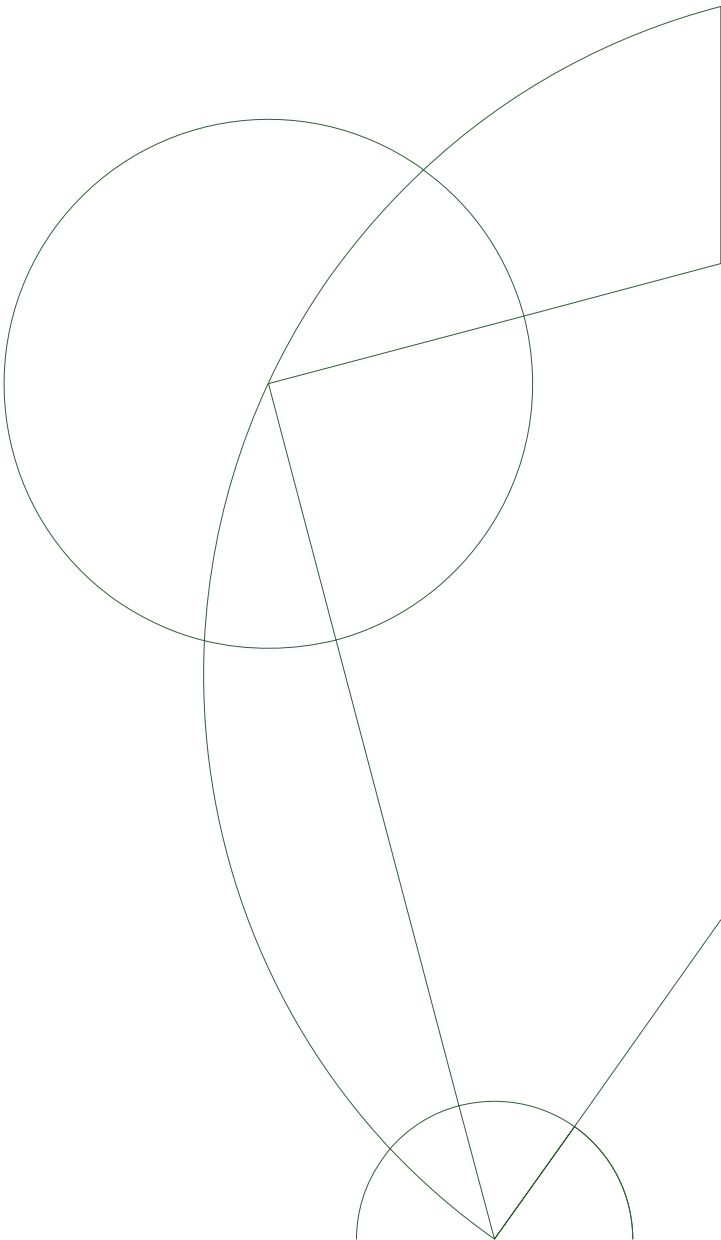
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Bachelor report

Jesper Henrichsen — TGW831

Vessel segmentation in retinal images

An implementation in Python using feed-forward neural networks via Shark library



Aasa Feragen & Christian Igel

June 13, 2016

Contents

List of Figures

List of Tables

1 Introduction

Image analysis is playing a larger and larger role in medicare. Therefore, research in image analysis constantly tries to push forward the reliability of state of the art image analysis techniques, so that doctors may confidently outsource part of their responsibility to computers. Having machines learn the statistical differences between healthy and disease on thousands of scans and screenings helps reduce the time from scanning to making the correct diagnosis. The potential for computers to assist doctors in diagnosing patients is an important reason for the increased focus on this research field in recent years.

There is evidence suggesting that analysing retinal blood vessels can aid in making early diagnosis of Alzheimer's disease [1]. Furtheremore, retinal blood vessels are used when determining the risk of diabetes patients to develop decreased visibility and in some cases blindness [4].
Another benefit of analysing retinal images are that they are comparatively cheap and non-invasive to make. It is therefore important to optimize segmentation processes for retinal images in a push to maximize the amount of valuable information that can be acquired from these images and the confidence that can be put into that information.

In this project the focus will be on a state of the art technique to segment blood vessel pixels in retinal images. The technique is developed by Marin et al [12] and was published in 2011. The purpose of this project is to make a program that implements this technique and to attempt to shed light on the possibility of vessel segmentation from the perspective of this technique.

2 Terms and concepts

This section gives a short explanation of terms and concept that are key to a better reading experience. Other important terms and concepts exists, but they are explained throughout the report when relevant. Important concepts, in the report, will be marked with *italic* the first time they are mentioned. Names of programs and images are marked with **bold** to indicate that their deeper understanding is either not important for the report or does not exist.

retina	Is a thin layer of tissue at the back of the eye. The retina is responsible for receiving the light that the lens is focused on and send it as neural signals to the brain for visual recognition [18].
segmentation	Segmentation is, from the perspective of this report, the process of dividing the pixels in the retinal images between those that represent blood vessels and those that do not.
FOV	*Field of view* (FOV) refer to the part of the retinal images which actually show the retina. Because of the natural round shape of the eye part of the image is outside FOV and therefore black. See Figure 1a.
foreground and *background*	Foreground is used to describe structures in the image that are represented by high gray scale intensity, making them bright. Background is everything else.

3 Implementation and design

The implementation of the general approach, as described by Marin et al. [12], is separated into two main areas. A preprocessing part in Python3 and a neural network part in C++.

The Python implementation is published under BSD open-source license and will be used in further research of this topic at The Image Section of Department of Computer Science at University of Copenhagen. For this reason, the essential parts of the implementation, at the time of writing, is provided in appendices A.3 to A.6.

The last phase, where a neural network is trained and tested, is implemented in C++ using the Shark library [8]. The implementation in C++ is not included since the many conducted experiments with the neural network training makes representation by a single version shallow, and it would therefore not add to the report. Also, for practical purposes, Shark provides a way of saving the trained model to a file so that the trained model can be used for later predictions. The Shark library itself is already available for public usage, and thus any future research benefit from the documentation in Shark itself rather than a deeper nested dependency.

3.1 Python classes

Handling preprocessing and feature extraction is done via a **Driver** class that can either be run interactively, where input is given through the terminal at run time, or by passing necessary arguments to the instantiation of the class, see appendix A.3 for the complete class implementation. A configuration file allows for the predetermination of certain input which can then be used across all instantiations. The configuration file, at the time of writing, is shown in appendix A.6. The driver will preprocess and extract features for either training scenario or general prediction. In the training scenario, a set of groundtruth images are necessary to compute the labels from. In the general prediction scenario, only a set of retinal images and a path to store the corresponding features as comma-separated values (CSV) files, are needed.
A **Preprocessing** class and **FeatureExtraction** class handles the preprocessing and feature extraction respectively. The implementation can be seen in appendices A.4 and A.5. The configuration file is also used to store the mean and standard deviation, which were used to normalize the dataset, when the driver is run for preparation of a training set. This allows normalization to be done with the same parameters for all future images. Python libraries such as Scikit-Image [19], SciPy [10], and OpenCV [2] were used for both preprocessing and feature extraction.

3.2 Shark and C++

The Shark Machine Learning library [8] provides a multitude of implemented algorithms when it comes to neural networks. The specifics about the neural network used in Marin et al [12] also left some parts open to interpretation, particularly considering the many possible compositions of the Shark methods and classes. A more thorough explanation of the chosen implementation will follow later in this report. The main focus of the C++ part of the program has been to train a neural network model, which is implemented in **neuralNet.cpp**.

3.3 Multiprocessed optimization

The time consumption of feature extraction and preprocessing was reduced in practice by the introduction of multiprocessing in the driver class. Running the preprocessing and feature

extraction of images in their own processes almost halved the time to process more than one image. The multiprocessing version takes advantage of the amount of CPU cores available on the host system. The measurement was made on a system running an Intel Core i5 CPU with 4 cores at 1.71GHz each and 8GB RAM by using the **time** command in linux[1], and by having the multiprocessed version available on a separate **git** branch[2]. Table 1 shows the issued command to process 20 images with the sequential version and the multiprocessed version respectively.

Table 1: Comparison of sequential and multiprocessed version

```
$ time python3 main.py  >> /dev/null &&
  git checkout -f multiprocessed &&
  time python3 main.py >> /dev/null

real    35m27.836s
user    35m25.977s
sys     0m1.945s
Checking out files: 100% (20/20), done.
Switched to branch 'multiprocessed'

real    18m47.968s
user    70m41.303s
sys     0m7.246s
```

As shown in Table 1 the real time of the sequential version is 35 minutes, while the same job is completed in 18 minutes in the multiprocessed version of the program. Equivalent to a 47% time reduction on this particular system.

4 Preprocessing of images

The image preprocessing is designed to attempt to increase contrasts between the background and foreground before the images are fed to the neural network. In the case of the retinal images, the contrasts which are sought to be increased are between vessel and nonvessel pixels. In this section, the steps of the preprocessing will be accompanied by the illustration of their results on images from the STARE database.

4.1 Extracting color band

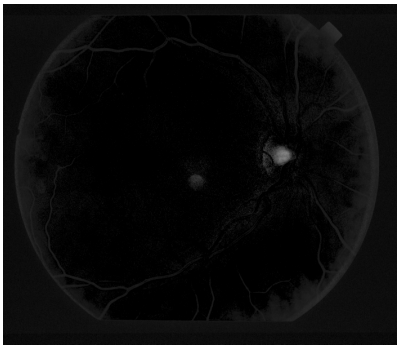
Each pixel in the image holds information about the intensity of red, green, and blue colors (RGB) which the individual pixel consists of. In Figure 1a is shown the original image IM0208 before any color band extractions. Figure 1b shows the extraction of the green color band. Figure 1 shows that the contrast between vessel pixels and nonvessel pixels is greatest within the green color band. Figures 1c and 1d shows the blue color band and the red color band respectively. The small amount of blue makes the gray scaling of the blue color band extraction seem dark, while the high value of red, in the retina pixels, make the gray scaled red color band extraction seem bright.

[1] **time** is a linux command that can print the execution time of another program.
[2] **git** is a free and open-source version control system. `https://git-scm.com/`

Figure 1: Illustration of the original image, im0208.ppm, in (a) and the different extracted color bands in (b-d) respectivly.

(a) Original image

(b) Green color band



(c) Blue color band

(d) Red color band

For these reasons the picture that was worked on during the preprocessing phase was the image with the green band extracted. The contrast between vessel and nonvessel pixels is highest in the green band image since the vessel pixels generally are darker than the background pixels. As seen in the image, the background pixels has more green in them which gives them a more yellow or orange color on the original image, while the vessel pixels mainly has red in them.

The choice of using only the green color band is consistent with Marin et al [12]. Using only the green color band may hold advantages in terms of speed during the preprocessing phase, even if using the full image in theory has the potential of providing greater detail to the neural network classifier.

The red band image was used, as seen in Figure 1d, to create a mask image of the pixels within the field of view. This was done using a threshold that exploited that the contrast between pixels in the field of view and pixels out of the field of view is much higher for the red band image. The mask that was

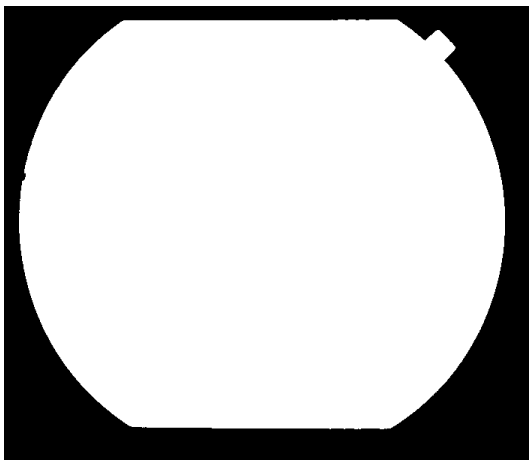


Figure 2: Illustrating the thresholded mask on im0208.

used with the image in Figure 1a can be seen in Figure 2.

