FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatik

The Needle Detection and Reconstruction based on a Convolutional Neural Network (CNN)

Simon Schlegl

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatik

The Needle Detection and Reconstruction based on a Convolutional Neural Network (CNN)

Die Nadel Erkennung und Rekonstruktion basierend auf einem Convolutional Neural Network (CNN)

Author:	Simon Schlegl
Supervisor:	Prof. Dr.-Ing. habil. Alois Knoll
Advisor:	Zhou, Mingchuan, M.Eng.
Date:	March 14, 2017

I assure the single handed composition of this bachlors's thesis only supported by declared resources.

Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Munich, March 14, 2017

München, den 14. März 2017 Simon Schlegl

Acknowledgments

I thank my advisor Mingchuan Zhou for being always available for any questions I had, and for guiding me through this thesis. Furthermore thanks to Gao Xiang I was able to train the networks on better and faster hardware than I could ever buy. Finally I thank Prof. Dr. Knoll for making this thesis possible.

Abstract

Assistive algorithms for proximity estimation used by surgeons during ophthalmic surgery have become a research interest recently. In order to obtain images, Optical Coherence Tomography (OCT) is used, which returns a 3D image of the eye. Proximity estimation then tries to find the distance between the retina and the surgeont's needle in the given image. This, however, is not as easy as the image of the needle can be distorted. Thus designing a conventional algorithm is quite complicated. In order to tackle these problems, needle detection using Convolutional Neural Networks is attempted in this thesis as well as Reconstruction of the needle afterwards. Needle Segmentaion using Convolutional Neural Networks shows promising results and surpasses the conventional algorithm in accuracy, but suffers from a slower performance.

Contents

1 Introduction

Ophthalmic posterior segment surgery is a very challenging task for a human surgeon to perform with only microscopic feedback. To reduce this problem, assistive algorithms for proximity estimation in microscopic images have become a field of interest [7].

Obtaining the microscopic images can be done by utilizing Optical Coherence Tomography (OCT), which is a technique to create high resolution 3D images of living biological tissue [9].

The obtained images then must be processed to find the needle in the image slices. Finally the different segments will be used to predict the position of the needle.

Processing the images is usually done with conventional algorithms, but in this thesis image segmentation based on Convolutional Neural Networks is attempted.

1.1 Optical Coherence Tomography

Optical Coherence Tomography (OCT) is a relatively new tissue imaging technique, that works similarily to ultrasound, but replaces sound by near-infrared light [6]. One advantage is, that OCT can be used in vivo, that is, on living human tissue without the need to take samples. This is also an absolute necessity for this problem, as removing the eye for the operation would defeat the whole purpose.

Comparing OCT to other in vivo imaging techniques in terms of resolution (see Figure 1.1) shows that OCT is able to produce higher resolution images than the alternatives.

	OCT	Ultrasound	MRI	Fluroscopy	Angioscopy
Resolution (μm)	1-15	80-120	80-300	100-200	≤ 200

Figure 1.1: Comparing the resolution of different imaging techniques taken from [6]

Moreover, as OCT uses near-infrared light, which is non-ionizing radition, there are no additional radiation risks, unlike for example with High Resolution Computed Tomography [6].

All of this makes OCT a favorable microscopic technique for obtaining real time images while conducting ophthalmic posterior segment surgery.

1.2 Problem Space

OCT imaging produces a 3D data cube of the retina split into many slices. All cubes used in this thesis are of dimension $1024 \times 512 \times 128$ (height by width by depth).

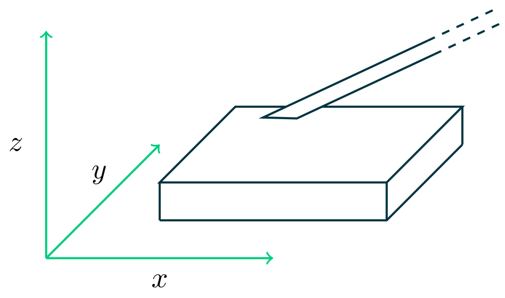

Figure 1.2: data cube dimensions

The 3D data cube is sliced along the zy plane, so there are 128 slices in one dataset (see Figure 1.2).

Looking at slice samples from a real dataset (see Figure 1.3) one can cleary make out an oval shape travelling up, this is the top part of the needle. OCT imaging is unable to penetrate metal, thus only the top part of the needle and a large black shadow below the needle can be seen.

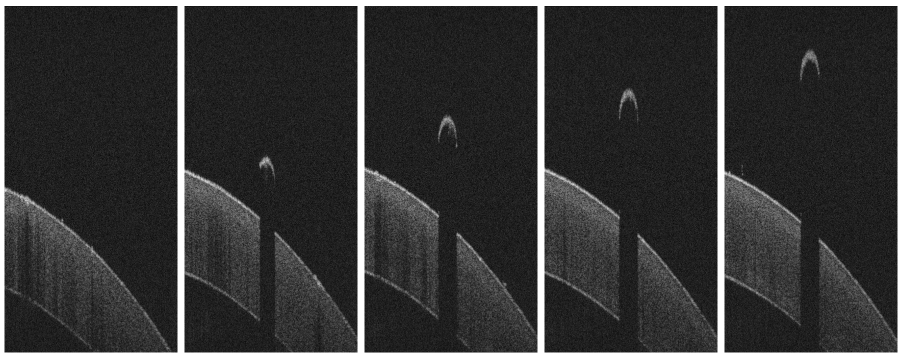

Figure 1.3: slice samples from a data cube

In order to be able to reconstruct the needle from the data cube, at first the position of the needle in every data slice must be found. For this a simple bounding box is both sufficient and already problematic to compute.

After having found all the bounding boxes of all needle slices, assembling all the different parts and predicting the position and orientation of the needle is possible.

1.3 Why CNNs

Detecting the needle is not as simple as detecting a “needle like” shape, as several distortions can occur. Distortions usually even split the needle shape up into two or more shapes so distinguishing which parts belong to the needle and which parts are just noise is necessary (see figure Figure 1.4).

Furthermore after the needle is not in the area of the cube anymore, refractions can occur that result in a needle like shape but upside down (see figure 1.5).

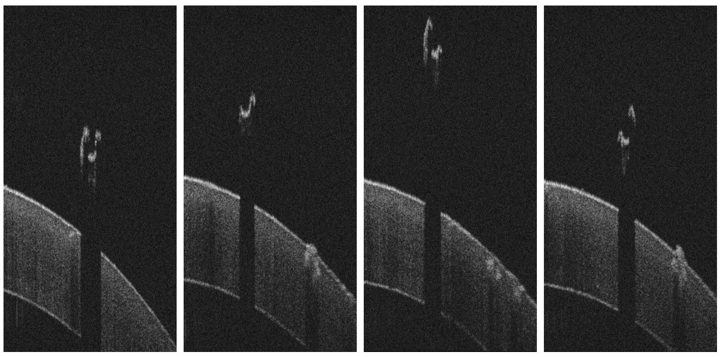
Figure 1.4: image with needle, but distorted

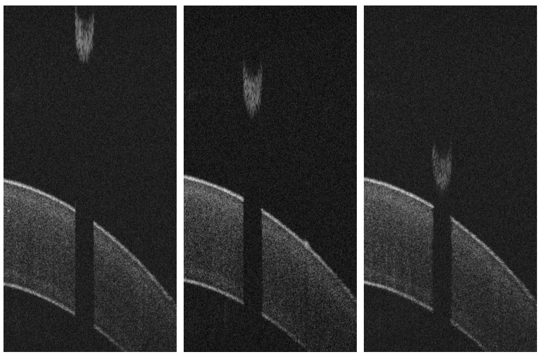
Figure 1.5: image without needle, but with refraction

Convolutional Neural Networks (CNNs) are a novel approach to classify patterns in images [5]. Seeing the difficulties of different needle like shapes it is possible that a CNN is able to produce more accurate results in detecting the needle slices.

For this particular problem of needle detection simple coarse image classification would not suffice. Luckily recent advances in computer vision showed promising results in dense pixel segmentation, that paved the way to more precise classifications of pixels [5].

Furthermore the fact, that the images only have two labels (background and needle) makes it possible to use the semi dense pixel segmentation output from a CNN to obtain pixel perfect images images.

2 Background

2.1 Artificial Neural Networks

Artificial Neural Networks (ANN) are an emerging subfield of Machine Learning, that have been developed by observing how the brain uses interconnected nodes to compute all different kind of pattern recognition tasks [1].

Unlike conventional programming neural networks are "programmed" by training the network with a large dataset of input data and desired output data [1].

There are many different structures for ANNs, but only a feed forward network will be introduced here, as more advanced structures won't be used in this thesis.

A basic feed forward network has a rather easy structure: A number of neurons, that are interconnected with each other, while each connection has an associated weight. In a feed forward network these connections must build a directed acyclic graph, so that a given input can flow in one direction to the output neurons [1]. These neurons are often grouped together into *layers* of neurons. The differentiation of so called *input*, *hidden* and *output* neurons simply implies the position of the neuron in the network. A neuron in the beginning of the network without any ingoing connections, where the input will be supplied, is called an *input neuron* (see neurons x_1, x_2 and x_3 in Figure 2.1). Similarly neurons, that do not have any further outgoing connections, are called *output neurons* (x_5 and x_6). All the remaining neurons are labeled *hidden neurons* (x_4).

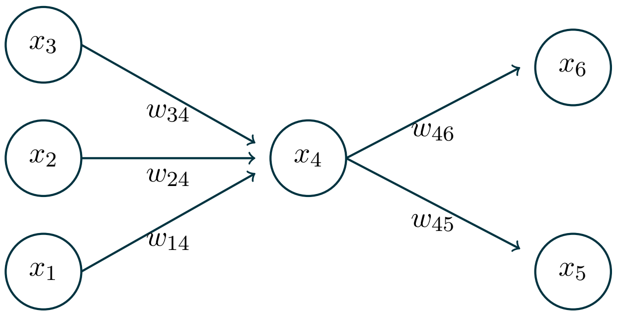

Figure 2.1: An example of a single neuron with three input connections

2.1.1 Forward Pass

A neurons input is computed by adding up all the outputs of the connected neurons weighted by the given weight of the connection, adding another value called *bias* and applying a function to it:

$$output = x_j = f(\sum w_{ij}x_i + b_j)$$

where w_ij are the weights associated with the connection from neuron i to j. This is summed over all the connections between any neuron i, that is connected to this particular neuron j (see Figure 2.1) [1].

To do a full forward pass, one must at first insert the input data in all the input neurons and then compute the output of each following neuron until the output layer is reached.

2.1.2 Backpropagation

Training an ANN is equivalent to modifying the weights connecting the neurons. For this an algorithm called *backpropagation* is used.

The basic idea is to add a cost function (denoted by E) at the end of the network to compute the error of the given input compared to the actual desired result. (This requires that input/output pairs for the given problem are available.) Then the partial derivatives for every weight in the network are calculated to find out how a small change in this weight will affect the error.

To decrease the error then, simply the negative derivative is added to the weight multiplied by a given learn rate η.

