NA	1	NA
13	Relu	100	NA	NA	1	NA
14	Maxpool	50	2	NA	2	0
15	Conv	50	1	400	1	0
16	Batch norm	50	NA	NA	NA	NA
17	Relu	50	NA	NA	NA	NA
18	Conv	50	1	26	1	0
19	Softmax	50	NA	NA	NA	NA

Table 3: *The first, larger, implementation of **ConvRedTo4th**. It has 9.8 million parameters.*

#	Layer type	Img size	Field size	#Filters	Stride	Pad
1	Conv	200	7	60	1	3
2	Batch norm	200	NA	NA	NA	NA
3	Relu	200	NA	NA	NA	NA
4	Conv	200	5	100	1	2
5	Batch norm	200	NA	NA	NA	NA
6	Relu	200	NA	NA	NA	NA
7	Maxpool	100	2	NA	2	0
8	Conv	100	5	120	1	2
9	Batch norm	100	NA	NA	NA	NA
10	Relu	100	NA	NA	NA	NA
11	Conv	100	3	160	1	1
12	Batchnorm	100	NA	NA	1	NA
13	Relu	100	NA	NA	1	NA
14	Maxpool	50	2	NA	2	0
15	Conv	50	1	400	1	0
16	Batch norm	50	NA	NA	NA	NA
17	Relu	50	NA	NA	NA	NA
18	Conv	50	1	26	1	0
19	Softmax	50	NA	NA	NA	NA

Table 4: *The third, smallest, implementation of **ConvRedTo4th**. It has 760 000 parameters. Note that the last convolutional layer has a quadratic field size of 3 instead of 5, differing from the other implementations.*

#	Layer type	Img size	Field size	#Filters	Stride	Pad	crop/upsample
1	Conv	128	7	80	1	3	NA
2	Relu	128	NA	NA	NA	NA	NA
3	Maxpool	64	2	NA	2	0	NA
4	Conv	64	5	120	1	2	NA
5	Relu	64	NA	NA	NA	NA	NA
6	Maxpool	32	2	NA	2	0	NA
7	Conv	32	5	150	1	2	NA
8	Relu	32	NA	NA	NA	NA	NA
9	Batch norm	32	NA	150	NA	NA	NA
10	Maxpool	16	2	NA	2	0	NA
11	Conv	1	16	450	1	0	NA
12	Relu	1	NA	NA	NA	NA	NA
13	Deconv	4	4	450	1	NA	0/2
14	Relu	4	NA	NA	NA	NA	NA
15	Batch norm	4	NA	450	NA	NA	NA
16	Deconv	16	10	200	1	NA	0/2
17	Relu	16	NA	NA	NA	NA	NA
18	Batch norm	16	NA	200	NA	NA	NA
19	Deconv	32	2	200	1	NA	0/2
20	Relu	32	NA	NA	NA	NA	NA
21	Batchnorm	32	NA	200	1	NA	NA
22	Deconv	64	2	150	1	NA	0/2
23	Relu	64	NA	NA	1	NA	NA
24	Batchnorm	64	NA	150	1	NA	NA
18	Conv	64	1	26	1	0	1
19	Softmax	64	NA	NA	NA	NA	0

Table 5: ***Deconv_128***: *An attempt to implement deconvolutional layers into the network architecture. The design has obvious flaws and errors. Implementing this in with straight forward- and backward passes in the SimpleNN-wrapper, the deconvolutional layers will eventually attempt to up sample an image of size* $128 \times 128 \times 3$ *based solely on information preserved in the downsampled* $1 \times 1 \times 450$ *vector. This fact alone makes the network design useless. Another, somewhat minor, flaw is that the batch normalizations are implemented before the relu-activations and not as frequently as recommended in the original batch normalization report [8]. The implementation flaw for deconvolutional layers came from having insufficient knowledge of how deconvolution could be implemented in MatConvNet. Most fundamentally, the flaws came from a lack of understanding in how one has to adapt data flow in the network to ensure that the deconvolutional layers are given sufficient information to make qualified up samples of input data. The implementation of batch normalization also showed a lack of understanding in that one wants to center and normalize data before relu-activations in order to fully utilize the layer's potential. Also note that the up sampled output segmentation is not obtained with the same spatial dimensions as the input image, which means that the concept of ground truth is violated (see section 5.3.4).*