4.2 Morphological gray opening

Gray opening is a form of opening applied to gray scaled images in image analysis. The term morphology originates from greek and means "study of shape". In the same sense, in gray opening a structuring element is used to evaluate a neighborhood of pixels for every pixel in the image. Morphological opening consists of a set simple ideas that when combinated can be very useful in putting certain shapes in the image into the background.

The structuring element that was used for this analysis was a 3×3 square, see equation (1). The reason for this step in the preprocessing was to remove a light reflex that occured along the center of some vessels [12]. An example of this can be seen in Figure 3, where the gray scaled image is shown before the grayscale opening filtering in Figure 3a and after in Figure 3b.

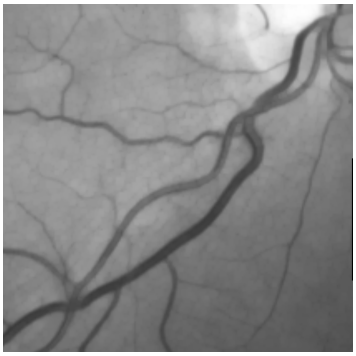
$$\text{Structuring element} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & \mathbf{1} & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{1}$$

The bold element in the matrix, seen in (1), marks the origin of the structuring element, where the origin is the pixel currently under consideration. The neighborhood is determined by the origin of the matrix, which in this case is all the surrounding pixels.

Morphological opening consists of *erosion* and *dilation* respectively and in that order. In grayscale erosion every pixel is set to the minimum value of the neighborhood pixels within the structuring element. Vice versa, in the dilation it is the maximum value that is used.

Figure 3: A comparison of the original central light reflex in retinal images and post gray opening by a 3×3 square

(a) Gray scaled image before morphological opening



(b) Gray scaled image post morphological opening

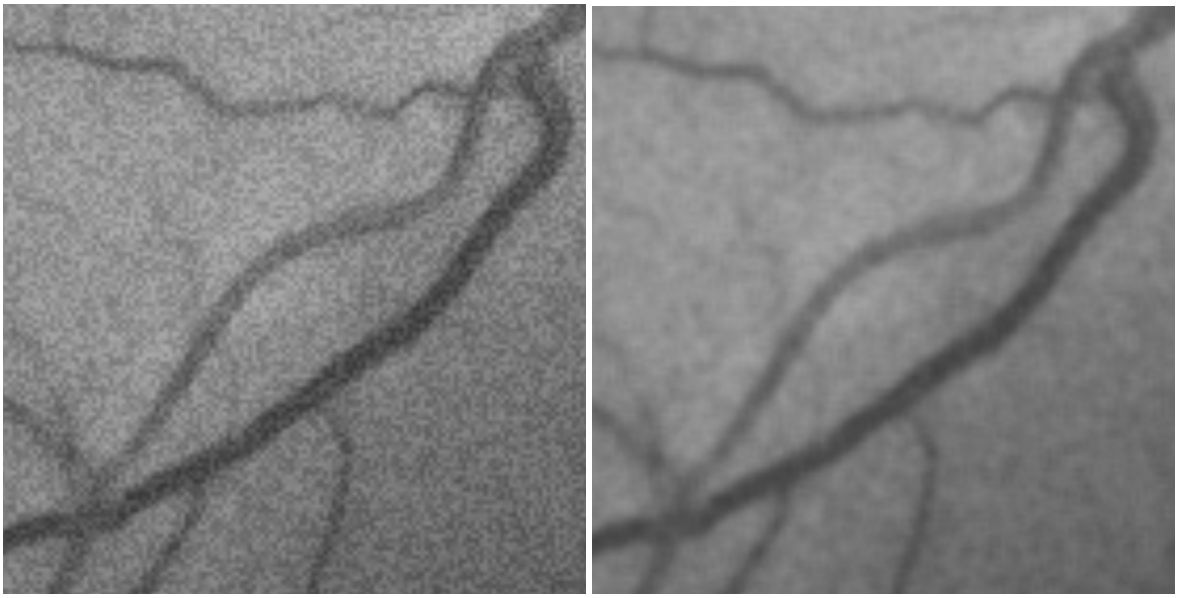
4.3 Filtering

In order to reduce noise in the images, a set of filtering methods are applied during the preprocess. This section will describe the filtering techniques that were used and how they were applied in the analysis.

4.3.1 Mean filter

Mean filtering is a technique where each pixel is set to the mean value of all the pixels in a specific area. That is, the mean where all pixels in that area has the same weight. The bigger the selected area is the more blurred the resulting image will become.

Figure 4: Image comparison between prior and post mean filtering



(a) Image prior to mean filtering **(b)** Image post mean filtering

The mean filter helps filter out *salt and pepper* noise. This notion covers pixels with much higher or much lower intensity than the surrounding pixels. They can potentially disrupt the classification step since there are no vessels that are represented as a single, none connected pixel in the image.

Figure 4 shows a comparison of before and after mean filtering is applied. Additional salt and pepper noise has been added in the images in order to better illustrate the effect of mean filtering. The images are zoomed in as well also to better see the effects of this filtering method. In Figure 4b the vessels in the image is slightly more blurred than in Figure 4a. However, when looking at the background in the two images it can be seen that the background is smoother after the mean filter is applied. A 3×3 filtering area was used in Figure 4b.

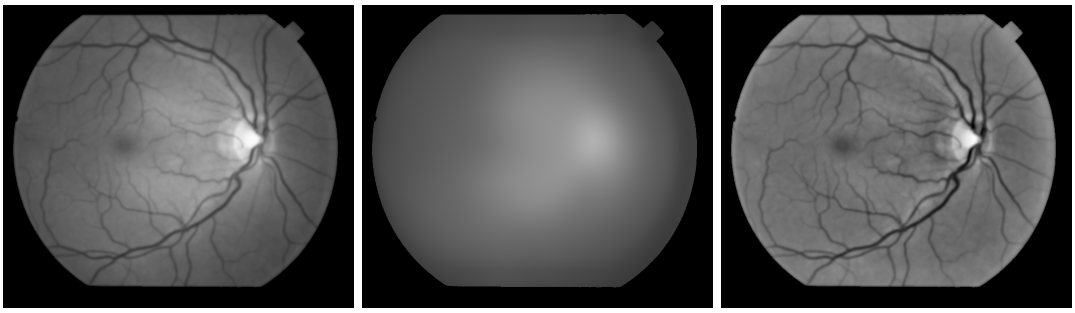
4.3.2 Gaussian filter

The gaussian filter technique uses properties from a gaussian distribution to set the intensity of each pixel. As in the case of mean filtering, the gaussian method that was used also compared each pixels with a specific area surrounding that pixel. With the gaussian filtering an area of 9×9 was used. In the gaussian filter the intensities of each pixel in the 9×9 square is weighted according to a 2 dimensional gaussian distribution. The gaussian distribution is curve shaped, or cone shaped in the 2 dimensional case. When setting the value of each pixel using this filter, the considered pixel is weighted as the center of the distribution. The shape is dependent on the variance of the distribution. In the case of this analysis, the parameters for the distribution was a variance of $\sigma^2 = 1.8$, as described in Marin et al. [12].
Thus the distribution is used as a point-spread function for the pixels in that area [17].

4.4 Background subtraction and linear transformation

A background image was computed by applying a mean filter of 69×69 to the image seen in Figure 5a. When applying this mean filter, the filter would be biased around the edges by the low intensity pixels outside the field of view area on the images. The mask, computed from the red color band of the original image, was used to overcome this problem. Thus the mask would have values of 1 for all pixels where the intensity of the red color whas above the threshold and 0 where it was below it. The mask was used to determine when the filter included pixels outside the field of view. In those cases, it would replace the outside field of view pixel intensities with the average pixel intensity from the pixels in the filter that was also inside the field of view.

Figure 5: Illustration of the background subtraction step and subsequent linear transformation of intensities



(a) Image prior to background subtraction

(b) Background image

(c) The resulting image after the background subtraction and subsequent linear transformation

The last image in Figure 5c shows the image after the background has been subtracted and the subsequent linear transformation. The linear transformation is necessary as some values might be negative after the subtraction, and the grayscale image only interprets values to be somewhere between 0 and 255. Thus it would also not be representative to show the image just after the subtraction, because how negative values are interpreted can be somewhat arbitrary.

Let I_S be the image matrix where the elements has the intensity values after the background subtraction then the transformed image matrix, I_L, is defined as

$$I_L = \frac{I_S - I_{S\min}}{I_{S\max} - I_{S\min}} \times \frac{1}{255} \tag{2}$$

The linear transformation in equation (2) is a mapping of numbers in one integer interval and into another. Therefore, each value is mapped according to its size in relation to the interval that it originates in and the one it is mapped to. In this case, it is mapped from a discrete range from the minimum value after the subtraction and up to the maximum value after the subtraction. The interval that it is mapped to has the grayscale image value range, which is $[0, \ldots, 255]$.

4.5 Transform intensity levels

For every pixel, p, in the linearly transformed image, I_L where $p = (x, y)\ \forall\ (x, y) \in I_L$, a new homogenized image, I_T, was computed. This was done by the following rules: at first

$$I_T(x, y) = I_L(x, y) + 128 - \text{mode}(I_L),$$

where $mode(I_L)$ is the gray level intensity value that is represented by the highest number of pixels in the I_L image. Afterwards, the values of I_T are rescaled into the gray level value range by the following cases:

$$gray\_level\_output = \begin{cases} 0 & \text{if } I_T(x, y) < 0 \\ 255 & \text{if } I_T(x, y) > 255 \\ I_T(x, y) & \text{otherwise,} \end{cases}$$

4.6 Vessel enhancement

The last step in the preprocessing phase was vessel enhancement using a *Top-Hat* transformation. Top-Hat transformation is defined as the difference between the image and the morphologically opened image. The transformation is based on first calculating the complementary image of the image resulting from the previous step, I_T, the complementary image will be denoted I_T^C:

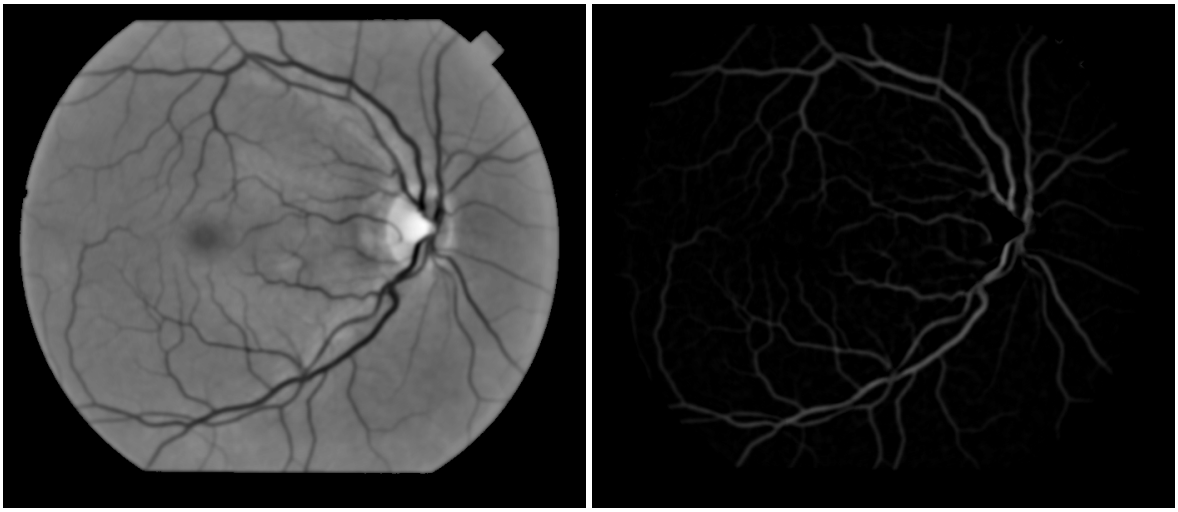
$$I_T^C = 255 - I_T,$$

Then the vessel enhanced image, I_{VE}, is created by

$$I_{VE} = I_T^C - \lambda(I_T^C),$$

where $\lambda(I_T^C)$ is the morphological opened complementary image of I_T. All the dark vessels will become bright by taking the complementary of the image, however, so will the background outside the field of view. In order to prevent brightening the background along with the vessels, the previously described mask was used to reset the outside FOV pixels to 0 intensity. The result is shown in Figure 6b.