A common modification is to also add momentum, which is just weighting in the result of the last update by a user defined hyperparameter α.

This gives the final formula used to update the individual weights [1]:

$$\Delta w_{ij}(n) = -\eta \frac{\partial E}{\partial w_{ij}} + \alpha \Delta w_{ij}(n-1)$$

Computing this derivative for every weight from ground up every step would take vastly too long, but luckily gradients can be computed cheaply using backpropagation.

Backpropagation works similarily to the forward pass, where the first layer of the neurons is computed first, then the neurons connected to the first layer, and so on. In Backpropagation the error is computed at the end of the network and then backpropagated to neurons connected to the output neurons to compute their given gradient, repeating this process until the first neurons are reached [2].

Choosing hyperparameters is also both quite important and hard. Having a too small η for example results in very slow convergence (or sometimes being stuck in a local minima). Too big η on the other hand may lead to rapid divergence.

For example the network deployed in this thesis is very sensitive to changes in η, deviating from the optimal value results in constant high loss (no convergence) or rapid approach of infinite loss after a quick drop in loss.

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a subcategory of machine learning designed to tackle the problem of image classification - a problem, which is difficult to design algorithms by hand for. (How to tell an algorithm what a bird looks like?)

CNNs are a more restricted version of more general Artificial Neural Networks (ANN). They were specifically designed with the task of image processing in mind. Traditional Artificial Neural Networks tend to have an enormous amount of weights for large inputs such as images [3]. Modelled after insights of how the brain processes visual input a CNN exploits some properties of images as inputs to simplify the model. This restriction is both suitable for image input and beneficial for training time [8]. So instead of being able to connect every input neuron to every output neuron, a CNN usually consists of a mix of *Convolution*, *Pooling* and *ReLU* Layers. While so called fully connected (FC) layers are commonly used as the last part of the network, this thesis will make use of a fully convolutional network, one in which all the fully connected layers are replaced by Convolutional Layers.

2.2.1 Convolutional Layers

Convolutional Layers are the heart of Convolutional Neural Networks. A layer consists of a set of filters, which are parameterized by their size, stride and padding. Although the number of filters of a Convolutional Layers can be varied, all filters share the same basic parameters.

Usually a filter used in CNNs is small in size (≤ 10). This small filter is then slided over the input image, while being displaced by the given stride in each step. In each step the filter's weights are multiplied with the values of the input region in this specific position, extending to the whole input dimension resulting in one single output neuron.

So a filter of size 3x3 on an input image of depth 3 (for RGB valued color images) thus has $3 \times 3 \times 3 = 27$ input neurons, so it needs 27 weights. In each step it will look at its input region and multiply the output of the given neuron, which, in the first layer, is equal to the pixel value of the input image, with the corresponding weight.

As the filter has (usually) a smaller size than the image, sliding the filter over the input image results in a number of output neurons. All of these output neurons are then combined to build the output layer for the filter (see Figure 2.2).

Note that, when sliding the filter over the image, the weights are always reused. If the filter has size 3x3 for an input image of depth of 3, then - completely independent of the size of the input image, the filter will use $3 \times 3 \times 3 = 27$ weights.

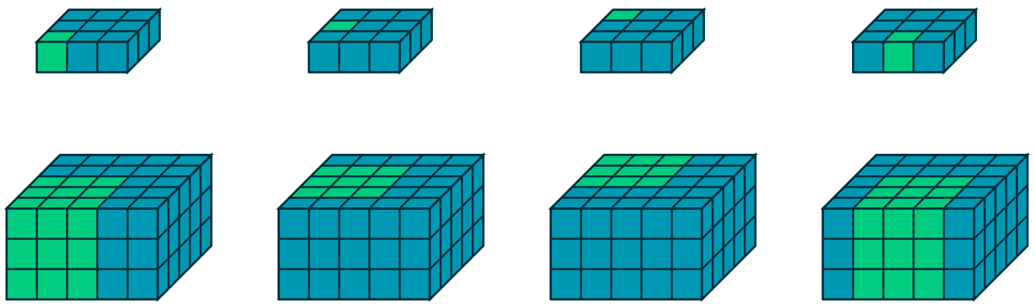

Figure 2.2: sliding a filter of 3x3 over an input image of 5x5 with a depth of 3, stride of 1 and no padding. Only one output layer and only the first four steps are shown.

This is different than using a normal neural network, where every connection would have its very own weight and no weight sharing would happen.

The stride parameter of a Convolutional Layer just tells the filter how large the jumps between consecutive steps are. By doing so the number of output neurons naturally decreases as well as the overlap between consecutive sliding steps (see Figure 2.3).

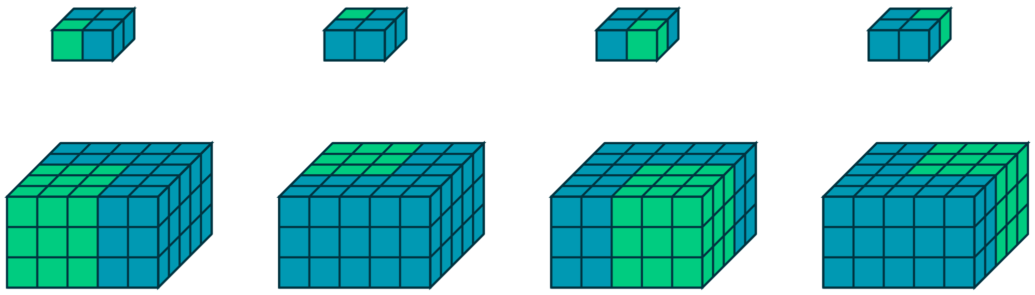

Figure 2.3: sliding a filter of 3×3 over an input image of 5×5 with a depth of 3, stride of 2 and no padding. Only one output layer is shown.

Finally the padding parameter defines how many neurons should be inserted around the border of the image (usually initialized with 0). This is sometimes useful if a specific output size is desired. Padding would result in the same network as Figure 2.3, if the input image were of size 3×3 and the padding would equal 1 (in both height and width).

Generally speaking the size of the output layer is

$$\frac{inputlength + 2 \times padding - kernelsize}{stride} + 1$$

where $inputlength$ is either input width or height, as the calculation is the same for both.

The motivation behind filters and weights sharing is, that these filters learn to detect small features in the input network, which in images usually consist of local pixel patterns. Sliding the filter over the image is done under the assumption that features can occur in every part of the image and thus general filters can be learned [3]. Such filters usually are edges in the first layers as seen in Figure 2.4.

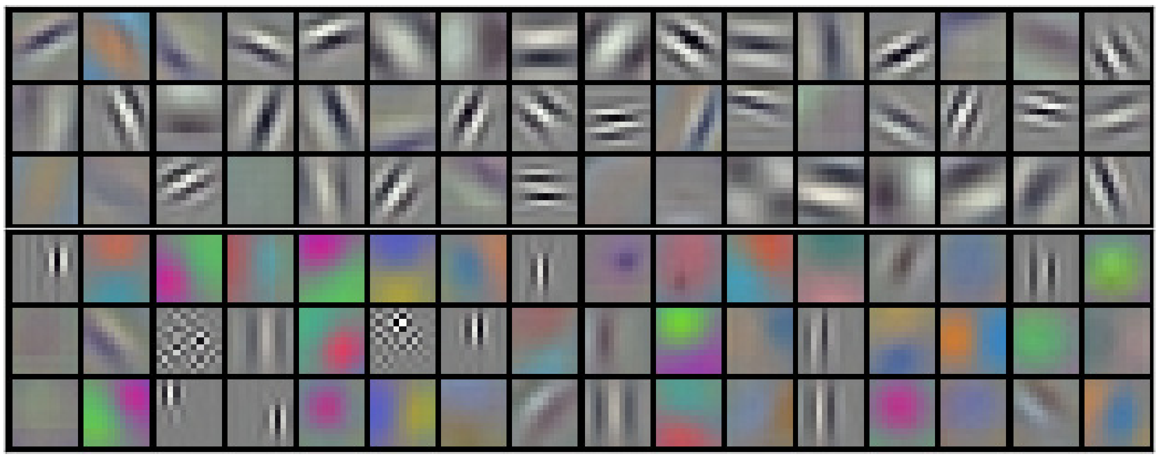

Figure 2.4: filters learned in the first layer of a convolutional neural network from a paper by Krizhevsky et al. [4].

2.2.2 Pooling Layers

Pooling layers are used to reduce the size of the input image to minimize the number of parameters and thus reduce overfitting [3].

Pooling layers take - similar to Convolutional Layers - an input patch of a given dimension. Other than Convolutional layers a pooling layer does not extend to the whole input dimension (see Figure 2.5).

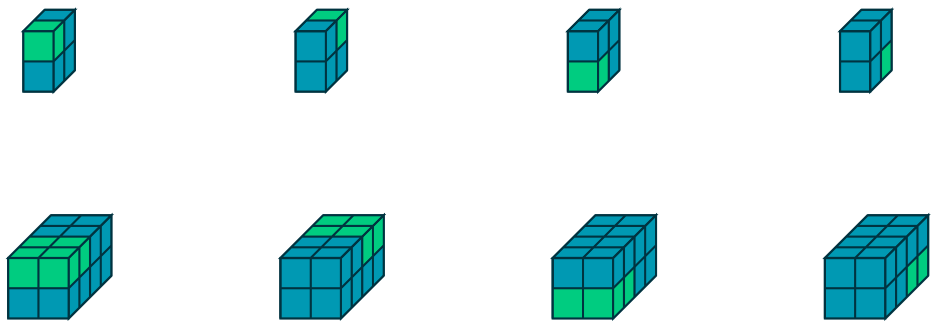

Figure 2.5: a pooling layer on a $4 \times 2 \times 2$ input with a filter of 2 and a stride of 2

While sliding the pooling window over the input, a user defined function is applied to the given input. Possible functions include *MAX*, resulting in the maximum value of the input patch, and *AVG*, which just averages the input patch. Pooling layers usually have a stride of 2 or larger in order to reduce the input size more drastically [3].

2.2.3 ReLU

After applying the convolution to the input, a function is usually applied to the output (see f in subsection 2.1.1). In a common Convolution Network this function is a so called rectifier or Rectified Linear Unit (*ReLU*), which, in the used framework, takes the form of $f(x) = max(x, 0)$ (see Figure 2.6) [3].

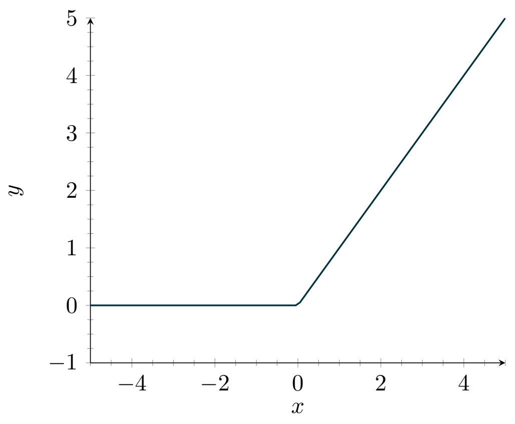

Figure 2.6: ReLU function used in this thesis

ReLU functions are by far not the only activation functions available for neural networks, they are just the most commonly used ones in CNNs, thus they are used in this thesis too.