#	Layer type	Img size	Field size	#Filters	Stride	Pad	crop/upsample
1	Conv	128	7	120	1	3	NA
2	Relu	128	NA	NA	NA	NA	NA
3	Maxpool	64	2	NA	2	0	NA
4	Conv	64	5	150	1	2	NA
5	Relu	64	NA	NA	NA	NA	NA
6	Maxpool	32	2	NA	2	0	NA
7	Conv	32	5	400	1	2	NA
8	Relu	32	NA	NA	NA	NA	NA
9	Batch norm	32	NA	400	NA	NA	NA
10	Deconv	70	8	500	1	0	0/2
11	Relu	70	NA	NA	NA	NA	NA
12	Batch norm	70	NA	500	NA	NA	NA
13	Conv	70	1	26	1	2	NA
14	Softmax	70	NA	NA	NA	NA	NA

Table 6: ***Deconv_light***: *An attempt to implement a single deconvolutional layer at the end of an otherwise ordinary convolutional downsampling network. The implementation shows lack of understanding of how the batch normalization works, as elaborated on in Table 5. The implementation of a single deconvolutional layer is also lackluster. Since the layer only uses information gained from the previous layer output, it serves only to up sample it's input similarly to an interpolation filter. Just like in Table 5, the up sampled output is not obtained with the same spatial dimensions as the input image which means that the concept of ground truth is violated here as well.*

Appendix C Chosing Classes Algorithm

```
% Sorts out the classes chosen from the data set SUN
clear all
clc

ImgFound = 0;
ImgSave = 0;

% Path to dataset
GrndPath = '/Users/adamlilja/Downloads/SUNRGBD/';

% Choosen classes
classes = ...
    {'bathtub', 'bed', 'bookshelf', 'box', 'chair', 'counter',...
    'desk', 'door', 'dresser', 'garbage bin', 'lamp', 'monitor',...
    'night stand', 'pillow', 'sink', 'sofa', 'table', 'tv', 'person',...
    'toilet', 'fruit', 'garbage_bin', 'night_stand'};

dirListing0 = sortGetDir(dir(GrndPath)) ;
for o = 1 : length(dirListing0)

TmpPathI = fullfile(GrndPath,dirListing0(o).name) ;
dirListingI = sortGetDir(dir(TmpPathI)) ;
disp(TmpPathI)

for i = 1 : length(dirListingI)
    TmpPathII = fullfile(TmpPathI,dirListingI(i).name) ;
    dirListingII = sortGetDir(dir(TmpPathII)) ;
    for ii = 1 : length(dirListingII)
        TmpPathIII = fullfile(TmpPathII,dirListingII(ii).name) ;
        dirListingIII = sortGetDir(dir(TmpPathIII)) ;
        enough = DeepEnough(dirListingIII) ;
        if enough == 1
                ImgFound = ImgFound + 1;
                ImgSave = MaybeSave(TmpPathIII, classes, ImgSave) ;
        else
        for iii = 1 : length(dirListingIII)
            TmpPathIV = fullfile(TmpPathIII,dirListingIII(iii).name) ;
            dirListingIV = sortGetDir(dir(TmpPathIV)) ;
            enough = DeepEnough(dirListingIV) ;
            if enough == 1
                ImgFound = ImgFound + 1;
                ImgSave = MaybeSave(TmpPathIV, classes, ImgSave) ;
            else
                for iv = 1 : length(dirListingIV)
                    TmpPathV = fullfile(TmpPathIV,dirListingIV(iv).name) ;
                    dirListingV = sortGetDir(dir(TmpPathV)) ;
                    enough = DeepEnough(dirListingV) ;
                    if enough == 1
                        ImgFound = ImgFound + 1;
                        ImgSave = MaybeSave(TmpPathV, classes, ImgSave) ;
                    else
                            for v = 1 : length(dirListingV)
                                TmpPathVI = fullfile(TmpPathV,dirListingV(v).name) ;
                                dirListingVI = sortGetDir(dir(TmpPathVI)) ;