Figure 6: Vessel enhancement on image im0208 from STARE database



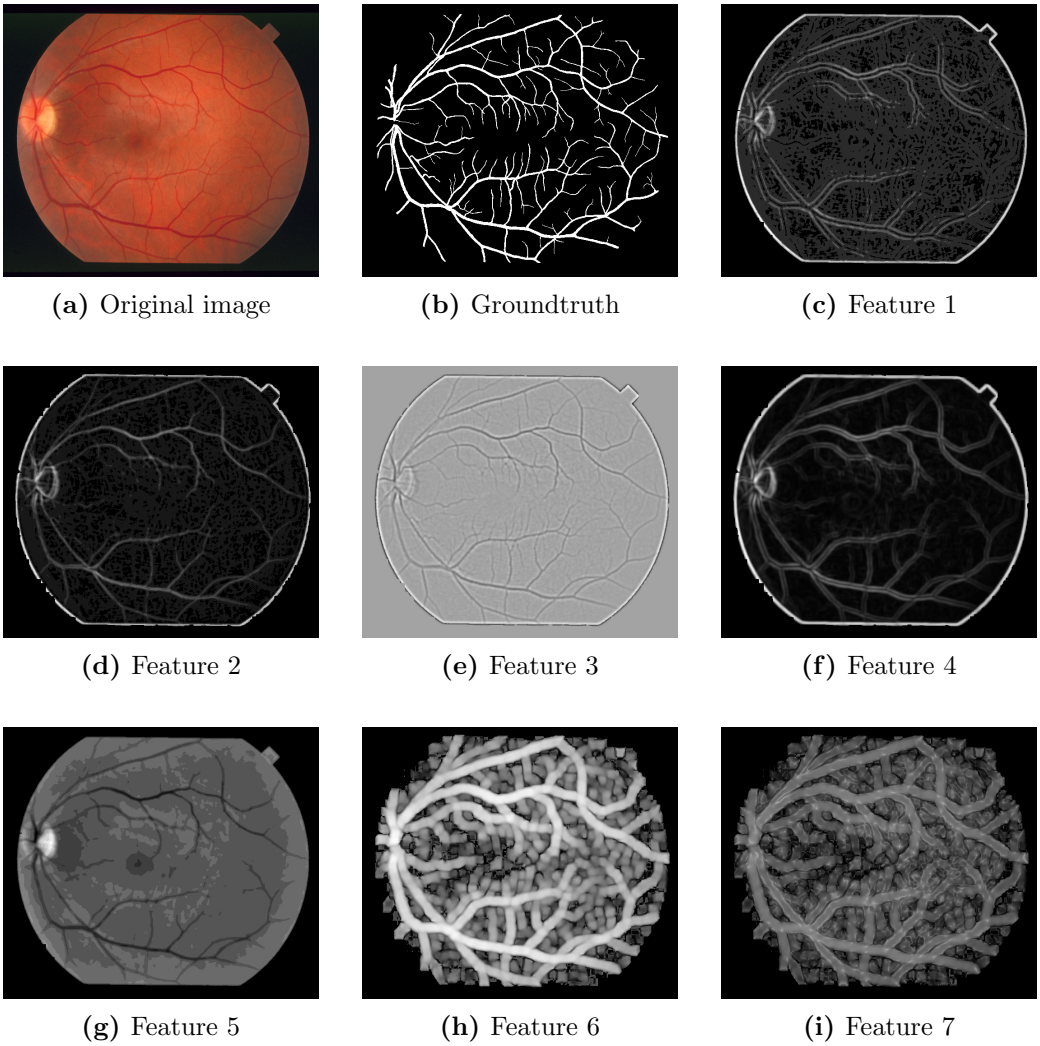
(a) The homogenized image, I_T, prior to top-hat transformation

(b) The image after top-hat transformation

5 Feature Extraction

The purpose of the feature extraction phase is to extract a set of numerical values which best describe the differences between the two classes that the classifier will decide between. In this regard, it is the difference between blood vessel pixels and non-blood vessel pixels that are of interest. This section will describe how a 7-dimensional feature set was extracted from the images acquired through the preprossesing step. The 7 features are all as documented in Marin et al [12]. See Figure 7 for a visualization of the extracted feature set. The visualization is over the features extracted for **im0255** from the STARE database [15]. The original image from the database and the manually segmented groundtruth image is seen in Figure 7a and Figure 7b respectively.

Figure 7: Visualisation of IM0255 from STARE database, its corresponding 7 features and the groundtruth image



(a) Original image **(b)** Groundtruth **(c)** Feature 1

(d) Feature 2 **(e)** Feature 3 **(f)** Feature 4

(g) Feature 5 **(h)** Feature 6 **(i)** Feature 7

In general the 7 dimensional feature space comprises two different types of features. The first 5 features are so called gray level based features and the second type are the first two Hu moment invariants. In the end, features are normalized to zero mean and unit variance by subtracting the mean, μ, and dividing by the standard deviation, σ. The normalization is done feature by feature.

5.1 Gray level based features

For all the gray level based features a neighbourhood of 9 pixels with the considered pixel in the center are included in the feature computation, with the exception of the fifth feature denoted by $f5$ in equation (7). The following are the gray level based features as described in Marin et al. [12]:

$$f_1(x,y) = I_T(x,y) - \min_{(s,t)\in S^9_{x,y}}\{I_T(s,t)\}, \tag{3}$$

$$f_2(x,y) = \max_{(s,t)\in S^9_{x,y}}\{I_T(s,t)\} - I_T(x,y), \tag{4}$$

$$f_3(x,y) = I_T(x,y) - \text{mean}_{(s,t)\in S^9_{x,y}}\{I_T(s,t)\}, \tag{5}$$

$$f_4(x,y) = \text{std}_{(s,t)\in S^9_{x,y}}\{I_T(s,t)\}, \tag{6}$$

$$f_5(x,y) = I_T(x,y), \tag{7}$$

For all gray level based features it is the gray level intensity in the homogenized image, acquired in the linear transformation step of the preprocessing phase denoted by I_T, that is used. Including a neighborhood surrounding the pixels may contribute to a more helpful pattern for the classifier, as true vesselpixels are always connected with other vesselpixels. The first four features are all measurements of the intensity in the considered pixel in relation to key statistical measures of the neighbourhood. That is, the intensity in relation to the neighbourhoods standard deviation, mean, maximum, and minimum.

5.2 Moment invariants based features

The last two extracted features extends on the idea of weighing the surrounding pixels when classifying a pixel as either vessel or nonvessel. In this case, by using moment invariants, which for this purpose are the first two Hu Moments named after Ming Kuei Hu who first published a paper on visual pattern recognition using moment invariants [7].

In order to recognize objects in images, different factors needs to be taken into account apart from general noise in the images. Specifically, rotation, translation, and scale are common ways that objects of the same class may vary. Therefore, the idea of invariants is to describe objects in an image by some measurable quantities that are insensitive to these variations of the same object. Mathematically, if the function f is the image function $f(x,y)$ and the function I is such an invariant and the function V is some degradation operations which either rotates, transform, or scales the objects described by $f(x,y)$, then I is an invariant if $I(f(x,y) = I(V(f(x,y)))$. In practice, however, general noise may mean that $I(f(x,y))$ only has to be reasonably close to $I(V(f(x,y)))$ [9]. This insensitivity is exactly what Hu moments provide. Hu was the first to derive a set of moment invariants that were invariant of not only scale and translation but also of rotation in images.

To better understand the underlying concept of Hu moments and moment invariants in

images, it is necessary to first consider moments. For an image function $f(x, y)$, the moment m_{pq} of order $p + q$, is the projection of $f(x, y)$ to the basis $x^p y^q$ such that

$$m_{pq} = \sum_x \sum_y x^p y^q f(x, y) \tag{8}$$

By extending n^{th} moment about the mean from statistics to 2D discrete sampling, image moments can be achieved. The n^{th} moment about the mean is described as

$$\mu_n = E[(X - E[X])^n] = \int_{-\infty}^{\infty} (x - \mu)^n \ f(x)dx \tag{9}$$

where X is a random variable and the expection of X, $E[X]$ is the average of X.

Since the projection consists of the linear combination

$$x_1^p \ y_1^q \ f(x_1, y_1) + x_1^p \ y_2^q \ f(x_1, y_2) +, ..., + x_m^p \ y_n^q f(x_m, y_n)$$

then $x^p y^q$ is a basis for m_{pq}. The moment m_{00} is just the number of white pixels in a binary image or an area of the image and the total intensity of an area in the case of this project where the moments were computed from gray scaled images. This is easily seen since

$$m_{00} = \sum_x \sum_y x^0 \ y^0 \ f(x, y) = \sum_x \sum_y f(x, y) \tag{10}$$

Similarly it can be seen that m_{10} sums over x and m_{01} sums over y. From moments m_{00}, m_{10}, m_{01} the centroid coordinates can be computed as follows:

$$\bar{x} = \frac{m_{10}}{m_{00}} \text{ and } \bar{y} = \frac{m_{01}}{m_{00}} \tag{11}$$

The centroids can be intuitively understood as the center of mass if the image was an object and the intensity of each pixel (x, y) described the weight of that object at point (x, y) then the centroid would be the point where the object could balance [5].

To extend equation 9 to 2D discrete sampling in order to be used in images, the centroid of the images can be treated as the mean, μ. For image analysis it can be written as

$$\mu_{pq} = \sum_x \sum_y (x - \bar{x})^p (y - \bar{y})^q f(x, y) \tag{12}$$

Hu defined a set of 7 moments which are named as Hu moments; however, in this report only the first 2 Hu moments will be extracted as part of the feature set. The first two Hu moments are defined as:

$$I_1 = \eta_{20} + \eta_{02}, \tag{13}$$

$$I_2 = (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2, \tag{14}$$

where η_{pq} are invariants with respect to scale and translation and are defined as

$$\eta_{pq} = \frac{\mu_{pq}}{\mu_{00}^{1 + \frac{p+q}{2}}} \tag{15}$$

and μ_{pq} are central moments [23].

5.2.1 Invariance

This section goes to show the concept of moment invariants by elaborating on how moment invariants are invariant to translation and scale.

From equation (12) it is seen how the central moments are defined with the centroids as their origin. Thereby, the moments in equation (15) are, by definition, invariant to translation since the centroid moves under translation [13, p. 16] [6].