2.2.4 Deconvolutional Layers

Deconvolutional Layers are not typically part of convolutional networks, but are necessary for this thesis, as the output consists of a full image and not just class probabilities.

Generally speaking Deconvolutional Layers are not much different from regular Convolutional Layers, they just work backwards, hence the name *Transposed Convolution*.

(The caffe documentation contains a detailed explanation)

Taking a quick look at the convolution example from before where a 3×3 filter was applied to a 5×5 input image resulting in a 3×3 output. The corresponding Transposed Convolution with the same parameters for this would be a 3×3 input image which outputs a 5×5 blob (see Figure 2.7).

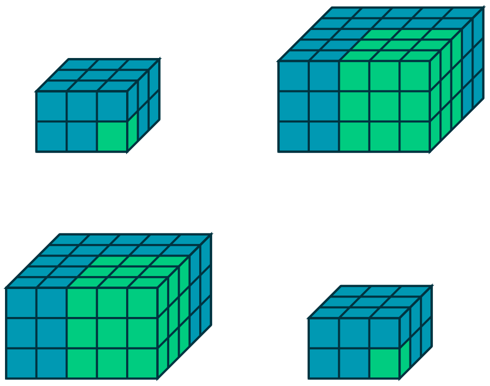

Figure 2.7: normal Convolutional Layer in comparison to Transposed Convolutional Layer

While the normal Convolutional Layer worked by multiplying each value of the filter with the corresponding value in the input blob, the Transposed Convolution works by multiplying all filter values by the single input value and adding the result into the input blob at the correct position. Note that *adding* the values in the output blob is very important as the output windows can now overlap with transposed convolution (see Figure 2.8).

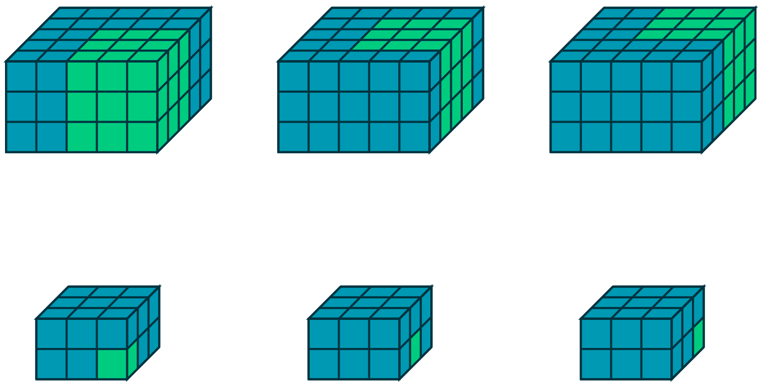

Figure 2.8: Overlapping of output windows in Transposed Convolutional Layer

Analogous to the Convolutional Layer the Transposed Convolutional Layer also shares all the same filters for each input layer to reduce parameters.

Transposed Convolutional Layers will play an important part in being able to produce full image segmentations later.

2.3 Used Tools

In the following section the tools used in this thesis will be introduced.

2.3.1 Caffe

As implementing a neural network as well as all the needed learning algorithms would be unnecessary, the existing deep learning framework *Caffe* was chosen. Caffe is a commonly used open source deep learning framework.

One obvious advantage of using Caffe is its speed, as Caffe can utilize the GPU for training. This is especially useful when training on large inputs such as images.

Declaring own network structures is also easily done with Caffe's own network description files.

```
layer {
  name: "pool2"
  type: "Pooling"
  bottom: "conv2"
  top: "pool2"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
```

Figure 2.9: example pooling layer definition in Caffe

Defining a pooling layer for example is as easy as in Figure 2.9. Each Layer has its input (bottom) layers and output connections (top), a type and the parameters defining how the Layer will behave.

The different configurations needed are all specified in the openly available online Caffe documentation.

Furthermore Caffe features a large amount of tutorials including code samples, which makes learning the framework fast.

Even the paper, from which the approach for this thesis was adapted used caffe for its networks so adaption was straightforward.

2.3.2 Docker

To simplify installation, Caffe was deployed to the server using Docker. Docker can be used to package entire applications with all needed configuration and dependencies into a package called a *container*, which can then be installed on every computer running docker. Docker can be described as virtualisation but more lightweight. The advantage of using Docker is, that all dependencies are already inside the container and thus installing Caffe using Docker, was really just installing Docker, then copying the Caffe container to the server and finally waiting until the container has finished setting up.

Using Docker would have been no option if `nvidia-docker` would not exist. Usually containers can not directly access hardware from the host, but nvidia docker forwards the GPU, which was very important, as the network is trained using the GPU. Training on the CPU would haven been theoretically possible, but training on the GPU was around 60 times faster than on a CPU.

2.3.3 OpenCV

For preprocessing the images OpenCV was chosen as it implements all algorithms needed and has high performance due to the possibilty to use OpenCL to compute the filters.

OpenCV offers a nice C++ API, which was very easy to use. Similarily to Caffe, it featured extensive online resources and documentation.

2.3.4 Point Cloud Library

Finally, Point Cloud Library (PCL) is used to visualize the reconstructed needle. PCL offers fast rendering of huge amounts of points, so it is more then capable to visualize the results from reconstructing the needle on even non-high end computers.

2.3.5 `mlpack` and `armadillo`

For regression and matrix operations the C++ library `mlpack` as well as `armadillo` were used, which provided easy access to commonly needed operations.

2.3.6 Rust

As a huge part of training a network is preprocessing the data and getting it into the right format, custom data transformation tools had to be written. The language chosen here was Rust, a new systems programming language, which made it easy to write performance conscious code while having all the advantages of a package ecosystem to choose image processing and GUI packages from.

Having a strong modern type system made it easy to quickly write data processing tools, while the compiler catched almost all bugs at compile time.

3 Needle Detection

Needle Detection describes the problem of extracting which pixels from the different image slices correspond to the needle (see Figure 3.1).

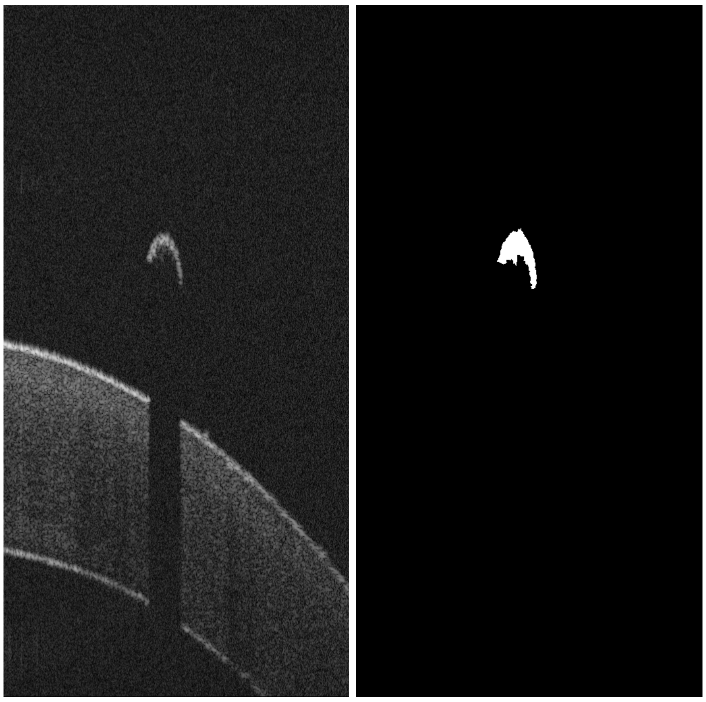

Figure 3.1: raw input slice and desired output for the segmentation

This has to be done for every slice in the dataset and finally combined to a complete needle.

3.1 Conventional Algorithm

Basically the conventional algorithm tries to detect the shadow in the image. It does this by scanning the image column by column and calculating how many white pixels were found. If the white pixels fall below a certain threshold the column is said to contain a shadow.

Finally from all components, that are in a component with shadow, the largest is taken.

This, however, is not enough, as for example with a shifted retina the right part of the retina will be detected as needle (see Figure 3.2).

A solution for this would be to trace the top pixels and detect the needle by the jump, or exclude all components, that are in short distance to the images border.

Figure 3.2: conventional algorithm has problems with thin retina

This is not implemented in this thesis, as this is a topic for further research.

3.2 Approach

This thesis follows a technique described in the paper *Fully Convolutional Networks for Semantic Segmentation* using Convolutional Neural Networks trained pixel to pixel to achieve image segmentation [5].

The first thing needed to build a segmentation network is to build a network that is able to classify the data.

3.2.1 Classification Network

Image classification networks usually follow a similar basic structure of alternating Convolutional Layers and ReLU Layers with Pooling Layers inbetween. After the layers have reached a sufficiently small size through repeated reduction in size by pooling layers, typically one (or more) fully connected layers are added at the end of the network, while the last layer has just as many neurons as the input data has labels. Each output neuron in the last layer is then interpreteted as returning a probability for the given class [3].

Figure 3.3 depicts such a simple network structure.

Note that while Convolutional Layers as well as Pooling Layers all output a set of 2D layers, the Fully Connected Layer discards all position information and is just a long 1D line of neurons.

This seems to be rather unfortunate when trying to create a network, that segmentates parts of the image, which, by definition, is a task, where the position is an important aspect.

But as it turns out, those image classification networks can easily be extended to form image *segmentation* networks as described in the paper *Fully Convolutional Networks for Semantic Segmentation* [5].

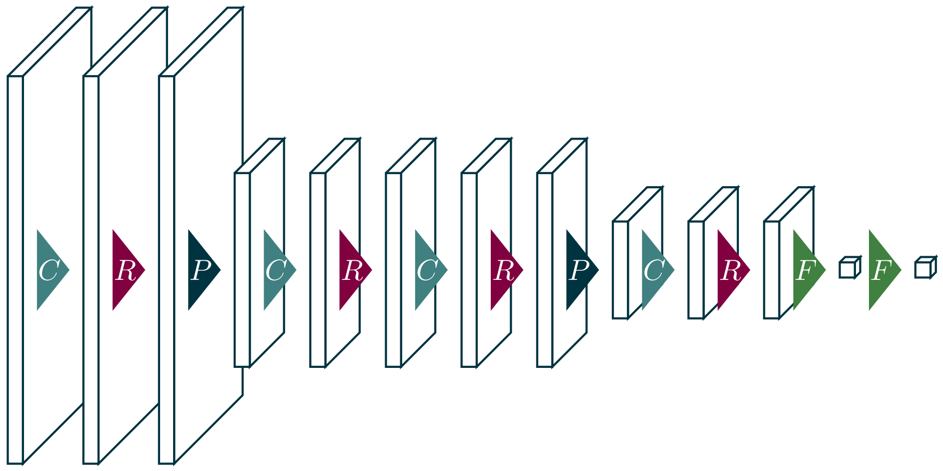

Figure 3.3: Example structure of basic classification network, C stands for a Convolutional Layer, R for a ReLU function, P for Pooling and F for a fully connected layer. Depth of layers not shown.