                                enough = DeepEnough(dirListingVI) ;
                                if enough == 1
                                    ImgFound = ImgFound + 1 ;
                                    ImgSave = MaybeSave(TmpPathVI, classes, ImgSave) ;
                                else
                                        for vi = 1 : length(dirListingVI)
                                             TmpPathXtionVII = fullfile(TmpPathVI,dirListingVI(vi).name) ;
                                             dirListingVII = sortGetDir(dir(TmpPathXtionVII)) ;

                                             enough = DeepEnough(dirListingVII);
                                             if enough == 1
                                                ImgFound = ImgFound + 1 ;
                                                ImgSave = MaybeSave(TmpPathVI, classes, ImgSave);
                                             else
                                                disp('Failure')
                                             end
                                        end
                                end
                            end
                     end
                end
            end
        end
        end
    end
end
end


function dirListing = sortGetDir(dirListing)

% Get a directory listing with only folders
% and not '.', '..'

for i=1:size(dirListing,1)
    dirIndex = 1;

    if strcmp(dirListing(dirIndex).name,'.') ...
            || strcmp(dirListing(dirIndex).name,'..') ...
            || strcmp(dirListing(dirIndex).name,'.DS_Store')
        dirListing(dirIndex) = [];
    else
        dirIndex = dirIndex + 1;
    end
end

% And return dirListing without removed elements
%dirListing(~cellfun('isempty',dirListing))

end


function enough = DeepEnough(dirListing)
% Check if the folder is sufficently deep down among the subfolders by
% checking if there is a folder named 'image'. If there is a folder named
% 'image' the dirListing is deep enough.

for i = 1 : length(dirListing)
    check = strcmp(dirListing(i).name,'image');
    if check == 1
        enough = 1;
        break;
    else
        enough = 0;
    end
```

```
end


function ImgSave = MaybeSave(TmpPath, classes, ImgSave)

% Path to SortDataset.m
Path = '/Users/adamlilja/Documents/MATLAB/Kandidatarbete';
NewPath = fullfile(Path,'Dataset');

flag = 0;
JsonWorks = 1;

% Load the segmentation
load(fullfile(TmpPath,'seg.mat'))

% Load the image
ImgPath = fullfile(TmpPath,'image');
D = dir(ImgPath);
ImgName = D(end).name;
img = imread(fullfile(ImgPath,ImgName));

if JsonWorks == 1
% Unique values in seglabel matrix
UniVal = unique(seglabel);
UniVal = nonzeros(UniVal);

for i = 1: length(classes)
    for ii = 1: length(UniVal)

        x = strmatch(classes{i}, names{UniVal(ii)}, 'exact');

        if x == 1
               ImgSave = ImgSave + 1;
               cd(NewPath)
               ImgFoundPath = fullfile(sprintf('img_%05d',ImgSave));
               mkdir(ImgFoundPath);
               cd(ImgFoundPath);
               save('image.mat','img');
               save('segmentation.mat','seglabel','names');
               cd(Path)
               flag = 1;
        end
                if flag == 1; break;end
    end
                if flag == 1; break; end
end

end
```