To understand the concept behind scale invariance let $f'(x, y)$ be the image $f(x, y)$ scaled by λ. Such that $f'(x, y) = f(\frac{x}{\lambda}, \frac{y}{\lambda})$. Let $x' = \frac{x}{\lambda}$ and $y' = \frac{y}{\lambda}$ and $dx = \lambda dx'$, $dy = \lambda dy'$. Then the moment μ'_{pq} is given by

$$\mu'_{pq} = \int\int x^p y^q f(\frac{x}{\lambda}, \frac{y}{\lambda}) dxdy$$

by substitution this can be written as

$$\begin{aligned} \mu'_{pq} &= \int\int (\lambda x')^p (\lambda y')^q f(x', y') \lambda dx' \lambda dy' \\ &= \int\int x'^p y'^q f(x', y') \lambda^2 \lambda^p \lambda^q dx' dy' \\ &= \lambda^2 \lambda^p \lambda^q \int\int x'^p y' q f(x', y') dx' dy' \end{aligned} \tag{16}$$

The double integral in equation (16) is equal to μ_{pq}, thus equation (16) can be written as

$$\mu'_{pq} = \lambda^2 \lambda^p \lambda^q \mu_{pq},$$

and

$$\mu'_{00} = \lambda^2 \mu_{00},$$

and substituting these terms into equation (15) yields

$$\eta'_{pq} = \frac{\mu'_{pq}}{\mu'^{1+\frac{p+q}{2}}_{00}} = \frac{\lambda^{2+p+q} \mu_{pq}}{\lambda^2 \mu^{1+\frac{p+q}{2}}_{oo}} = \frac{\mu pq}{\mu^{1+\frac{p+q}{2}}_{00}} = \eta_{pq}$$

hence η_{pq} is scale invariant since the scale factor λ cancels out so that the scaled moment and nonscaled moment are equal.

Using the scale and translation invariance of the invariants, η_{pq}, and adding invariance with respect to rotation, is exactly what Hu moments provide. However, to show how Hu derived a set of invariants of rotation is more complex and thus outside the scope of this report, instead see Mukundan [13] and Hu [7] for papers on this topic.

5.3 Dazzling ring

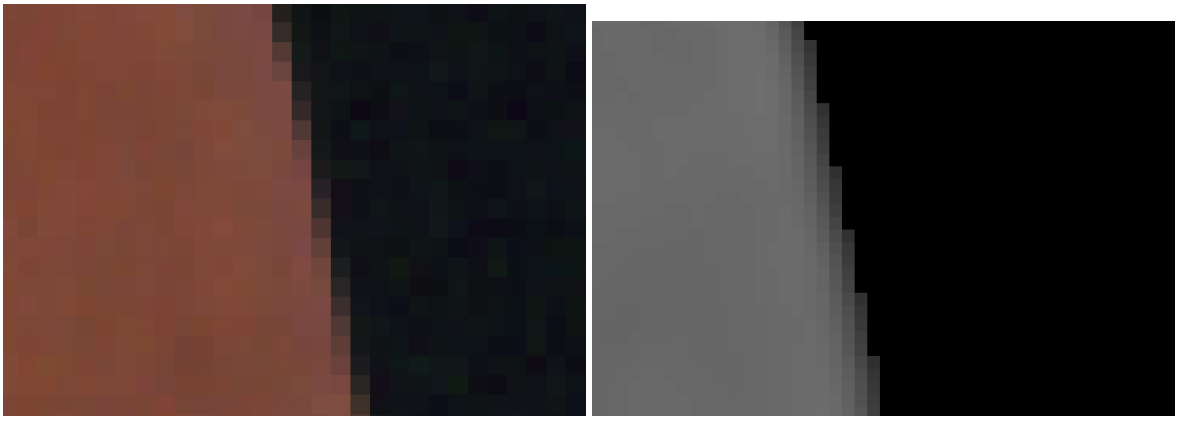
Notice from Figures 7c to 7f that a bright ring surrounds the FOV. This is an unwanted result that originates from the lower values in the neighbourhood of pixels at the border of the FOV in the homogenized image, I_T. To remove this it was necessary to first make sure that only non zero values where used, and secondly that only non zero values where included in the neighborhood. However, the bright ring is a result of gradual color dimming towards the edges of the homogenized image.
Finally, there are two approaches to remove the bright ring. It can either be removed prior

to feature extraction. Or it can be removed after the feature extraction.

Intuitively, it would be advantageous to remove the ring prior to feature extraction. To explore this, experiments were conducted with an eroded mask of size 6×6, 8×8, and 10×10 pixels. Though, only the square of size 10×10 sufficiently removed the ring on the training images. The eroded mask was then multiplied with the homogenized image immediately after the preprocessing phase.
However, as it turned out, the better results were acquired by removing the bright ring post feature extraction. At the end of this report in section 6.3, the effects of the neural network trained on the features, where the ring is removed post feature extraction, is shown in Table 2a. Similarly, the results from the neural network trained on the features, where the ring was removed on the homogenized image prior to feature extraction, is shown in appendix A.2.

This part of the feature extraction was not described in Marin et al. rather it was deemed necessary after discovering the bright ring upon visualizing the features in Figure 7. It is not possible to conclusively determine whether a bright ring occures on the features used in Marin et al. since these are not visualized in the paper. However, from Figure 8 it appears that the bright ring inherently would be a problem since, as Figure 8 shows, the gradual color dimming is present already on the original image.

Figure 8: Zoom on pixels along the edge on the original image and the homogenized image, I_T.



(a) The original image im0255.ppm. **(b)** Homogenized image of im0255.ppm

To see the effect of this ring on the final results, when left in the features, see appendix A.1. Appendix A.1 shows lower results on average for all measures compared to the results in Table 2a at the end of this report.

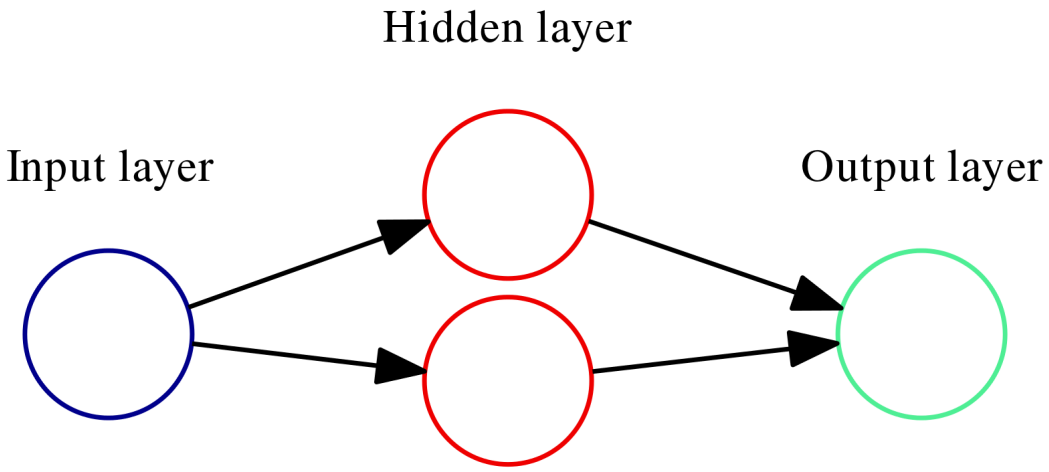
6 Classification

Classification is the process of assigning labels to each data entry in the data set. In this case, each entry represents a pixel and the data set are the features extracted from the retinal images. The classification is binary, either a 1 for a blood vessel pixel, or 0 otherwise. A great many algorithms exists to process such classification, and possibly just as many ways that they can each be tweaked and configured. Each with their own advantages. In this project, the choice of classification algorithm falls upon a feedforward neural network following the directions given in Marin et al [12].

6.1 Neural Networks

An *artificial feedforward neural network*, which for ease of notation will just be referred to as a *neural network*, is composed of layers of neurons. Generally, there are three types of layers: *input layer*, *output layer*, and *hidden layers*. The input layer refers to the layer containing the input neurons, which are the neurons that the input to the network is fed to. The output layer contains the neurons which the predicted class can be read from. The hidden layers refers to all other layers where the input is send to and from other layers in the network. See Figure 9 for an illustration of a very simple neural network structure with one input neuron, one output neuron, and one hidden layer with two neurons.

Figure 9: Simple example of a neural network topology



The edges in Figure 9, illustrating the output of one neuron being transfered to another, is an imitation of the synaptic connections in an organic neural network [3]. In an organic neural network the learning process is believed to happen by strengthening or weakening certain neural connections through the dendrites [20]. This learning process is imitated by the adjustment of weights and biases of the artificial neural network in the training phase. The training phase is analogous to a teacher with an answer book introducing a new subject to a class. In the beginning, there might be many wrong answers but through the teachers feedback the students gradually obtain a better intuition about the subject and may get to a point where they can answer new questions based on their experience.

6.1.1 Activation function

The neurons in the neural network are functions called *activation functions*. The activation function can be thought of as a decision maker that decides whether the input given to it warrants a signal to the next layer in the network. In the simple case, a neurons activation function may only decide between sending 0 or 1 to the next layer in the network, such a neuron is called a *perceptron*. In the case of a perceptron, the computation that is done in the activation function is held up against a threshold. If the value of the function is greater than this threshold, then it may send 1 to the neurons of the next layer and 0 otherwise.

In the continous case, the signal is not just 0 or 1 but lies in the range $[0, 1]$, thus the

value that is send from the neuron to the next layer can be thought of as the neuron's confidence that a certain feature is present. In this regard, a feature may be one of the features that were fed to the network, or it may be any other abstract feature that the network has learned to look for. The threshold in the continous case is replaced by a bias that is applied separately to each layer.

There are many possible choices for an activation function, however, there are some properties that a good activation function should have in order for the network to best be able to compute nontrivial problems. For reasons that will be explored in the next section of this report, then the activation function should be differentiable. Secondly, it should be nonlinear. Otherwise, if the function is linear, then the whole network, regardless of the number of layers and neurons, will essentially be linear itself.

A typical choice for an activation function, and also the chosen activation function in this project, is a *sigmoid*. The sigmoid is defined as

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{17}$$

The sigmoid therefore holds the properties that are mentioned above but it also conveniently reflect the input that is given. If z is large then $e^{-z} \approx 0$ and $\sigma(z) \approx 1$. Reversely, if z is small then $e^{-z} \to \infty$ and $\sigma(z) \approx 0$ [14]. This behaviour can be intuively seen from the graph of the sigmoid function in Figure 10.

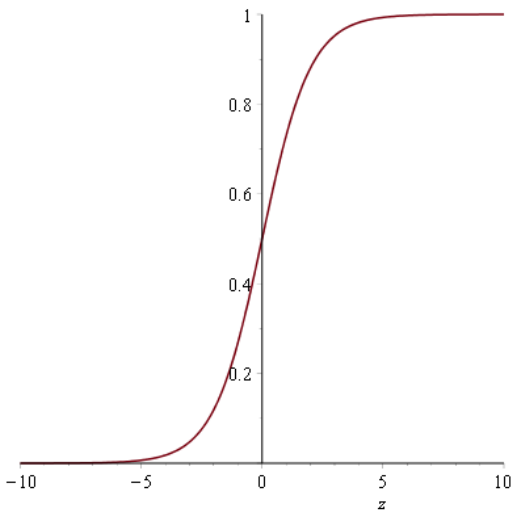


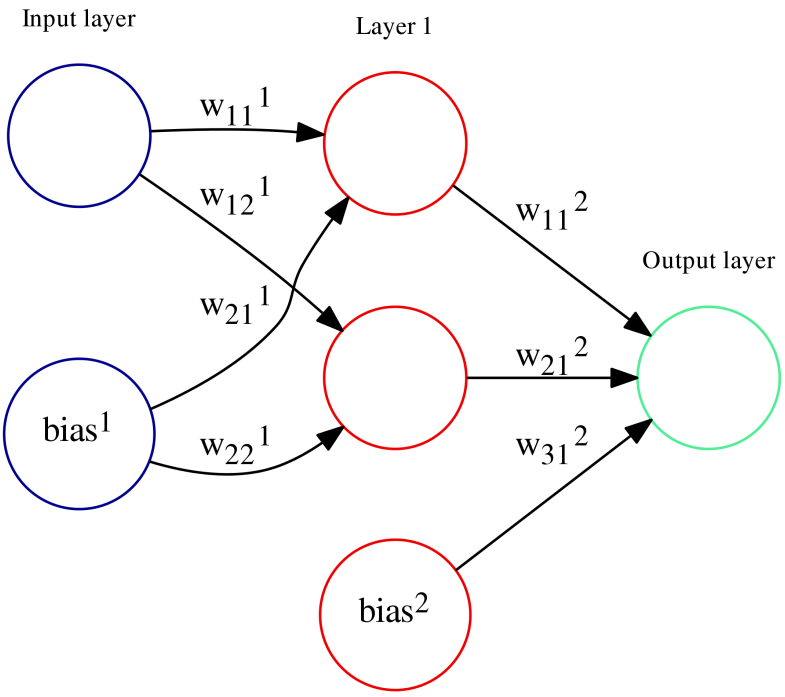
Figure 10: Plot of a sigmoid function

The input z for the activation function is the weighted sum of output from the activation functions in the layer below. This is reflected in Figure 11. From Figure 11 it can also be seen that each layer adds a bias to the weighted sum. Therefore, the input z is given by

$$\text{z= } w^l_{1k}a^{l-1}_1 + w^l_{2k}a^{l-1}_2 +, ..., + w^l_{jk}a^{l-1}_j + b^l$$

where w^l_{jk} is the connection from the jth neuron in layer $l-1$ to the kth neuron in layer l and a^l_k is the kth activation function in layer l. The bias in layer l is just denoted b^l, as there is only one bias for each layer.