3.2.2 Transformation to a Segmentation Network

The first step to take when transforming a classification network to a segmentation network is not very surprisingly done by removing the last fully connected prediction layer and converting all fully connected layers to convolutional layers. Converting a fully connected layer to a Convolutional Layer is done quite easily: Simply create a Convolutional Layer where the filter size spans the entire input region [5].

The just removed fully connected prediction layer is then replaced by a 1×1 convolution with number of filters equal to the classes present. This is thought to create a coarse prediction heatmap for the different classes [5].

After preparing the classification network, the network must be extended to a segmentation network. This is done by upsampling the coarse output from the classification network using Transposed Convolution Layers (Or Deconvolutional Layers as they are called in Caffee). This alone would yield way too blurry results, but luckily a solution exists: Adding skip connections from previous layers of the classification network to the segmentation network [5].

Taking for example a simple classification network seen in Figure 3.4.

After replacing the final layer with a Convolutional Layer, one can add Transposed Convolutional Layers as one sees fit. Important now is, that if upsampling the layers is done, adding the output of old layers to the current layer is possible (if the sizes are comparable).

In Figure 3.5 this is done after the first upsampling to combine the output of the 3rd pooling operation with the current layer.

But before combing the output of the previous layers with the current layer, the paper suggest to add an additional prediction layer, which is just a Convolutional Layer with a 1×1 filter and again with the number of filters equal to the number of classes in the dataset.

After applying the convolution, the paper suggests using an elementwise add to combine the two layers, but here a simple concatenation of all the layers along the depth di-

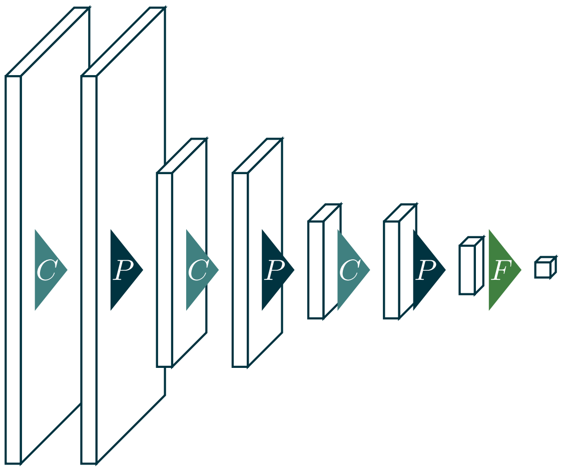

Figure 3.4: Example of classification network. C stands for a Convolutional Layer; as each C is followed by a ReLU, this is omitted, P for Pooling. Depth of layers not shown.

mension is chosen, to support concatenation of differently sized blobs.

Of course one is able to add additional Convolutional Layers inbetween the Concatenation and Transposed Convolutional Layers as done in the example directly between the classification and the segmentation network as well as after the first upsampling.

3.2.3 Training

Training the segmentation network has to be done in two steps.

At first the classification network is trained with image/label pairs. After reaching a sufficient accuracy rate - according to the paper - the weights of the classification network must be transfered to the segmentation network - which is possible as the classification network is a part of the segmentation network.

After that one can train the full segmentation network with the image/image pairs, where the second image shows the desired segmentation for the given input image.

It was found after many trials that this is only sometimes necessary and heavily dependent on the loss function used. Training this network for example with an `EuclideanLoss` layer without transplanting the network did not result in any convergence.

On the other hand training with a `SoftMaxLoss` layer, sequence information enhanced and smaller network, transplanting was not necessary.

3.3 Preparing the Data

Instead of training the network with the raw images, already preprocessed images are feeded to the network. This preparation is partly the same done when detecting the needle with a conventional hand written algorithm.

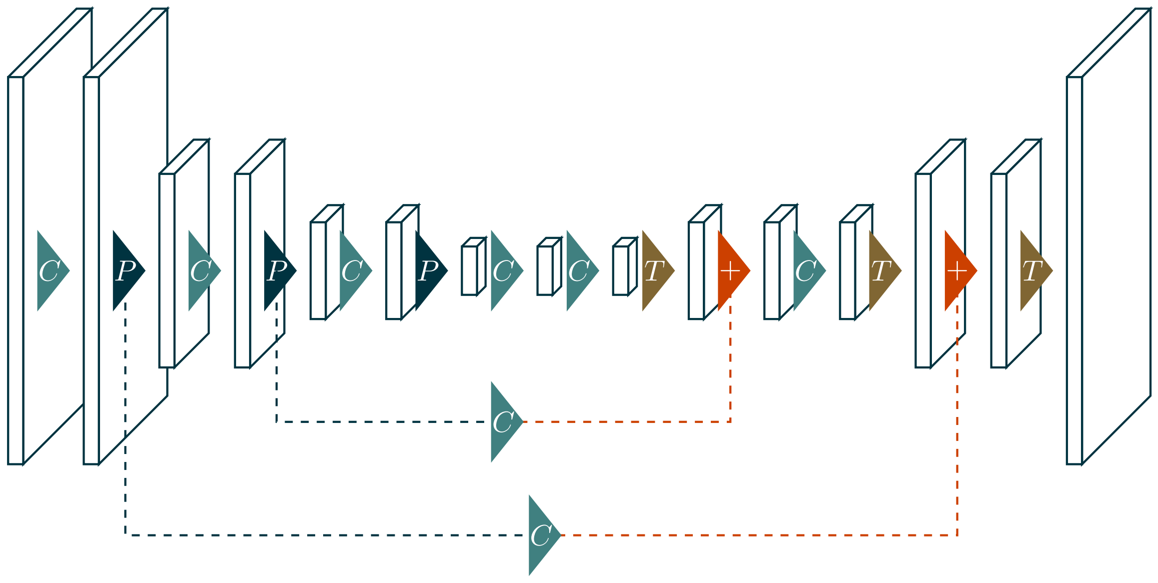

Figure 3.5: Classification network from Figure 3.4 extended to segmentation network. C stands for a Convolutional Layer; as each C is followed by a ReLU, this is omitted, P for Pooling, $+$ for Concat Layers and T for Transposed Convolution Layers. Depth of layers not shown. Note that this is only one possibilty!

3.3.1 Treshold

The first transformation done to the input images is taking a treshold, which results in more contrast, but also vastly more noise (see Figure 3.6).

3.3.2 Medianfilter

In order to reduce the noise introduced by tresholding the image is transformed using a medianfilter, which works by looking at a small rectangle around each pixel and sets the pixel to the value that occurs the most around in given rectangle.

3.3.3 Removal of small components

Finally all components, that are too small to be a needle and thus likely are noise, are removed from the image. This can be done by calculating components of white pixels and then counting how many pixels in each component occur, discarding those which fall below a certain threshold (see Figure 3.6).

3.3.4 Sequence Information Encoding

Additionally an experimental approach was taken to include some of the sequence information into the images.

Taking a short look at Figure 3.7 one can see that the needle travelles up at first in the dataset, and finally after leaving the image, a reflection can be seen travelling down again as seen in Figure 3.8. A human labelling the dataset by hand is also able to exploit this

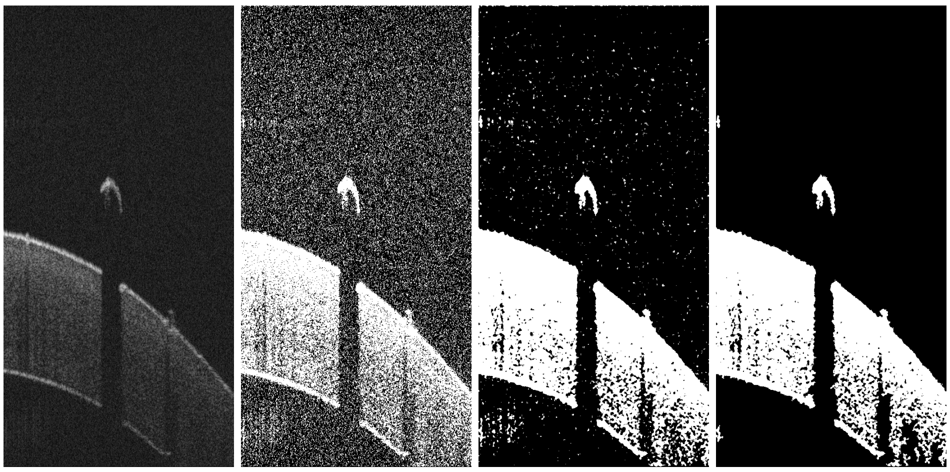
Figure 3.6: All steps taking during preparation

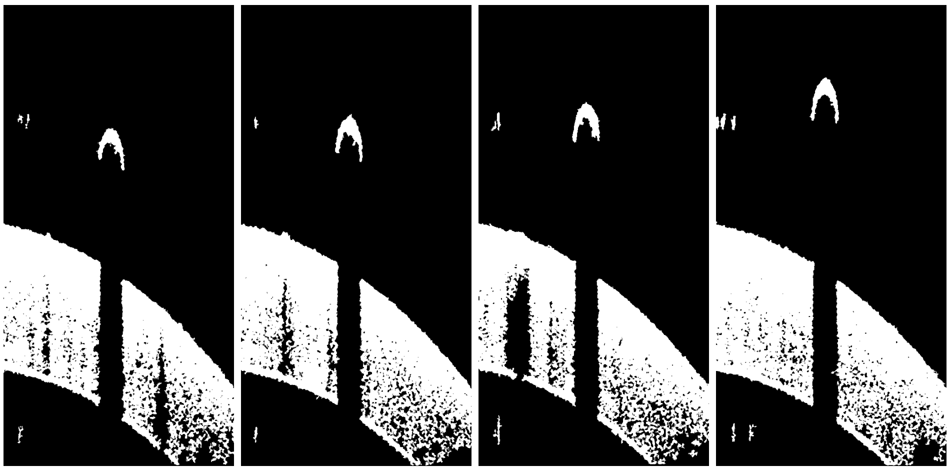
Figure 3.7: Samples from the dataset, ordered, with needle

fact, which is needed in the case of distortions, as just seeing a single image is not always conclusive whether it's actually a needle or not.

In order to give the network access to the information whether the blob is moving up or down, a simple solution is to supply the network only with the last 2 images additionally. This has the advantage, that all of this information can be encoded into the RGB channels of the image and thus no further database creation tool must be written for custom multidimensional data (see Figure 3.9).

By doing so, differentiating between a shape travelling up a shape travelling down can be done by taking a look at the border of the shapes. If the shape has a red line at the top and a blue at the bottom it must be travelling up. On the other hand, if red and blue are switched, the shape is travelling down (see Figure 3.10).

In theory the network should be able to learn this differentiation and in fact the classification performance of the network does increase with this modification.