Appendix D Reformat Label Values Algorithm

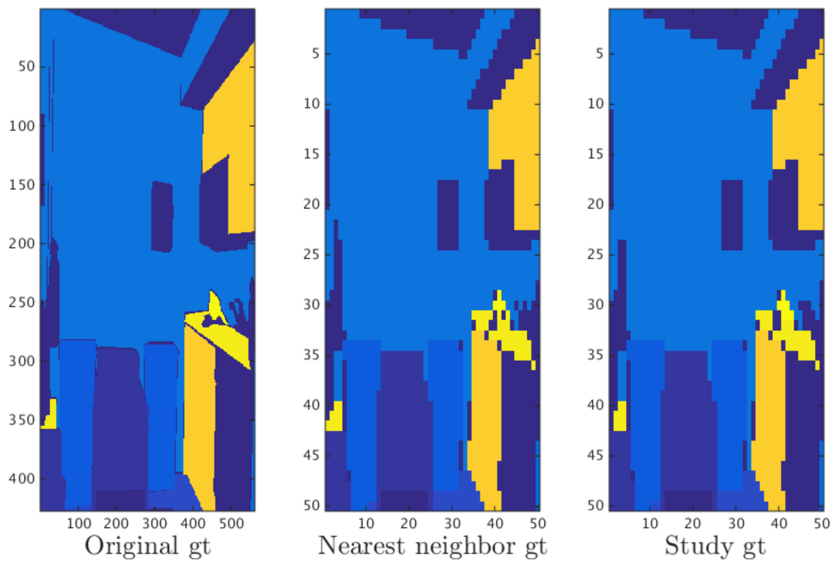


Figure 22: *Ground truths compared after downscaling. To the left is the original ground truth image. In the middle is a downscaled ground truth using Matlab's "nearest neighbor" method. To the right is a downscaled ground truth using the custom method described in section 3.1.2 and with code specified here in Appendix D.*

```
clc; clear all;

% Script to relabel all seglabels in dataset with fixed class values,
% as: classVal = getClassValue('classes'). After this script obtains the
% occuring classes in a segmentation. All undesired classes have values = 0.

imdb = load('imdb.mat');
imdb = imdb.imdb;

for i=1:size(imdb.images,2)
    i
    % Go to label folder
    labelPath = strcat(pwd,'/Dataset/',...
        imdb.images( i ).dataPath,'/' );

    % Get segmentation
    segm = load(strcat(labelPath,'segmentation.mat'));
    % Get seg label number corresponding to a class name
    names = segm.names;
    % and get segmented image
    segm = segm.seglabel;

    % Get occuring classes in segmentation
    occuringLabelNbrs = unique(segm);
    occuringLabelNbrs = nonzeros(occuringLabelNbrs);
    % and how many they are
    nbrClassesInImg = max(size( occuringLabelNbrs ));

    % Also, determine the class values for the currrent segmentation,
    % according to our numbering
    prevClassValues = zeros(1,nbrClassesInImg);
    % and save corresponding class names
    classes = cell(1,nbrClassesInImg);

    % for all occuring classes, fill in prevClassValues and classes
    for j=1:nbrClassesInImg
        classes{j} = names{ occuringLabelNbrs(j) };
        prevClassValues(j) = occuringLabelNbrs(j);
    end

    % " Tell, don't ask" - get desired class values for our classes
    classVals = getClassValues(classes);

    % Initialize a 4D segmentation, with depth 4th dim for each occuring
    % class in current labeling
    tmpSegm = zeros( size(segm,1), size(segm,2), 1, nbrClassesInImg );

    % And loop through to get relabeled matrices of each labeled class
    for j=1:nbrClassesInImg%nbrOfClassLabels % labels 0,1,..
        % Get the prevClassVals(j)-labeled matrix of current segmentation
        % in 1:s
        tmpSegm(:,:,:,j) = ( segm == prevClassValues(j) );
        % and multiply by new label
        tmpSegm(:,:,:,j) = tmpSegm(:,:,:,j)*classVals(j);
    end

    % Finally, sum up 4th dimension into 2d segmentation (singleton 3rd dim)
    segmentation = single( sum(tmpSegm,4) );

    % Finally, save new segmentation to file
    clearvars -except segmentation imdb labelPath
    segmFile = fullfile(labelPath, 'segm.mat');
    save(segmFile, 'segmentation')
    clearvars -except imdb
end

%%
labelPath = strcat(pwd,'\Dataset\',...
        imdb.images(4).dataPath,'\' );

% Visualize reformated labels
subplot(1,3,1)
newSegm = load(strcat(labelPath,'segm.mat'));
newSegm = newSegm.segmentation;
imagesc(uint8(newSegm))
hold on
% Get segmentation
segmOrig = load(strcat(labelPath,'\segmentation.mat'));
segmOrig = segmOrig.seglabel;
subplot(1,3,2)
imagesc(segmOrig)

% and get original image
origImg = load(strcat(labelPath,'\image.mat'));
origImg = origImg.img;
subplot(1,3,3)
imagesc(origImg)
```