Figure 11: Simple neural network example with biases



6.1.2 Cost function

A method called *gradient descent* is applied in order for the network to learn. However, in order to use the *gradient descent* method, a proxy measure, or surrogate, of the error function is used. The proxy measure is called a *cost function* or sometimes a *loss function*. The reason for this, is that the cost function provides a smooth function of the weights and biases in the network [14]. The cost function is lower bounded, most commonly by 0, thus to optimize the classification on the training set is to minimize the cost function so that it converges towards the lower bound.

The cost function that was used in this project was the *cross-entropy* loss function. The cross-entropy loss function is positive and lower bounded by 0 and is defined as

$$CE = -\frac{1}{n}\sum_x y\ln(a) + (1-y)\ln(1-a), \tag{18}$$

where n is the number of entries in the training set and x denotes each input from that set and y the desired output, the labels in the training set. The differentiation of the cross-entropy with respect to the weights and the biases can be shown to be

$$\frac{\partial CE}{\partial w_{jk}} = \frac{\partial CE}{\partial b_l} = \frac{1}{n}\sum_x \frac{\sigma'(x)x_j}{\sigma(z)(1-\sigma(z))}(\sigma(z)-y), \tag{19}$$

which is shown by Nielsen [14].

The cross-entropy method has an advantage over many other loss functions since its derivative can be expressed independently of the derivative of the sigmoid function $\sigma(z)$, which, as will be explored closer in the next section, prevents a slowdown in the learning rate when the value of the sigmoid is high, indicating a high error. This is achieved by seeing that the

derivative of the sigmoid can be expressed as

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)) \Rightarrow \frac{e^{-z}}{(1 + e^{-z})^2} = \frac{1 - \frac{1}{1+e^{-z}}}{1 + e^{-z}} \tag{20}$$

since the numerator of the right hand side of the last equation is

$$1 - \frac{1}{1 + e^{-z}} = \frac{e^{-z}}{1 + e^{-z}}.$$

The expression on the right hand side, in equation (20), becomes that of the left hand side. Going back to equation (19) it can be seen that the denominator cancels the expression in the numerator and equation (19) can be written as

$$\frac{\partial CE}{\partial w_j k} = \frac{\partial CE}{b_l} = \frac{1}{n} \sum_x x_j(\sigma(z) - y) \tag{21}$$

Thereby, it can be seen from equation (21) that the derivative of the cross-entropy is independent of the derivative of the sigmoid, ensuring that the learning rate does not slow down as a result of the derivative of the sigmoid being close to 0 [14] [21].

6.1.3 Gradient descent

Now that a cost function is established, with a well defined derivative, it is time to explore the meanings of the *gradient descent*. In broad terms, the gradient descent is, as the name indicates, a method to descend to the minimum of a function through the gradient. Computing the gradient, ∇CE, of the cost function, CE, with respect to the weights and biases in layer l, will yield a vector that can be interpreted as pointing in the direction where the cost function grows the most. That is, if a positive step, η, is taken in the direction of the gradient, e.g. $\eta \nabla CE$, it would be expected that $CE(\eta \nabla CE)$ produces a greater value than before.
In the case of the neural network, such greater value indicates a higher error than before such a step. However, if the same step were to be taken in the opposite direction, the value would be expected to decrease, meaning a lower error rate for the neural network. Therefore, a rule to update the vector of weights and biases, v, can be formed

$$v \rightarrow v - \eta \nabla CE$$

The above should be interpreted as setting the value of v to the values of v subtracted by the step in the opposite direction of the gradient. Thereby, the weights and bias would be descending to a minimum of the cost function CE [14].

6.1.4 Backpropagation

Backpropagation is an algorithm that effectively implements the gradient descent in a neural network by partially differentiating the layers in the network through the chain rule, starting from the output layer. For each entry in the training set that is given as input, the feed-forward phase of the neural network training will produce an output in all the activation functions of the network. Afterwards, the backpropagation algorithm allows for the partial derivatives of the cost function, with respect to the weights and biases, to back propagate.
The cost can be computed in the training phase, as the learning is supervised entailing that the labels are known. However, the label classes, when computing the cost function, are to be considered constants since they cannot be changed. Instead the weights and biases in the network may be altered. To this extend the backpropagation will be run for every training

element as the underlying function is unknown, and so a change in the weights and biases are only made by a small step at a time. This implies a descending to a local minima of the cost function over the training set.

Using the chain rule, the derivative of the error of the last layer, L, can be computed as

$$error^L = \frac{\partial CE}{\partial a_j^L}\sigma'(z_j^L)$$

By going backwards to the previous layers from layer L, the derivative of the error in layers $l \in \{L-1, L-2, ..., 1\}$ can again be found using the chain rule and is given by

$$error^l = ((w^{l+1})^T error^{l+1}) \odot \sigma'(z^l)$$

where the $\odot$ marks the elementwise multiplication of elements in vectors v and w so that $v \odot w = v_j w_j$. Thereby, the backpropagation can be seen to compute the gradient of the loss function by propagating the derivative of the error backwards through the network [14]. This gradient is then what is fed to the gradient descent optimizer function, which can then update the weights, as stated in the previous section 6.1.3.

Modes of learning Backpropagation provides three modes of learning, *online learning*, *bactch learning*, and *stochastic learning* [21]. In the online learning mode, the weights and biases are updated per entry in the dataset so that each time a gradient is computed, after a forward pass, the weights in the network will be updated. In the batch learning mode, the gradient may be computed over the whole dataset before the optimizer updates the weights and biases with the average gradient. The stochastic learning computes the gradient from randomly chosen entries in the dataset in an attempt to speed up the learning process for very large training sets and networks. For this project batch learning was chosen, as it generally has the best approach in reaching a minimum of the cost function without getting stuck, thus it is usually preferred when possible and affordable [21].

RProp The specific backpropagation algorithm chosen for this project was the *resilient backpropagation*, RProp, which is implemented in Shark [8]. RProp is a batch update algorithm that adapts the learning step during the training. A set of rules dictates the learning step based on wether the sign of the derivative changed from the last iteration [16] [24]. The learning step itself is therefore not a parameter in the design of this project. A more elaborate explanation of RProp will not be included in this report, instead see Riedmiller and Braun [16].

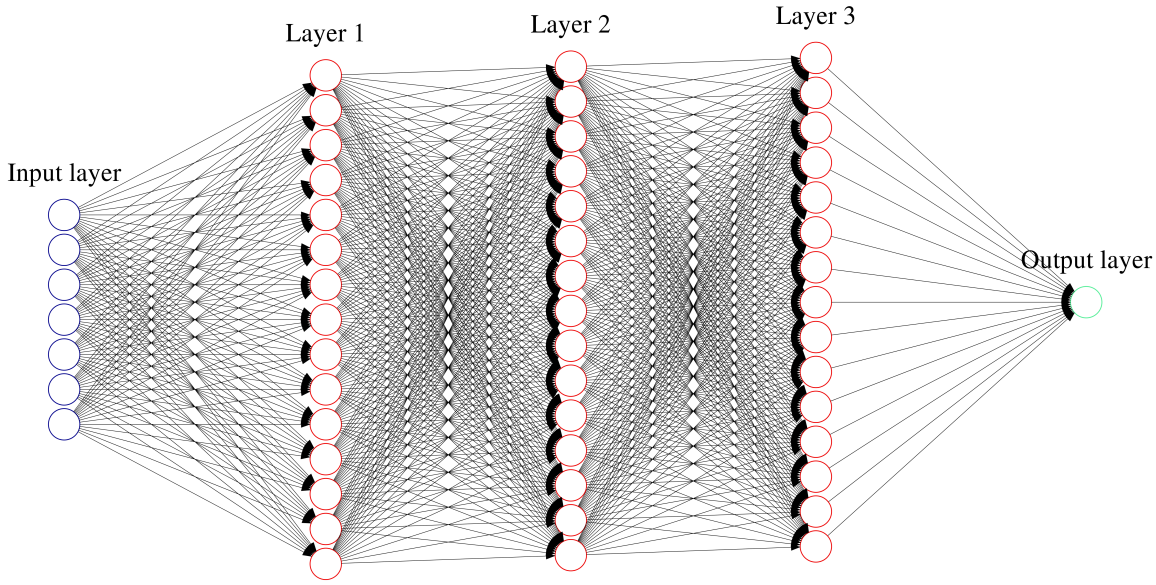
6.2 Training

A training set of 20 images from the STARE database [15] was used in the training phase. These images had been manually segmented by an ophthalmologist so that a groundtruth for the correct labels existed for the images. This is necessary since the neural network introduced in this project is, as described in the previous section 6.1, a supervised learning algorithm. This means the images were classified as any other image in the forward phase and then the cost for the backpropagation phase were computed using the groundtruth images. In this context, a white pixel meant label 1, a vessel pixel, and a black pixel corresponded to a non vessel pixel.

The topology of the neural network is as described in Marin et al [12] and is depicted in Figure 12. Notice that the biases and weights have been omitted in order to simplify the

illustration. However, the actual network did include weights and biases in line with how they were described in the previous section.

Figure 12: Neural network topology



In Figure 12 it can be seen that there is an input neuron for each of the 7 features that are fed to the network. The 3 columns of red colored neurons are the hidden layers. In this topology there are 3 hidden layers with 15 neurons in each. Since the classification is binary there is only need for one output neuron. Despite the many edges it can be infered from Figure 12 how all neurons in a layer feed to all neurons in the next layer exclusively.

The purpose of the training phase is to iteratively train a model on already known images in order to acquire a generalizable understanding of the classification problem. To this extend, the known images are the set of manually labeled groundtruth images, which should be large enough for the neural net to learn patterns that are generally different between different label classes across the population, rather than just different for the examples in training set.

It is not a trivial task to acquire a model that generalizes well over yet unseen data from training on a training set and much depends on the particular dataset. Therefore, some experimentation was done in order to settle on methodologies for the training phase.

6.2.1 Cross-validation

Where the aim of the backpropagation algorithm is to minimize the error on the training set, then the purpose of the *cross-validation* technique is to prevent overfitting the model to the training set. That is, in the process of learning the training set the model will start following the patterns in the training set close as to minimize error, but this also risks training the model to follow the training set too close. If this happens, *overfitting* has occured and the model will generalize badly to classify yet unseen data. Which, after all, is the overall goal of training the model.