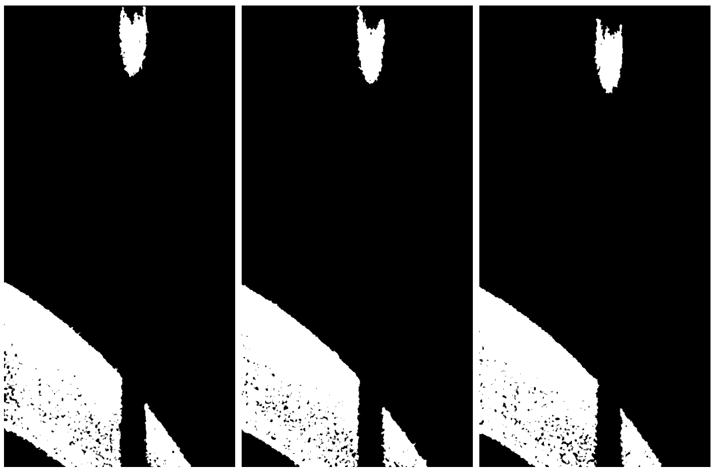
Figure 3.8: Samples from the dataset, ordered, reflection after needle was seen

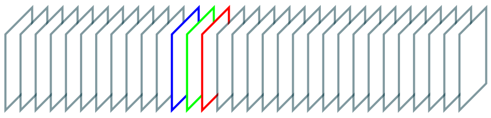
Figure 3.9: When this is the whole dataset with the first slice on the left and the last on the right, the current image is encoded in the red, the previous in green and the one before the previous in the blue channel

3.3.5 Scaling

Finally the input images for the network are scaled down by a factor of 4 to speed up both training and prediction. This is only possible due to the fact, that the images exhibit only white or black pixels, but nothing inbetween. Thus it is possible to combine the low resolution output with the high resolution picture in the beginning to obtain a high resolution prediction in the end.

3.3.6 Labelling

In order to be able to train the network all the input images had the be given correct labels.

For this a conventional algorithm for detecting the needle, implemented in C++ was used [7].

As this algorithm was not able to reach 100% accuracy, fixing the labels by hand was necessary. For this a custom tool was written to be able to quickly draw in the parts which were needles into the images (see Figure 3.11).

The tool first preprocesses all the given input data with the previously described methods and applies the conventional algorithm for needle detection to all the images.

It then opens up a window where a human can select which parts belong to a needle per mouse.

By pressing the arrow keys one can navigate through all the images.

Furthermore the tool creates all label files needed for training.

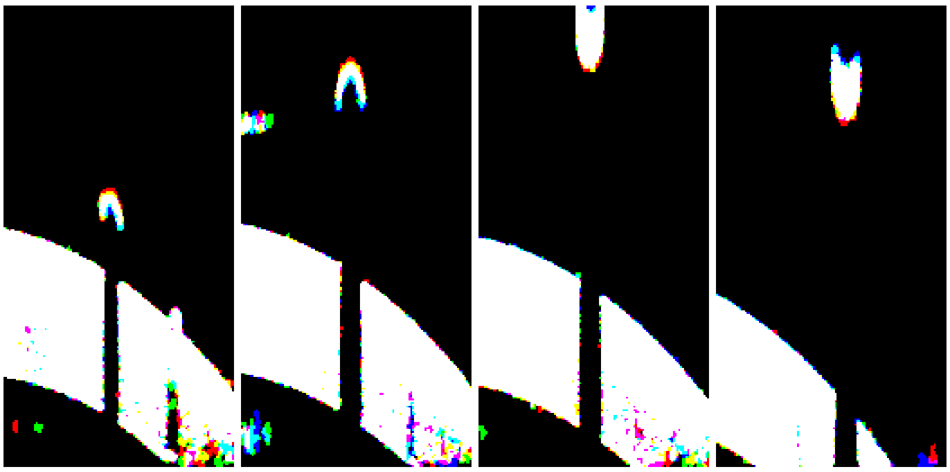

Figure 3.10: Input images augmented with sequence information. The first two are shapes travelling up, while the last two are shapes travelling down.

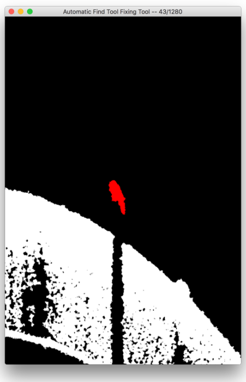

Figure 3.11: Fixing Tool

3.4 Network Design

The final designs of the network are a result of very many attempts of training the network. Unfortunately there are no conclusive rules for designing neural network architectures (yet) other than rules of thumbs.

Many attempts randomly resulted in exploding gradients, or in constant, but high loss equilibriums. Sometimes changing the hyperparameters was the solution, and sometimes the layer configuration was wrong.

Just finding a network that converges to a lower loss value than the initial loss was already challenging, but unfortunately low loss does not equal good results.

It is not clear if the final network architecture is too big and a smaller with different parameters could produce the same or even better accuracies. This is a topic yet to be researched.

3.4.1 A Long Way of Trial And Error

Training neural networks sadly is not as straightforward as one might think. Here only a very short excerpt from all the failed attempts to make the network detect the needle.

Input Scaling

When training with a `SoftMaxLoss` layer the input data has to reflect how many classes there are. For example if the images have to be segmented into 42 areas, then the ground truth image must contain values between 0 and 41. If however the scaling of the input image is somehow wrong and thus the values are wrong (only 0 to 10), the network will *not* warn about the inequality.

Wondering why the network is unable to predict some values is now rather frustrating guesswork, what went wrong.

Being Stuck in Local Minima

Another common problem is somehow being stuck in a local minima. This can sadly have a handful of reasons. It can also occur - as in this thesis - when the initial values are not correctly initialized and the network is stuck. Sometimes the network was able to "shake itself" free from the minima, but only when the learning rate was high enough to allow this. A too high learning rate however quickly lead - after a fast drop in loss - into a sudden increase again (probably due to high momentum), which leads right back into the previous local minima.

Slightly Wrong Results

The most frustrating problem with training neural networks, however, is, when they *sort of* do what they are supposed to, but then have small artifacts and errors (see Figure 3.12). Is it a problem with the input data? With the loss function? One of the parameters in the ≈ 600 line parameter file for the network? Is this just a general problem and this kind of network is generally unable to learn to predict this task?

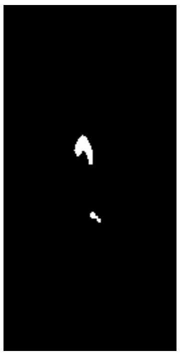

Figure 3.12: A slight artifact in segmentation

It seems that one has to develop some "intuition" about neural networks to quickly identify where exactly the problem is and how to solve it.

3.4.2 Final Network

The final network architecture is rather simple, just a short classification network, that turned out to be sufficient for this task of needle reconstruction and a equally small segmentation extension (see Figure 3.13).

The architecture closely follows what was described previously. One major difference are the scoring layers connecting layers form the classification network with those in the segmentation network: The paper suggests using the number of classes as the number of filters, but allowing the network to have more filters than classes resulted in better convergence in this network[1][5].

Interestingly it was found, that a good classification network (precision of $\geq 98\%$) yields a suboptimal segmentation network and vice versa a suboptimal classification network (precision of $\approx 89\%$) yields a good segmentation network. Maybe the suboptimal classification network overfittes, but the segmentation network needs the computational space from the classifiation network to perform well.

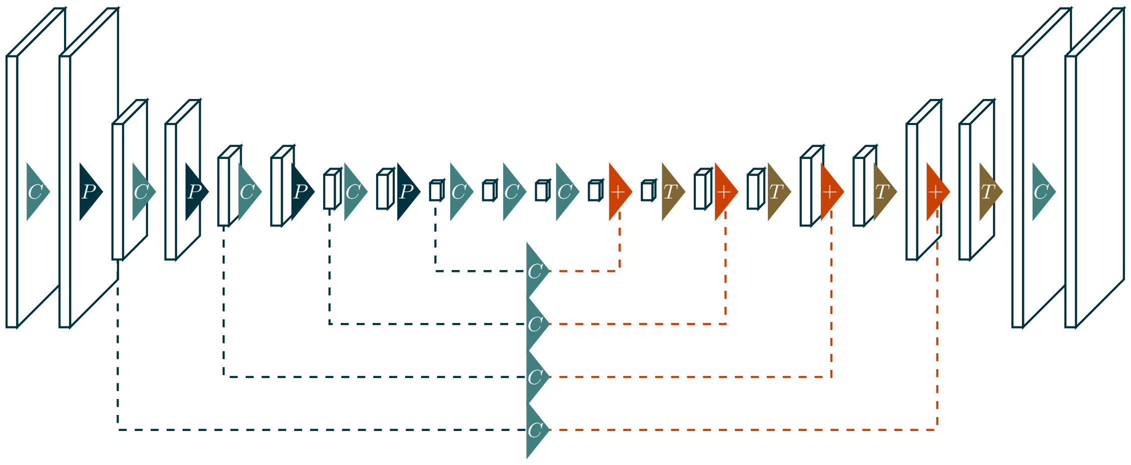

Figure 3.13: Final network architecture, this includes the classification network in the beginning and is extended to the segmentation network at the end

3.5 Results

For being able to understand the segmentation precision a special approach was chosen. As every dataset exhibits the same pattern of *no needle* $\rightarrow$ *needle* $\rightarrow$ *no needle but reflection*, all information is mapped such that at 25% the needle begins and at 75% the needle ends. This makes comparing the different datasets not as confusing.

So if in a dataset the index of the first needle image (taken from the ground truth) is at 50 and the currently evaluated image has index 25, it will be shown at $\frac{1}{4} * \frac{25}{50} = 12.5\%$

[1]at least it seems that way, using only 2 layers results in an apparent stagnation in loss decrease at ≈ 0.0044, while increasing the layers the loss can go below this value. Note that the learning rates had to be adjusted, so it is very well possible, that a different learning rate in the first case would have resulted in a similar loss

The images are then sorted into buckets of 2% and the values are averaged over all images in the given bucket.

Comparing each image it is measured how many pixels in the end result where additionally there (false positive) and vice versa how many pixels were missing from the ground truth (false negative).

3.5.1 Conventional Algorithm

At first the precision plot for the conventional algorithm is given for comparison.