Appendix E Spatial localisation algorithm

```
function spatialloc( seg , img)
% This function takes an image and its corresponding segmentation matrix
% as input and produces a written output where the different objects, that
% have been identified in the image, are located. It also creates a label
% for each object and prints it in the segmentation. Only objects greater
% than or equal to 1% of the image's spatial extent are considered. This
% can be adjusted in the vector'thresholds'.


dim = size(seg);

% Display segmentation
f = figure;

subplot(1,2,1)
imagesc(img);

subplot(1,2,2)
imagesc(seg);

% All our classes
classNames = {'chair' 'floor' 'garbage bin' 'refrigerator' 'wall' ...
    'laptop' 'computer' 'keyboard' 'window' 'printer' 'fax machine' ...
    'coffee machine' 'sofa' 'lamp' 'bed' 'bench' 'stairs' 'piano' ...
    'person' 'monitor' 'shelf' 'semi large container' 'door' 'table' ...
    'small container' };

% Minimum pixel area for all classes in parts per hundred
thresholds = [0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 ...
    0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01];

% Dividing the image into 9 regions
regions = length(seg)/3;

% Find all classes in segmentation matrix
classes = unique(seg);

% If the class background (0) is in the image, remove it from the list
if (any(classes == 0))
    classes = classes(2:end);
end

% Output string
descr = '';

% Loop through all classes except class 0
for i=1:length(classes)
    % Save the indices of one class in a new matrix
    tmp = seg==classes(i);
    % Finds the connected components (blobs)
    blobs = bwconncomp(tmp);
    % Finds the area of the blobs
    area = regionprops(blobs,'area');
    % Checks if the area of the blobs is larger than the threshold
    tmp2 = find([area.Area] > (thresholds(classes(i))*dim(1)*dim(2)));
    % Stores the blobs that meet the conditions
    tmp3 = ismember(labelmatrix(blobs),tmp2);
```

```
% Gets the coordinates of all the elements in the blobs
coord = regionprops(tmp3,'pixellist');
%center = regionprops(tmp3,'centroid');

%Loops through all blobs
for j=1:length(coord)
    %Checks how many pixels of the object are in the left side of the
    %image...
    left = (coord(j).PixelList(:,1) < regions);
    %... and the right ...
    right = (coord(j).PixelList(:,1) > 2*regions);
    %... and the center
    xCenter = ((coord(j).PixelList(:,1) > regions) & ...
        (coord(j).PixelList(:,1) < (2*regions)));
    % Checks for nonzero elements, 1 if there are, 0 if there aren't
    xPosss = [any(left) any(xCenter) any(right)];
    % Specifies in how many regions the object is, x-wise
    xRegion = length(find(xPosss==1));

    % Does the same but in the y-dim.
    top = (coord(j).PixelList(:,2) < regions);
    bottom = (coord(j).PixelList(:,2) > 2*regions);
    yCenter = ((coord(j).PixelList(:,2) > regions) & ...
        (coord(j).PixelList(:,2) < 2*regions));
    yPosss = [any(top) any(yCenter) any(bottom)];
    yRegion = length(find(yPosss==1));

    % The following block of code checks over how many and which
    % regions the object spans.

    % Enters if the obj is in only one region x-wise
    if (xRegion == 1)
        if (left)
            xPos = 'on the left hand side, ';
        elseif (right)
            xPos = 'on the right hand side, ';
        elseif (xCenter)
            xPos = 'in the middle, ';
        end
    end
    % Enters if the obj is in only one region y-wise
    if(yRegion == 1)
        if (top)
            yPos = 'in the top';
        elseif (bottom)
            yPos = 'in the bottom';
        elseif (yCenter)
            yPos = 'in the center';
        end
    end

    % Checks if the obj is in a corner
    if (left & top)
            xPos=''; yPos='in the top left corner';
    elseif (left & bottom)
            xPos=''; yPos='in the bottom left corner';
    elseif ((right) & bottom)
            xPos=''; yPos='in the bottom right corner';
    elseif ((right) & top)
            xPos=''; yPos='in the top right corner';
    end
    % Enters if the object is in 2,3 or 4 regions.
```

```
if((xRegion==1 && yRegion==2) || (xRegion==2 && yRegion==2) || ...
        (xRegion==2 && yRegion==1))
    if (any(left) && any(top))
        xPos=''; yPos='in the top left corner';
    elseif (any(left) && any(bottom))
        xPos=''; yPos='in the bottom left corner';
    elseif (any(right) && any(bottom))
        xPos=''; yPos='in the bottom right corner';
    elseif (any(right) && any(top))
        xPos=''; yPos='in the top right corner';
    elseif (xRegion==1 && all(xCenter))
        xPos='';
        if (any(top))
            yPos='in the middle, and in the top, center part';
        elseif (any(bottom))
            yPos='in the middle, and in the bottom, center part';
        end
    elseif (yRegion==1 && all(yCenter))
        xPos='';
        if (any(left))
            yPos='in the center, and in the left, middle part';
        elseif (any(right))
            yPos='in the center, and in the right, middle part';
        end
    end
end
% Enters if the object is from left to right in the image
if (xRegion == 3)
    xPos = 'from left to right, ';
    if(yRegion == 2)
        if (any(bottom))
            yPos = 'in the bottom and center';
        elseif (any(top))
            yPos = 'in the top and center';
        end
    end
end
% Enters if the object is from top to bottom in the image
if (yRegion == 3)
    yPos = 'from top to bottom';
    if(xRegion == 2)
        if (any(left))
            xPos = 'in the left and middle, ';
        elseif (any(right))
            xPos = 'in the right and middle, ';
        end
    end
end

% Write the specific class on the specific object in the
% segmented image.
subplot(1,2,2)
text(median(coord(j).PixelList(:,1)),...
    median(coord(j).PixelList(:,2)),classNames{classes(i)}...
    ,'FontSize', 14, 'Color', [1 0 0])

% If the object is only in the middle region
if (strcmp(xPos,'in the middle, ') && strcmp(yPos,'in the center'))
    descr = horzcat(descr,' ',horzcat('The ',...
        classNames{classes(i)},' is located',' ',yPos));
% Otherwise
else
```

```
            descr = horzcat(descr,' ', horzcat('A ',...
                classNames{classes(i)},' is located',' ',xPos,yPos,...
                ' of the image.'));
        end
    end
end

% Dimension for textbox
dim = [.12 .93 .8 .06];
% Output
annotation('textbox',dim,'String',descr,'FontSize',11);
% Set window size
set(f, 'Position', [0, 0, 1700, 900])
```