Cross-validation seeks to randomly divide the training set into separate folds of equal size before the training begins. In this project, the training set was divided into 5 folds. However, in order to prevent features from the same images to spill into more than one fold, then the randomized draw was done at the level of the CSV file. Since feature sets from each image

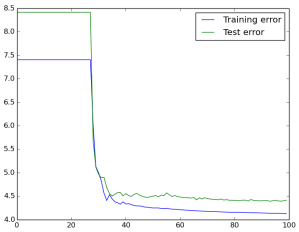
are saved in separate CSV files, then the easiest way of randomizing the folds, while ensuring that features from one image are not present in more than one fold, is to shufle the order of CSV file names before the features are loaded into Shark [8]. The Fisher-Yates method to draw randomly without replacement was implemented as a helper function to shuffle the set of CSV file names [11] [22].

The cross-validation then meant that the neural network was trained once for each fold. Each time, a new fold was selected as a test set and put aside. The remaining folds were combined into one set which the training was run on. After each trained model, the classification error was determined on the selected test set. In Figure 13 the error for each fold is plotted for a cross-validation training with 100 iterations.

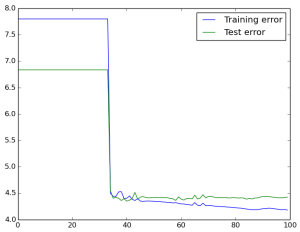
Figure 13: Plot of the training and test error over 5 folds cross validation with 100 iterations

Fold	Error	Acc
1	4.41%	95.59%
2	4.43%	95.57%
3	4.55%	95.46%
4	4.00%	96.00 %
5	3.94%	96.06 %

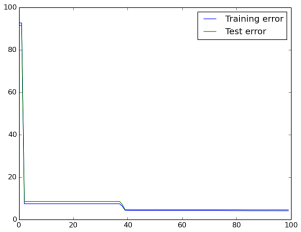
(a) Accuracy over the 5 folded testsets shows an accuracy of 95.65% ± 0.2%



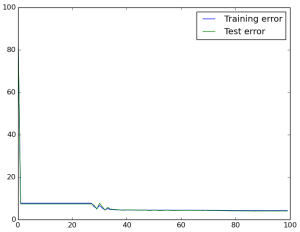
(b) First fold test error was 3.84%



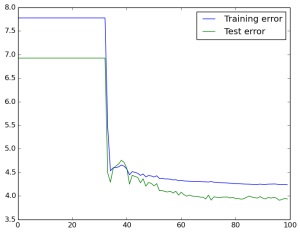
(c) Second fold test error was 4.81%



(d) Third fold test error was 4.84%



(e) Fourth fold test error was 3.86%



(f) Fifth fold test error was 4.01%

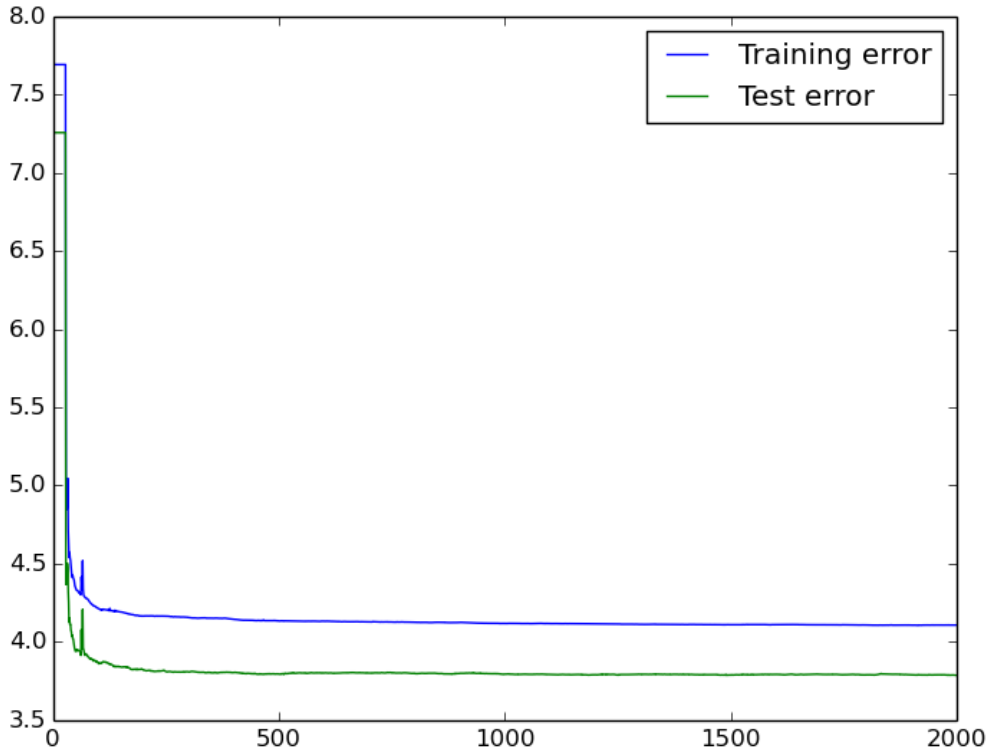
6.2.2 Early stopping

One drawback of using a fixed number of iterations in the training phase is that the model may start to overfit the training set if the neural net is trained for too long. Thus the training length itself is susceptible to overfitting. To prevent this, a method called *early stopping* is often used in order to cut short the learning process if overfitting is occuring. To detect overfitting, the training and test error is monitored to determine if overfitting has begun. Intuitively, overfitting in this case would show as a point where as the training error minimizes the test error slowly starts to go upwards again. In the plots from Figure 13 it is equivalent to one of the green lines starting to crawl upwards again as the iterations continues, though it is not the case in any of these plots.

Early stopping can be done together with cross-validation by selecting a validation set in addition to the training and test set. The training will then stop as soon as the error on the validation set starts to increase.
However, in order to determine wether the training set and the model were subsceptible to

overfiting, experiments with longer iterations were conducted. In this regard, the error was plotted from the training of a neural network model with 2000 iterations. The error can be seen in Figure 14.

Figure 14: Neural network model trained with 2000 iterations



As Figure 14 shows, the learning rate slows down and the error is close to constant already when 500 iterations is reached. As Figure 14 indicates, the neural network training would not benefit from early stopping since there is no upward slope in the plot, and the training could be shortened by just keeping a fixed amount of iterations.

For this reason, early stopping was chosen not to be included, as the need to avoid overfitting the neural network model did not seem present. Instead, a fixed number of iterations were used. In most cases, 100 iterations were used in a tradeoff between execution time and error minimization.

6.2.3 Balanced training set

Another pitfall is to introduce a bias towards the learning error on non blood vessels. The equally weighted error may bias the learning towards non blood vessels, since only a small proportion of the pixels in retina images are blood vessels. The reason is that only $\sim 10\%$ of the pixels represent blood vessels. Therefore, a small improvement on the error on non blood vessels can yield a much higher overall accuracy than the same numerical improvement on errors made on blood vessel pixels. It was therefore of interest to see if the performance of the neural network could improve if the errors were treated equally.

Overall, there are two ways to do this, the first is to weight the specific errors during the

training phase by weighting errors on non vessel pixels by $\frac{1}{10}$. The second method, and the one which was chosen for this project due to the simplicity of implementing the method, was to train on a balanced training set. The balanced training set consists of equal amounts of non blood vessels and blood vessel pixels. In practice, the non blood vessel entries were chosen randomly and limited to the amount of blood vessel pixels in the image.

However, the results of these tests showed a lower performance of neural network predictions on full images when trained on balanced training sets. For this reason, the measured results of trained neural network models in the next section, does not include this method either.

6.3 Testing

The testing phase has been closely related to the training phase, as the testing has been performed on a fold selected as test set after each training phase. However, this section will also include similar measurements as Marin et al [12] in order to easily compare the results in this report with Marin et al.

Besides the overall accuracy of the neural network, Marin et al [12] also uses measures for *sensitivity* (Se), *specifity* (SP), *positive predictive value* (Ppv), and *negative predictive value* (Npv). They are defined as

$$Se = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}, \tag{22}$$

$$Sp = \frac{\text{True Negatives}}{\text{True Negatives} + \text{False Positives}}, \tag{23}$$

$$Ppv = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}, \tag{24}$$

$$Npv = \frac{\text{True Negatives}}{\text{True Negatives} + \text{False Negatives}}. \tag{25}$$

Sensitivity indicates the ratio between the number of true positives predicted by the neural network compared with the total amount of blood vessel pixels in the image. Similarly, the specifity expresses the ratio between the number of true negative values predicted and the total number of negative values. Since the images are composed of mostly negative values, a smaller deviation in the specifity is expected. On the other hand, prediction errors will easilier manifest itself in the sensitivity measurement.
The positive predictive value indicates the ratio between true positive predictions out of all positive predictions. Conversely, the negative predictive value indicates the ratio between true negative predictions out of all negative predictions.

The overall accuracy is the ratio of correct predictions over the total amount of pixels in the image.
Common for all these measurements is the observation that the closer to 1 the better the performance.

Table 2a shows the results of a trained network for each of the images in the STARE database. Each image was used as a fold and thus a test was conducted once per image. The remaining

images were combined to one training set which the network was trained on for 100 iterations. The average result per measure is shown in the last row of the table. For comparison, the results from Marin et al. [12] is shown in Table 2b.

Table 2: Performance results on STARE database compared with Marin et al. The **bold** marks the highest and lowest values for each column.

(a) Performance results from this project.

Image	Se	Sp	Ppv	Npv	Acc
im0001.ppm	0.4945	0.9873	0.7583	0.9603	0.9505
im0002.ppm	0.4588	0.9852	0.6832	0.9631	0.9509
im0003.ppm	0.6678	0.9803	0.6722	0.9799	0.9625
im0004.ppm	**0.3369**	0.9974	0.9085	**0.9510**	0.9499
im0005.ppm	0.6337	0.9794	0.7489	0.9650	**0.9488**
im0044.ppm	0.6220	0.9878	0.7903	0.9725	0.9626
im0077.ppm	0.7944	0.9844	0.8043	0.9834	0.9702
im0081.ppm	**0.8786**	**0.9666**	**0.6666**	**0.9905**	0.9603
im0082.ppm	0.6933	0.9903	0.8510	0.9759	0.9684
im0139.ppm	0.6564	0.9829	0.7620	0.9716	0.9577
im0162.ppm	0.7342	0.9819	0.7516	0.9802	0.9646
im0163.ppm	0.8489	0.9784	0.7619	0.9876	0.9686
im0235.ppm	0.7093	0.9867	0.8307	0.9735	0.9632
im0236.ppm	0.6795	0.9890	0.8555	0.9699	0.9620
im0239.ppm	0.5550	0.9933	0.8797	0.9619	0.9577
im0240.ppm	0.6379	0.9858	0.8240	0.9630	0.9529
im0255.ppm	0.6528	0.9916	0.8801	0.9678	0.9622
im0291.ppm	0.4382	**0.9987**	**0.9476**	0.9712	0.9707
im0319.ppm	0.4206	0.9977	0.8806	0.9767	**0.9748**
im0324.ppm	0.4294	0.9945	0.8462	0.9611	0.9573
Average	0.6171	0.9870	0.8052	0.9713	0.9608

(b) Performance results from Marin et al. [12].