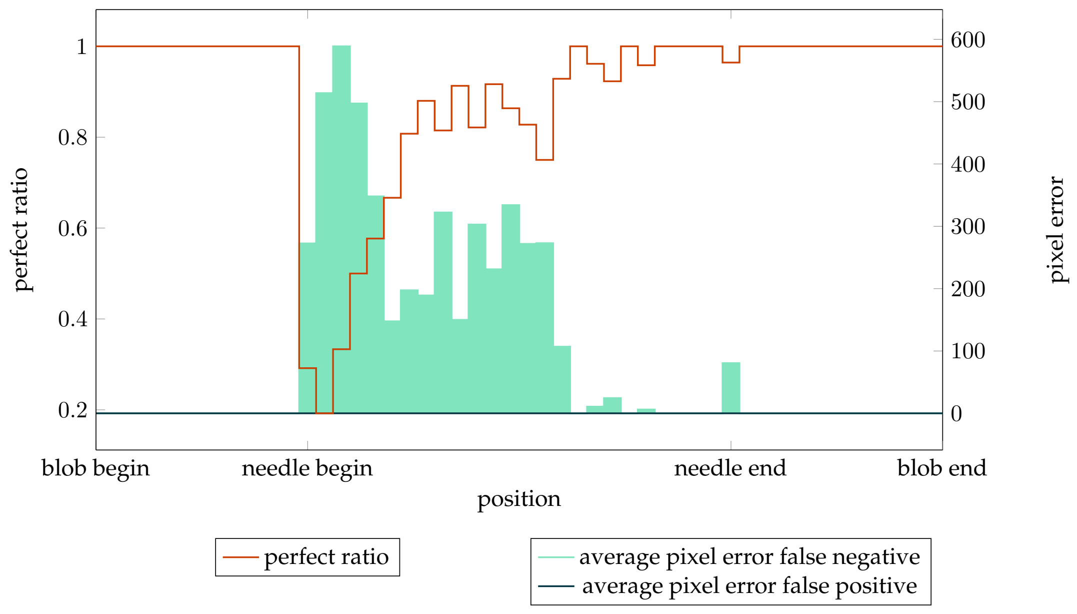

Figure 3.14: plot of precision of conventional handwritten algorithm

The algorithm was tested using the same testing set as the CNN. The red line shows the ratio of perfectly filtered images (compared to the ground truth), while the two cyan lines each show the false negative and false positive pixel errors. Note that the algorithm has *not one* false positive error, but quite a lot of false negatives.

Also noteworthy is the sudden drop of accuracy when the needle begins. This comes from the fact, that the shadow is not instantly seen or may not extend through the whole retina during the very first beginning. So an algorithm looking for the shadow naturally has a disadvantage.

The metric, however, that will be more important later, is how wrong the bounding box is calculated. A missing bounding box is (if occurance is rare) not as bad as a wrong bounding box, as a wrong bounding box may distort needle reconstruction.

The graph (see Figure 3.15) shows - as expected - an increase in missed bounding boxes in the beginning. Also, as there were no false positives, there is a constant rate of 0 for bounding boxes, that were in the output images of the conventional algorithm, but not in the ground truth.

Furthermore the width and peak point error (each measured how many pixels they were

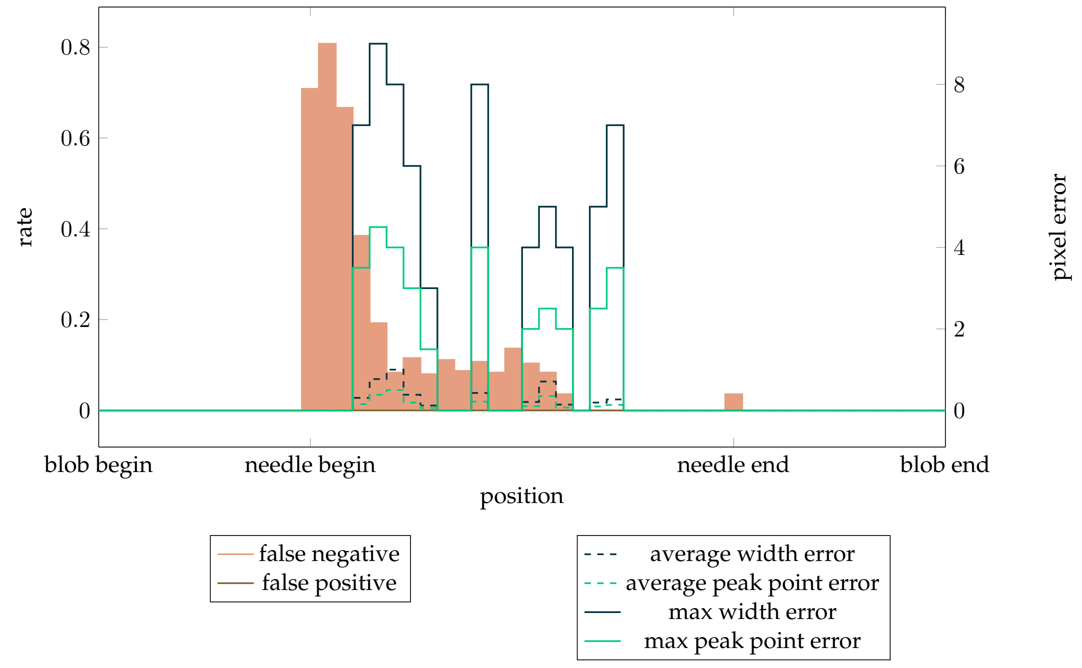

Figure 3.15: plot of precision of conventional handwritten algorithm for bounding boxes

off) is low. Averaged over each 2% slot, it does not even exceed 1 pixel error. [2]

The maximum error is with around ≈ 9 pixels, which happens (looking at the average) rather rarely.

The rate of missing bounding boxes, however is quite dramatic in the beginning with around 80%, but quickly drops to a steady 10% until around the middle, from where basically all bounding boxes are calculated almost correctly.

3.5.2 CNN

Segmentation precision is measured between the ground truth for segmentation and the output of the network combined with the input. The combination step is done outside the network, as it is easily implementable (although the network might have been able to learn this step).

The combination tool simply upsamples the output of the CNN (and applies a small median filter to reduce pixel noise) by 4x and then combines both images. This is done by masking the full resolution input image with the output of the CNN and then calculating which components of the input image are still largely there (a threshold must be chosen).

For example taking a look at an sample image in Figure 3.16.

The first image is the input to the network, while the second is the upsampled output of the CNN. The input is then masked with the output, which is shown here as a overlay. All white regions are in both input and output image, while all red are only found in the

[2]Interesting to note is the apparent dependency between peak and width error. This is the case as the peak point is the upper left corner of the bounding box plus half of the width. Also as the peak point is located at half the width, the peak point error naturally is scaled by 2, if no error in z direction occured.

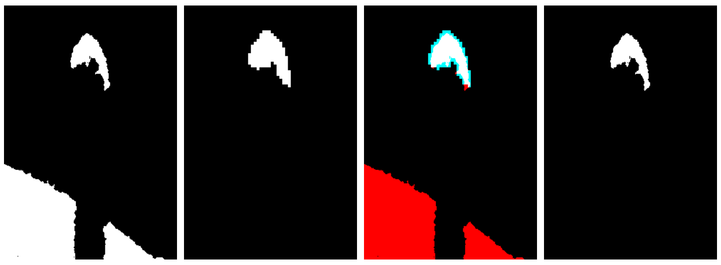

Figure 3.16: The steps in combining the output of the CNN with the input to obtain high resolution images

output image. The cyan areas are regions only found in the output.

Finally the result image contains all components, that have been masked by at least 60% of the CNN output.

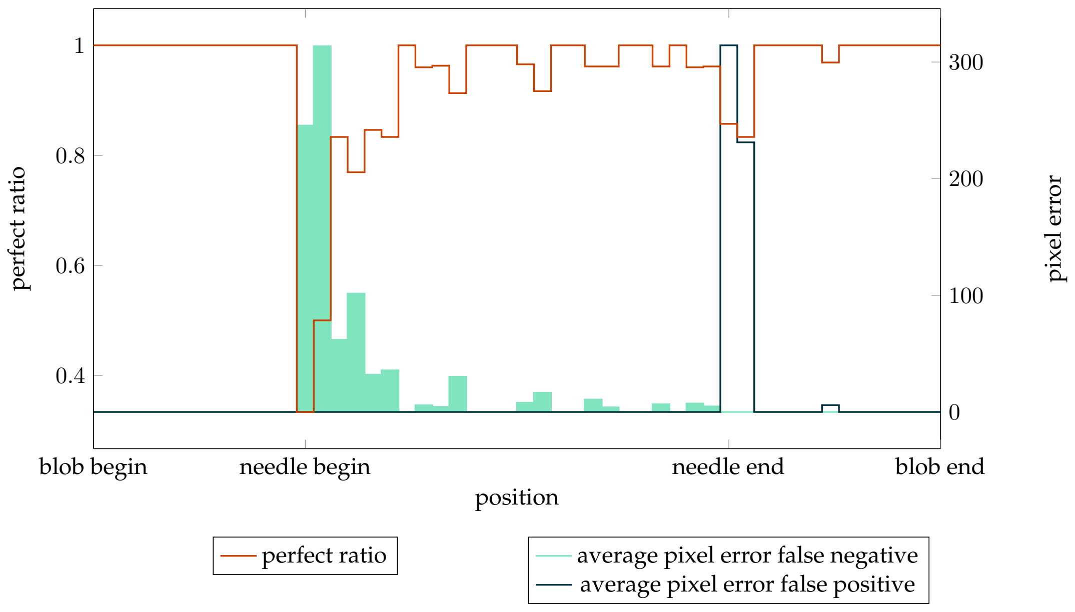

Figure 3.17: plot of precision of CNN

Taking a look at the same metrics, but this time with the output of the neural network (see Figure 3.17), it can be seen, that the neural network reaches a higher overall rate of perfectly predicted needle positions, while having less false negative pixel classifications. In contrast to the conventional algorithm it has some difficulties with finding the end of the needle and has some false positives where the needle should have ended.

Comparing the width and peak point error of the neural network (see (Figure 3.18)) with the conventional algorithm, again, the neural network is able to find more bounding boxes than the conventional algorithm. Similarily to the conventional algorithm it has trouble finding the needle in the beginning and this time also when the needle ends.

The average error in width and peak point is also comparable. The maximum error in peak and width segmentation however is higher with the neural network (around ≈ 12).

Note that this is the result of finetuning all parameters for the needle to the test data set.

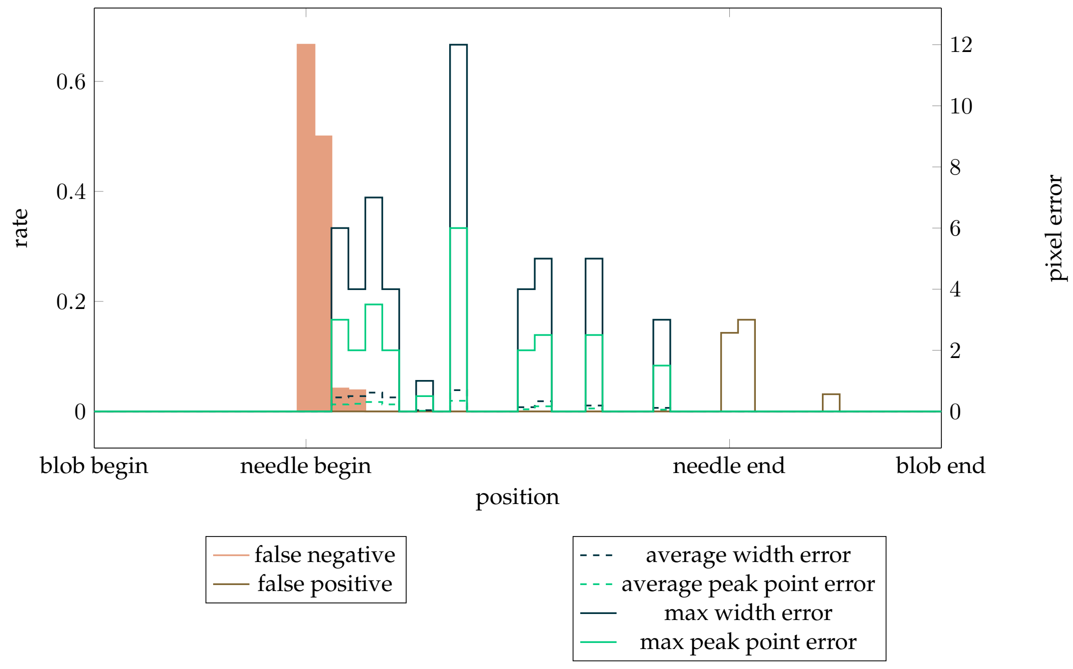

Figure 3.18: plot of precision of CNN output for bounding boxes

As the dataset is rather small, it is possible, that with more data the conventional algorithm shows more stabilty. A bigger dataset is needed to be able to fully compare both methods.