Image	Se	Sp	Ppv	Npv	Acc
im0001.ppm	0.5997	0.9844	0.8245	0.9527	0.9425
im0002.ppm	0.5074	0.9931	0.8798	0.9527	0.9489
im0003.ppm	0.6534	0.9892	0.8422	0.9699	0.9619
im0004.ppm	**0.4159**	0.9953	0.9094	**0.9379**	0.9365
im0005.ppm	0.5884	0.9857	0.8508	0.9452	**0.9372**
im0044.ppm	0.7958	0.9712	0.7250	0.9803	0.9559
im0077.ppm	0.8183	0.9771	0.8142	0.9772	0.9593
im0081.ppm	0.8682	**0.9673**	0.7512	0.9848	0.9572
im0082.ppm	0.7729	0.9818	0.8355	0.9731	0.9595
im0139.ppm	0.6670	0.9809	0.8110	0.9601	0.9467
im0162.ppm	0.8109	0.9724	0.7601	0.9795	0.9567
im0163.ppm	**0.8781**	0.9675	0.7603	**0.9854**	0.9581
im0235.ppm	0.7796	0.9770	0.8246	0.9697	0.9530
im0236.ppm	0.7765	0.9787	0.8374	0.9687	0.9537
im0239.ppm	0.6910	0.9850	0.8598	0.9599	0.9504
im0240.ppm	0.6802	0.9825	0.8639	0.9497	0.9402
im0255.ppm	0.7039	0.9882	0.8926	0.9599	0.9534
im0291.ppm	0.5840	**0.9961**	**0.9172**	0.9698	0.9675
im0319.ppm	0.6776	0.9872	0.7694	0.9799	**0.9689**
im0324.ppm	0.6225	0.9763	**0.7245**	0.9628	0.9441
Average	0.6944	0.9819	0.8227	0.9659	0.9526

6.3.1 Unbiased threshold

In Marin et al. [6], one optimal threshold was found for the STARE database as a whole and used for the reported results. This optimal threshold reportedly provided the highest results. However, the threshold itself was found while training on all images at once, thus introducing a dependency between the reported results and the optimal threshold. For these reasons, and for the purpose of reproducability of the results in this report outside of the label images in the STARE database, a fixed threshold of 0 was used for the output neuron. Alternatively, an optimal threshold could have been computed from the training set alone and used for each image, but to reduce computation time this option was not chosen for this project. It is therefore an important distinction between the results from this project and the results from Marin et al. [12].

The average sensitivity score of 0.6171 from this report, is lower than the equivalent measure from Marin et al. which was 0.6944, see Table 2b. However, Table 2a shows a higher average score for specifity and overall accuracy which may indicate a mere shift in the distribution of labels. Which, again, could be indicative of the benefit of the customized threshold used in Marin et al. Thus the difference may only come down to a difference in the distribution of labels dictated by the threshold.

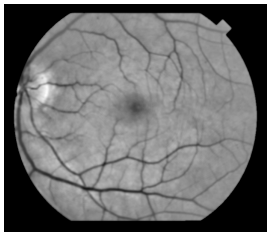
In any case, because the test images were allowed to bias the threshold in Marin et al. then it is inconclusive whether their results accurately represents the general performance of their neural network. To this extend, it is interesting to see, from Table 2, that the performance results in this report matches up to the results in Marin et al.

6.3.2 Visual representation

The sensitivity says the most about the impression of the predicted image because of the images sparse population of vessel pixels. In Figures 15b and 15c is seen the visualization of

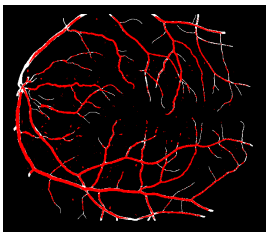
the predicted image for the image with the highest sensitivity, **im0081.ppm**. The predictions are marked with red on top of the original image and the manually segmented image respectively. Similarly, in Figures 15c and 15e is seen the predicted image which resulted in the lowest sensitivity score, **im0004.ppm**. To get a better idea of what the network had to work with, the homogenized image is shown in Figures 15a and 15d respectively. From Figure 15d the round shape of the retinal is distorted, this is due to the mask having registered part of the image as outside the FOV. Consequently, this particular image has had lower intensity levels than 50, on the gray scale range, in this part of the FOV. This part of the image might have survived if the threshold used to create the masked was lowered, but it is doubtful that anything meaningful would have been recovered from this area regardless.

Figure 15: Predicted blood vessel pixels marked with red overlayed groundtruth and original image. Showing the images with the highest and lowest results for sensitivity.

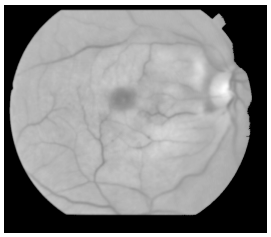


(a) The homogenized image for im0081.ppm

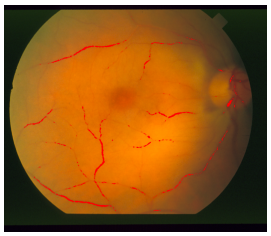
(b) Predicted blood vessels for im0081.ppm put on top of the original image.



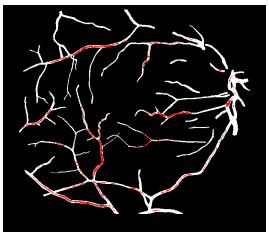
(c) Predicted blood vessels for im0081.ppm on top of the groundtruth image.



(d) The homogenized image for im0004.ppm



(e) Predicted blood vessels for im0004.ppm overlayed the original image.



(f) Predicted blood vessels for im0004.ppm on top of the groundtruth image.

7 Conclusion

In terms of results, there is a clear indication that on average confidence can certainly be put into automatically segmented images. Furthermore, the results of this report provides support for the methodologies described in Marin et al [12]. The neural network is fully capable of making classifications. Although, it is not possible to leave the segmentation entirely up to the neural network, as proved by Figure 15f, but there is still great potential for machines to assist during this process. Even if human involvement is ultimately needed, then something is still won if that involvement can be reduced by the help of neural networks or other classifiers.

But, how well is good enough? There is no way to know how well predictions can be done. In this sense, the optimization of machine learning algorithms deployed in this field of research is a continuous exploration into the possibilities of enhancing hospital staff with more tools for the job. In the end, humans and machine learning alike are susceptible to making mistakes. However, the combined efforts may allow for quicker decisions with higher confidence in their decisiveness.

The combined efforts of humans and machines is, in the field of vessel segmentation, exactly what pushes forward the frontier of state of the art methods. Just as with the very principal of the neural network which seeks to find a minimum of the error function. So too do the researchers in their experiments seek to find the set of methodologies and classification algorithms with the best chosen paramaters, which minimizes the false predictions on retinal images. A commonly given analogy is that of mountain climbers in darkness trying to get down from a mountain. For whatever reason, the mountain climbers have only a flashlight with limited battery, so they will have to carefully choose when they use it. However, not able to see the route in its entirety they will need to use the flashlights at certain points to orient themselves of a direction.
The mountain in this case is used as analogy for the error function, which is sought to be minimized, and the distance between points when they use the flashlight is similar to the learning rate. However, instead of being low on battery the learning rate increases computation time the lower it is set. A similar climb is done by the ongoing research in this field, which as well seeks error minimization but on a much larger scale.

To this extend, it is relevant to consider what might change about the current results if the training was done with weighted error to balance the disproportion of vessel to non vessel pixels. Or, if the amount of neurons were to be increased or decreased. Does the network benefit from all 7 features, and if so, could it benefit from even more?

In the end, to conclude anything in absolute terms would, even if the results are promising, be misrepresenting the experiments conducted in this project and machine learning as a whole. If anything, this project goes to show that the possible twists and tweaks of the segmentation process are, if not endless, then practically so. In this regard, this project only explored a very limited variety of possibilities. However, the results can serve as an important benchmark for changes to the current configuration in this corner of the possibility spectrum, to see whether they influence the network to perform better. Thereby joining the combined efforts to descend from the mountain, just like the neurons in the network.

References

[1] Fatmire Berisha, Gilbert T. Feke, Clement L. Trempe, J. Wallace McMeel, and Charles L. Schepens. Retinal Abnormalities in Early Alzheimer's Disease. *Investigative Ophthalmology & Visual Science (IOVS)*, 48, May 2007.

[2] G. Bradski. *Dr. Dobb's Journal of Software Tools*.

[3] Dan Cloer. The Synaptic Connection. `http://www.vision.org/visionmedia/article.aspx%3Fid%3D321`, 2004. [Online; accessed 12-May-2016].

[4] Josè Cunha-Vaz. Predicting Progression of Diabetic Retinopathy with the Retmarker. *Retina Today*, 2009.

[5] Jan Flusser, Barbara ZitovÃą, and TomÃąÅą Suk. *Moments and Moment Invariants in Pattern Recognition*. Wiley, 2009.

[6] Guido Gerig. Lecture: Shape Analysis Moment Invariants. `http://www.sci.utah.edu/~gerig/CS7960-S2010/handouts/CS7960-AdvImProc-MomentInvariants.pdf`. [Online; accessed 01-May-2016].

[7] Ming Kuei Hu. Visual Pattern Recognition by Moment Invariants. *IRE Transactions on Information Theory*, IT-8:179–187, 1962.

[8] Christian Igel, Verena Heidrich-Meisner, and Tobias Glasmachers. Shark. *Journal of Machine Learning Research*, 9:993–996, 2008.

[9] Jan Flusser Barbara Zitova Tomas Suk. *Moments and Moment Invariants in Pattern Recognition*. John Wiley & Sons, 2009.

[10] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 27-May-2016].

[11] Donald Knuth. *The Art of Computer Programming*, volume 2. Reading, Mass : Addison-Wesley, 2nd edition, 1969.

[12] Diego Marìn, Arturo Aquino, Manuel Emilio Gegùndez-Arias, and Josè Manuel Bravo. A new supervised method for blood vessel segmentation in retinal images by using gray-level and moment invariants-based features, 2011.

[13] R. Mukundan and K.R. Ramakrishnan. *Moment Functions in Image Analysis: Theory and Applications*. World Scientific, 1998.

[14] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[15] STARE ProjectWebsite. STructured Analysis of the REtina, STARE. `http://www.ces.clemson.edu/~ahoover/stare/`, 2013. [Online; accessed 21-May-2016].

[16] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. *IEEE Press*, pages 586–591, 1993.

[17] Robert Fisher, Simon Perkins, Ashley Walker and Erik Wolfart. Gaussian smoothing. `http://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm`, 2003. [Online; accessed 29-February-2016].

[18] Healthline Medical Team. Retina. `http://www.healthline.com/human-body-maps/retina`, 2015. [Online; accessed 30-May-2016].

[19] Stefan van der Walt. Scikit-image: image processig in Python. http://scikit-image.org/, 2016. [Online; accessed 27-May-2016].

[20] Wikipedia. Artificial neuron — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Artificial_neuron, 2016. [Online; accessed 14-May-2016].

[21] Wikipedia. Backpropagation — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Backpropagation, 2016. [Online; accessed 18-May-2016].

[22] Wikipedia. Fisher-Yates shuffle — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle, 2016. [Online; accessed 11-May-2016].

[23] Wikipedia. Image moments — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Image_moment, 2016. [Online; accessed 08-April-2016].

[24] Wikipedia. Rprop — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Rprop, 2016. [Online; accessed 1-June-2016].