It is however remarkable, that the CNN was able to learn how to detect a needle and finally even segment the needle from the images.

3.6 Performance Analysis

Finally the performance was tested on the same server used for training.

All analysis was done on a Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz with a GeForce GTX 980 Ti.

3.6.1 Per Image Performance

Stage	μ ms	σ
Loading	0.71	0.10
Filtering	7.87	2.22
Detection	17.02	20.55

Figure 3.19: Conventional algorithm performance

The conventional algorithm just loads the data, preprocesses it and then tries to find the needle.

As processing for every image is completely independent from all other images it is possible to completely parallelize the process to 128 threads, which could yield a theoretical performance of ≈ 30 analysed datasets per second.

Note that if the dataset is dumped directly into the main memory of the computer loading from disk won't be necessary!

It is also important to note, that the conventional algorithm was not extensively optimized, as it only served as a reference implementation. It exhibits quadratic runtime due to an unfortunate merging algorithm, which can be seen in the standard deviation. Implementing merging with a simple bounding box check should be sufficient and will push likely push detection time to $\approx$ 10ms or less.

The CNN involves more steps, as it has to pass through more preparations steps. Unfortunately as the CNN interface is in Python a small roundtrip was needed through saving and loading the files. It is possible to combine the C++ code directly with the the CNN and save those times, but this was not attempted in this thesis.

Stage	μ ms	σ
Loading	**0.72**	**0.11**
Filtering	**5.64**	**1.11**
Sequence Enhance	**0.27**	**0.03**
Saving	0.58	0.06
Loading	1.81	0.23
CNN	**97.46**	**4.67**
Saving	0.94	0.06
Loading	3.02	0.16
Combining and Bounding Boxes	**6.63**	**0.38**

Interestingly, although filtering is the same in both cases, it takes less time in the preprocessing for the CNN. This might be the case, as the preprocessing tool for the CNN loads everything at first, and then filters everything. Thus the resulting differences may be due to cache differences.

Also this shows one big difference between the conventional algorithm and the CNN: The CNN shows a rather constant processing time, as the number of operations is always the same. This is not necessarily the case with a conventional algorithm as in this case.

3.6.2 Batch Performance

Caffe offers to process all given images in a batch. This means, that per forward pass not one, but the whole batch is being processed, which speeds up processing.

Here only the CNN processing steps are shown.

Compared to the non batched version batching is able to be $\approx$ 10ms faster.

3.6.3 Comparison

Looking at performance the conventional algorithm has an absolute advantage here. It requires both less steps and is faster by a factor of more than 2. The rather slow perfor-

Stage	μ ms	σ
Loading	1.69	0.21
CNN	**83.74**	**1.86**
Saving	0.78	0.68

Figure 3.20: Detection the needle with a CNN and batch size of 128

mance of the CNN is probably a result of the number of layers in the network as well as the problem of transfering everything between CPU and GPU.

4 Reconstruction

The next and last step after having found all needle slices in the the dataset is to assemble all those parts into a needle.

For this part only reconstruction of a special kind of needle is shown. This special kind of needle has a very distinct feature which is easily detectable: A change in diameter shortly after the tip (see Figure 4.1).

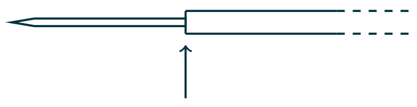

Figure 4.1: Looking at the needle from above, the tip is on the right. Note the change in diameter

4.1 Point of Radius Change

After detecting the different needle parts in every input slice, a very simple thing to do is measuring the width of each needle part (see Figure 4.2). This makes the change in diameter very obvious.

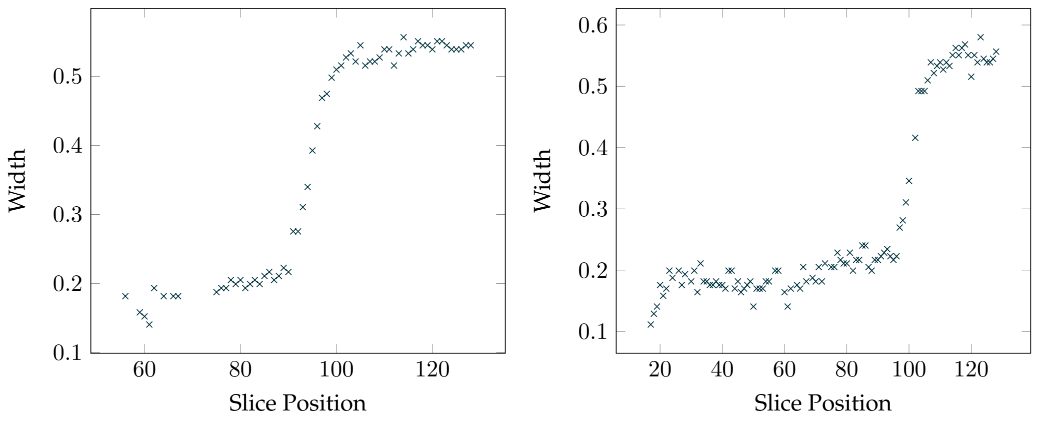

Figure 4.2: Plot showing how the width of the needle shape in the images changes in two different datasets

Unfortunately due to manufacturing and imaging errors there is no single point of radius change but a gradual change. The desired point then is when the gradual change has stopped. As all of the width values are noisy one has to find the point of change via regression with linear functions of the following form

$$y = mx + t$$

The basic form of the function can be seen in Figure 4.3.

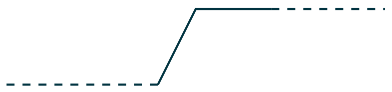

Figure 4.3: Basic function used for regression

As only the point where the change stops is important, one can discard all the points with insufficient width and only regress the rest. This makes regression easier as only two lines have to be regressed.

Regressing two lines is done by finding an optimal split between the points. For this simply every possible split is calculated, regressed and the error of the regression saved. Finally the algorithm iterates over all errors and chooses the split, which has the least error in both regressed lines.

A result of this can be seen in Figure 4.4.

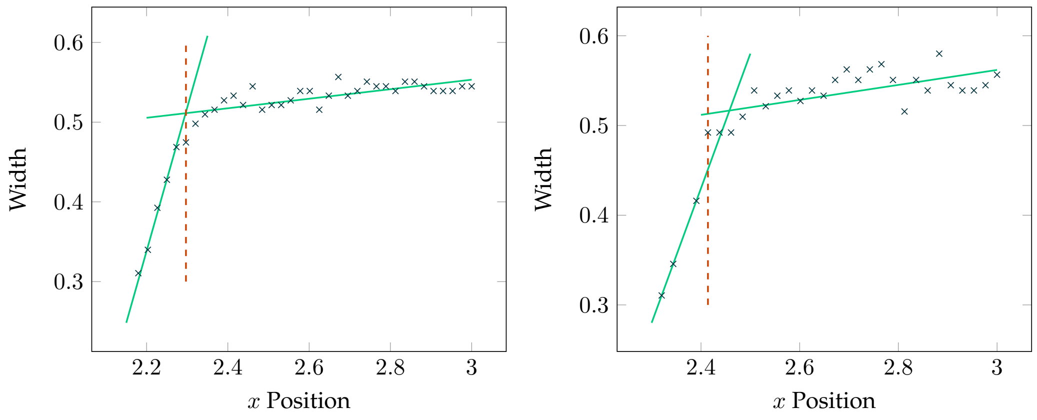

Figure 4.4: Only the thicker parts (width ≥ 0.3) are shown from Figure 4.2. The lines are the optimal regressed linear functions. The red line shows where the split was found

Using the split, the datapoints can be separated into points belonging to the needle body (the thick part) and the needle tip.

4.2 Radius

Needle radius regression is also needed for correctly reconstructing the needle. For this the needle parts are now analyzed from above (see Figure 2.4), while special attention is drawn to the edges of the needle parts as well as the *peak point*, which is simply the middle point between the edges.

The only radius that is needed, is the radius of the thicker needle body. Luckily the split where the needle body starts was already calculated, so filtering out the points belonging to the needle body is very easy now.

The basic idea of finding the radius is regressing a line through the peak point of all the body needle slices and another line through one edge of the needle.

When regressing the lines it is very important, that both are parallel. This can simply be done by regressing the line through the peak points and reusing m for the regression through the edge points, so computing the last parameter t can simply be done by

$$t = \frac{1}{n}[\sum_{i=1}^{n} y_i - mx_i]$$

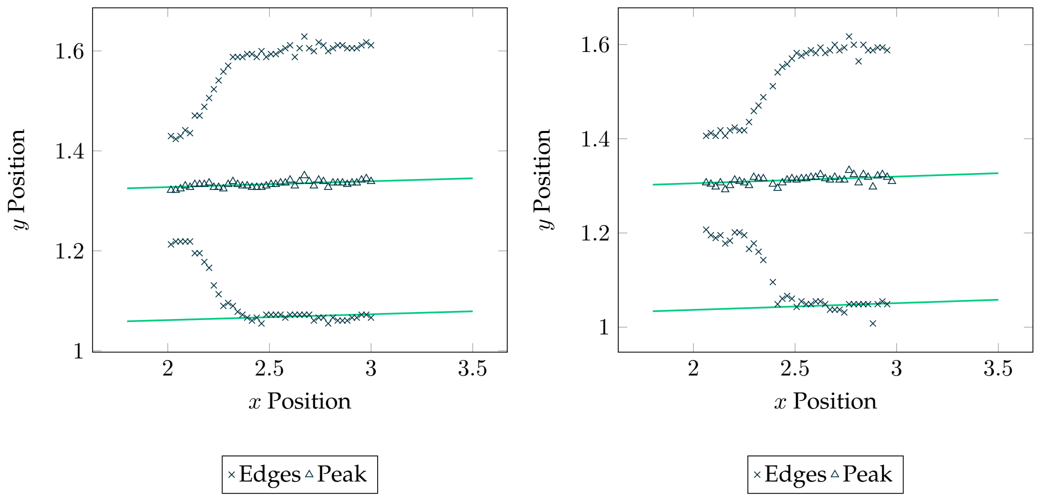

Figure 4.5: Needle as seen from above; Peak point is just average between two edges.

After both lines were found, the distance can be calculated (If the lines were not parallel here, there would be no definite distance).

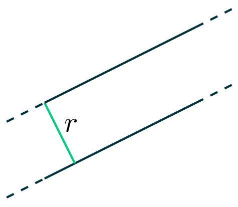

Figure 4.6: distance between lines equals radius of needle body

Assuming the line through the peak points is given by

$$y = mx + t_1$$

and similarily the line through the edge points is given by

$$y = mx + t_2$$

Note that the variable m must stay the same in both cases, then the distance between those two lines equals the radius.

Then the radius is simply given by

$$r = \frac{|t_1 - t_2|}{\sqrt{m^2+1}}$$

Also note that choosing which edge to use is arbitrary, as both edges will yield the same result:

(w_i is the distance between the left and the right edge of the needle slice at the given position and y_i denotes the peak point position.)

$$t_{left} = \frac{1}{n}[\sum_{i=1}^{n} y_i + \frac{w_i}{2} - mx_i] = \frac{1}{n}[\sum_{i=1}^{n} y_i - mx_i] + \frac{1}{2n}\sum_{i=1}^{n} w_i = t + \frac{1}{2n}\sum_{i=1}^{n} w_i$$

And similarily for the other edge

$$t_{right} = t - \frac{1}{2n}\sum_{i=1}^{n} w_i$$

The distance is then given as

$$|t_1 - t_2| = |t - t \pm \frac{1}{2n}\sum_{i=1}^{n} w_i| = \frac{1}{2n}\sum_{i=1}^{n} w_i$$

Thus estimating the distance $|t_1 - t_2|$ between the lines is nothing else than the average distance between all edge points divided by 2.

4.3 Orientation

Obtaining the orientation requires a 3D result. As one has already calculated which points belong the the needle body, and those points are already in 3D space, a simple yet effective approach is to project those points once to the xy plane and once to the xz plane and calculate the regression in each plane (see Figure 4.7).

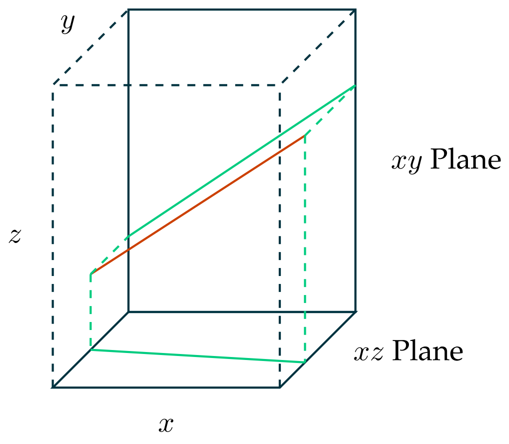

Figure 4.7: orientation projected to the two planes

Assuming the regression in the xy plane yielded

$$y = m_{xy}x + t_{xy}$$

and in the xz plane yielded

$$y = m_{xz}x + t_{xz}$$

then the orientation vector is simply given by $\vec{v} = -\begin{bmatrix} 1 \\ m_{xy} \\ m_{xz} \end{bmatrix}$

4.4 Control Point

As the control point is defined as the point in the middle of the needle where the diameter changes, just taking the peak point at the width change won't be enough. Calculating it however is quite simple after having already calculated the radius, peak point and orientation.

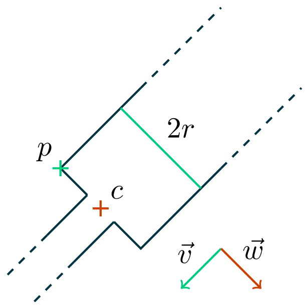

Figure 4.8: middle point of needle

Already calculated is $\vec{p}$, the peak point of the needle at the split between body and needle tip, as well as the radius r and the orientation vector $\vec{v}$. Obtaining $\vec{c}$ is then simply creating the to $\vec{v}$ perpendicular vector $\vec{w}$, scaling it by r and adding it to $\vec{p}$.

As the orientation vector $\vec{v}$ is in 3D, one has to specify in which plane $\vec{w}$ must lie, which is easy in this case, as the plane is simply created by the two vectors $\vec{v}$ and $\vec{d} = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}$

The normal of this plane is then $\vec{n} = \vec{d} \times \vec{v}$, around which $\vec{v}$ is then rotated by 90 degree. As the vector is perpendicular to the plane created by $\vec{v}$ and $\vec{n}$, just calculating the cross product $\vec{w} = \vec{v} \times \vec{n}$ yields the same result.

Finally $\vec{c}$ is given by

$$\vec{c} = \vec{p} + r\vec{w}$$

4.5 Putting it all together

The just described methods were translated into C++ code. Although it uses a conventional detection algorithm for the needle, the code is written as such, that one should be able to just plug in any other method of needle detection.

For regression a library called `mlpack` was used, which used `armadillo`, a linear algebra library internally. This was particularily useful, as the needle model had to be transformed, which was done using matrices. Having `armadillo` already available, the necessary matrix operations were already implemented.

For visualisation the Point Cloud Library was used.

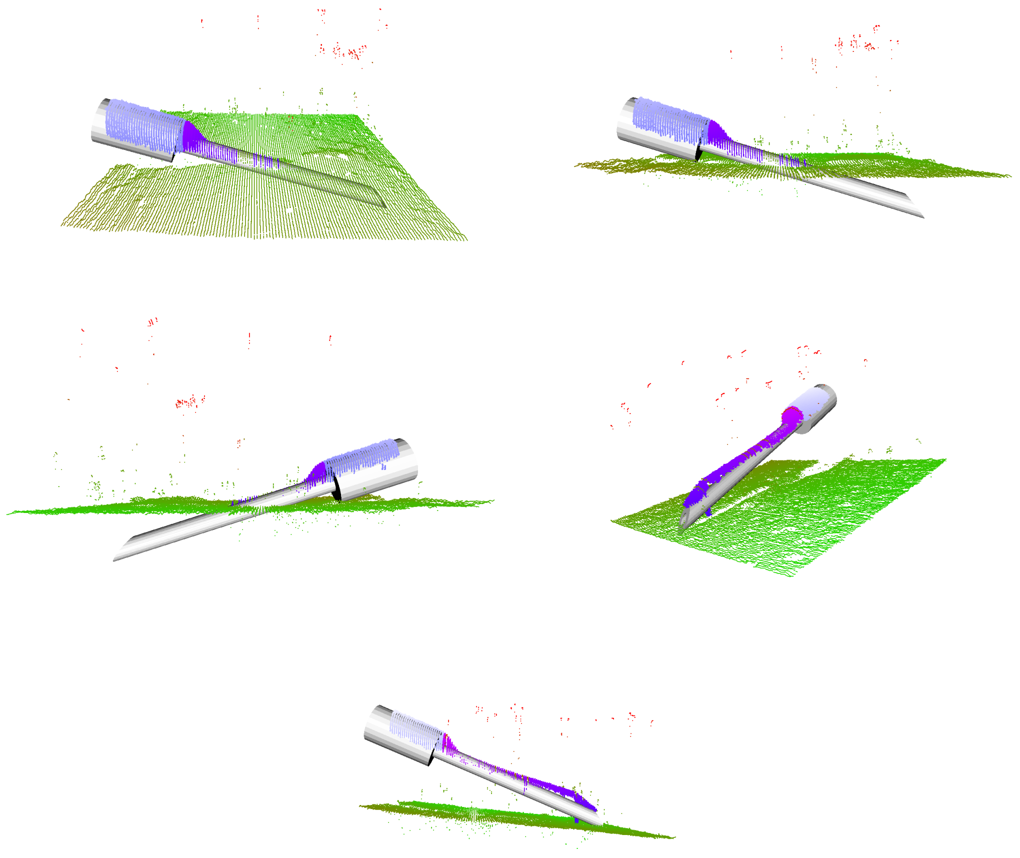

Figure 4.9: Visualization of reconstruction

The visualization contains the top layer of pixels from the filtered images seperated by whether the algorithm outputs it belongs to a needle or the retina (green for retina). It then reconstructs the needle with the described methods, loads and transforms the 3D model of the needle to the correct position and opens up an interactive window where the needle reconstruction can be seen.

5 Conclusion

Needle Detection based on a neural network was attempted in this thesis and the results given. Both methods deliver comparable results, but one has to choose which to use under which circumstances. If only the information whether there is a needle or not (classification) is needed, a neural network offers an algorithm with a high precision of $\geq 98\%$. If the width and peak point are important and missing needle slices can be ignored, then maybe the conventional algorithm is the better choice.

There is definetely room for improvement in both cases, but it's most likely easier with the conventional algorithm. As already stated, debugging or trying to improve the performance of a neural network is quite complicated. Furthermore the neural network may break in unpredictable ways, which could end catastrophically in the case of operations in a human eye. Completely detecting the needle using a CNN should probably only be done if the final network can be fully understood.

6 Future Work

Over the course of this thesis one particular problem became very apparent: An improvement of understanding what CNNs really do would be very helpful for the whole process. This includes a scientific solid ruleset of how to choose hyperparameters, layerparameters and a architecture for a given problem. This may very well be very hard, but is nevertheless important with anything involving neural networks. Furthermore, coming from a world of handwritten algorithms, a standard technique of debugging neural networks would be of advantage. There are advances in visualizing neural networks (see [10]), but making them easily available in caffee for example would most likely be helpful.

Besides the obvious problems with neural networks, the approaches are each relatively slow as they compute a lot of stuff on the CPU. Being an image processing problem this could probably be accelerated when fully implemented on the GPU. This would also save a lot of roundtrips needed when sending data between GPU and CPU.

Finally the approach with neural network described here used 2D convolution on the input slices enriched with some sequence information, but maybe full 3D convolution of the whole data cube could reach better performance for this given task.

Bibliography

[1] Ajith Abraham. Artificial neural networks. *handbook of measuring system design*, 2005.

[2] Robert Hecht-Nielsen et al. Theory of the backpropagation neural network. *Neural Networks*, 1(Supplement-1):445–448, 1988.

[3] Andrej Karpathy. Cs231n convolutional neural networks for visual recognition, 2016.

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

[5] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[6] Evelyn Regar, JA Schaar, E Mont, Renu Virmani, and PW Serruys. Optical coherence tomography. *Cardiovascular radiation medicine*, 4(4):198–204, 2003.

[7] Hessam Roodaki, Konstantinos Filippatos, Abouzar Eslami, and Nassir Navab. Introducing augmented reality to optical coherence tomography in ophthalmic microsurgery. In *Mixed and Augmented Reality (ISMAR), 2015 IEEE International Symposium on*, pages 1–6. IEEE, 2015.

[8] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[9] Joseph M Schmitt. Optical coherence tomography (oct): a review. *IEEE Journal of selected topics in quantum electronics*, 5(4):1205–1215, 1999.

[10] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.