A Appendices

A.1 Results on features with bright ring

Table 3: Performance results on STARE database features that includes bright ring seen in Figures 7c to 7f

Image	**Se**	**Sp**	**Ppv**	**Npv**	**Acc**
im0001.ppm	0.4421	0.9886	0.7703	0.9533	**0.9450**
im0002.ppm	0.4738	0.9820	0.6531	0.9632	0.9482
im0003.ppm	0.6509	0.9763	**0.6365**	0.9777	0.9569
im0004.ppm	**0.2808**	0.9982	0.9256	**0.9455**	0.9450
im0005.ppm	0.6526	**0.9755**	0.7260	0.9658	0.9464
im0044.ppm	0.5839	0.9878	0.7812	0.9694	0.9596
im0077.ppm	0.8037	0.9776	0.7574	0.9828	0.9636
im0081.ppm	**0.8464**	0.9638	0.6537	0.9873	0.9550
im0082.ppm	0.6810	0.9875	0.8228	0.9732	0.9634
im0139.ppm	0.6221	0.9830	0.7615	0.9675	0.9539
im0162.ppm	0.7157	0.9812	0.7453	0.9782	0.9623
im0163.ppm	0.8246	0.9768	0.7485	**0.9852**	0.9650
im0235.ppm	0.6774	0.9878	0.8441	0.9691	0.9601
im0236.ppm	0.6205	0.9906	0.8684	0.9633	0.9571
im0239.ppm	0.4993	0.9948	0.9014	0.9546	0.9520
im0240.ppm	0.6029	0.9856	0.8261	0.9562	0.9465
im0255.ppm	0.6502	0.9905	0.8703	0.9665	0.9600
im0291.ppm	0.4300	**0.9986**	**0.9423**	0.9705	0.9698
im0319.ppm	0.3805	0.9975	0.8737	0.9728	**0.9709**
im0324.ppm	0.4531	0.9934	0.8299	0.9621	0.9573
Average	0.5946	0.9859	0.7969	0.9682	0.9569

A.2 Results on features with bright ring removed prior to feature extraction.

Table 4: Performance results on STARE database features where the bright ring from Figures 7c to 7f has been removed prior to feature extraction.

Image	**Se**	**Sp**	**Ppv**	**Npv**	**Acc**
im0235.ppm	0.7057	0.9854	0.8254	0.9716	0.9605
im0240.ppm	0.6530	0.9902	0.8681	0.9667	0.9601
im0291.ppm	0.4331	0.9963	0.8420	0.9750	**0.9721**
im0236.ppm	0.6941	0.9893	0.8473	0.9743	0.9661
im0002.ppm	0.5866	0.9864	0.7331	0.9740	0.9625
im0001.ppm	**0.8441**	0.9661	**0.6679**	**0.9872**	0.9570
im0324.ppm	0.8308	**0.9782**	0.7618	0.9857	0.9668
im0004.ppm	0.6241	0.9807	0.7621	0.9633	0.9484
im0239.ppm	0.6989	0.9822	0.7504	0.9770	0.9620
im0255.ppm	0.6553	0.9795	0.7369	0.9701	0.9535
im0319.ppm	0.7304	0.9863	0.8232	0.9767	0.9658
im0082.ppm	0.5972	0.9865	0.8345	0.9556	0.9468
im0003.ppm	0.6179	0.9852	0.7570	0.9718	0.9596
im0081.ppm	0.4305	0.9988	**0.9514**	0.9705	0.9701
im0163.ppm	0.7047	0.9866	0.8402	0.9711	0.9611
im0005.ppm	0.5018	0.9830	0.6780	0.9651	0.9509
im0162.ppm	0.4718	0.9865	0.7516	0.9556	**0.9454**
im0139.ppm	0.5088	0.9905	0.7932	0.9658	0.9584
im0077.ppm	**0.2930**	**0.9981**	0.9255	0.9463	0.9458
im0044.ppm	0.4823	0.9947	0.8961	**0.9531**	0.9505
Average	0.6032	0.9865	0.8023	0.9688	0.9582

A.3 Driver class

```
"""
    Driver is the main driver class to run image preprocessing and feature
    extraction with necessary image paths and helper classes etc.

    Example run
    ```
        python3 driver.py
    ```
"""
import os, numpy, glob, re
from os import path
from PIL import Image
from preprocessing import Preprocess
from featureExtraction import FeatureExtraction
import json
from dictionary import text

"""
    The driver class is the entry point for initializing image preprocessing and
    subsequent feature extraction via the FeatureExtraction class and Preprocess
    class.

    The driver can be used interactively by just issuing
    ```
        python3 main.py
    ```
    or it can be included in another program by importing the driver class and
    presetting required arguments via the configuration file.

    The default configuration file is `./configurations.json` where standard
    JSON format is applied. The configuration file is also used by the driver
    object to store the mean and standard deviation from the training set. This
    way mean and standard deviation can be used to normalize the training set
    and any future images.

    The configuration is seperated into two JSON objects where the first
    `generalSettings` consists of user defined settings as follows:
        @param `DestinationFolder` {string} The folder path for saving the
            computed features.
        @param `ImagePath` {string} The folder where the images are located
        @param `LabelPath` {string} The folder where the groundtruth images are
            stored used as labels
        @param `extract_balanced` {bool} Wether to extract balanced feature sets
            of 50/50 class 0 and class 1
        @param `override_params` {bool} If true, any stored mean and standard
            deviation will be overwritten by the computed mean and standard
            deviation from this set of extracted features.

    The other JSON object is the `paramaters` object which is used by the driver
    object to read and write mean and standard deviation vectors to.

```

```
    @class Driver
"""

class Driver:
    # initializers
    def __init__(
            self,
            ImagePath=None,
            LabelPath=None,
            DestinationFolder=None,
            mode=None
        ):
        self.configfile = "configurations.json"
        self.agree = ['y', "yes"]
        self.extract_balanced_notset = False
        self.override_params_notset  = False
        ImagePath, LabelPath, DestinationFolder = self._getConfiguration()
        self._setDestinationFolder(DestinationFolder)
        if self._runTrainOrTestMode(mode):
            self._setImagePath(ImagePath)
            self._setLabelPath(LabelPath)

        else:
            self._setImagePath(ImagePath)

    def _getConfiguration(self,
        ImagePath=None,
        LabelPath=None,
        DestinationFolder=None):
        with open(self.configfile, 'r') as f:
            confs = json.load(f)

        gsets = confs['generalSettings']
        if (gsets['ImagePath'] and not ImagePath):
            ImagePath = gsets['ImagePath']
        if (gsets['LabelPath'] and not LabelPath):
            LabelPath = gsets['LabelPath']
        if (gsets['DestinationFolder'] and not DestinationFolder):
            DestinationFolder = gsets['DestinationFolder']
        try:
            self.extract_balanced = gsets['extract_balanced']
        except KeyError:
            self.extract_balanced_notset = True
        try:
            self.override_params  = gsets['override_params']
        except KeyError:
            self.override_params_notset = True

        # returns ImagePath, LabelPath and DestinationFolder in order
        # for any passed arguments to have precedence. And to reuse
        # the same path validation functions.
        return [ImagePath, LabelPath, DestinationFolder]
```

```
104
105
106     def _runTrainOrTestMode(self, mode):
107         if mode:
108             self.mode = mode
109             return mode == "training"
110         print(text['train_or_test'])
111
112         if input(text['choose_mode']).lower() in self.agree:
113             self.mode = "training"
114             return True
115         else:
116             self.mode = "testing"
117             return False
118
119     def _setDestinationFolder(self, path=None, insist=True):
120         destPath = path or input(text['choose_dest_folder'])
121         if os.path.exists(destPath):
122             self.destinationPath = destPath
123         else:
124             print(text["dest_folder"], destPath, text["no_path"])
125             if (insist):
126                 self._setDestinationFolder()
127
128     def _setImagePath(self, path=None, insist=True):
129         ImagePath = path or input(text['type_image_path'])
130         if os.path.exists(ImagePath):
131             self.ImagePath = ImagePath
132         else:
133             print(text['image_folder'], text["no_path"])
134             if (insist):
135                 self._setImagePath()
136
137     def _setLabelPath(self, path=None, insist=True):
138         LabelPath = path or input(text['type_label_path'])
139         if os.path.exists(LabelPath):
140             self.LabelPath = LabelPath
141         else:
142             print(text["label_folder"], text["no_path"])
143             if (insist):
144                 self._setLabelPath()
145
146     def _getStoredMeanStd(self, confs, override=False):
147         if override:
148             confs["parameters"]["featureMeans"] = self.mean.tolist()
149             confs["parameters"]["featureStd"]   = self.std.tolist()
150             with open(self.configfile, 'w') as f:
151                 json.dump(
152                     confs,
153                     f,
154                     sort_keys=True,
155                     indent=4,
```

```
                    separators=(',', ': ')
                )

            mean    = numpy.array(confs["parameters"]["featureMeans"])
            std     = numpy.array(confs["parameters"]["featureStd"])
            return [mean, std]


        """
            @method run
            @param `extract_balanced` if set to `True` the extracted feature set
            will include an equal amount of nonvessel and vessel pixels.
        """
        def run(self):
            print(text["run_mode"].format(self.mode))
            if self.mode == "training":
                labels = [f for f in os.listdir(self.LabelPath)
                    if path.isfile(path.join(self.LabelPath, f))]
                if not len(labels):
                    raise FileNotFoundError(
                        2,
                        "No labels found in labelpath",
                        self.LabelPath
                    )

                images = []
                for label in labels:
                    name = re.search("^im[0-9]{4}", label)
                    img = name.group(0) + ".ppm"
                    images.append(img)
                if len(labels) == len(images) or not len(labels):
                    print(text["disp_label_dir"], self.LabelPath)
                    print(text["disp_image_dir"], self.ImagePath)
                    print(text["disp_dest_dir"], self.destinationPath)
                    print(text["disp_conf_file"], self.configfile)
                    meanVectors = numpy.empty((len(labels), 7))
                    varVectors  = numpy.empty((len(labels), 7))
                    feObj = []
                    for i, x in enumerate(images):
                        print("\nImage ", i+1, " of ", len(images))
                        fe = FeatureExtraction(
                            image=path.join(self.ImagePath, images[i]),
                            GTPath=path.join(self.LabelPath, labels[i])
                        )
                        fe.computeFeatures().normalize(comp_only=True)
                        # save mean and standard deviation from training:
                        feObj.append(fe)
                        meanVectors[i]  = fe.mean_vector
                        varVectors[i]   = fe.var_vector
                    self.mean    = meanVectors.mean(axis=0)
                    self.std     = numpy.sqrt(varVectors.mean(axis=0))

```

```
208                 with open(self.configfile, 'r') as f:
209                     confs = json.load(f)
210 
211                 if (confs["parameters"]["featureMeans"]):
212                     override = (self.override_params_notset or
213                         self.override_params)
214                     if self.override_params_notset:
215                         if input(text["write_mean_std"]).lower() in self.agree:
216                             self.mean, self.std = self._getStoredMeanStd(
217                                 confs,
218                                 True
219                             )
220                         else:
221                             self.mean, self.std = self._getStoredMeanStd(confs)
222                     elif self.override_params:
223                         self.mean, self.std = self._getStoredMeanStd(
224                             confs,
225                             True
226                         )
227                     else:
228                         self.mean, self.std = self._getStoredMeanStd(confs)
229 
230                 ## Normalize data with mean of all training image features
231                 ## and write features to csv files
232                 for i, fe in enumerate(feObj):
233                     print(
234                         "Normalizing ",
235                         fe.source,
236                         " and writing features to ",
237                         images[i] + ".csv"
238                     )
239                     fe.normalize(
240                         mean=self.mean,
241                         std=self.std
242                     ).exportCSV(
243                         filename=path.join(
244                             self.destinationPath,
245                             images[i] + ".csv"
246                         ),
247                         balanced=self.extract_balanced
248                     )
249             else:
250                 print("Some label names are not as exptected")
251                 print("Names are expected to start with '^im[0-9]{4}'")
252                 print("e.g, im0001.ah.ppm")
253         else:
254             images = [f for f in os.listdir(self.ImagePath)
255                 if path.isfile(path.join(self.ImagePath, f))]
256             print(text["disp_image_dir"], self.ImagePath)
257             print(text["disp_dest_dir"], self.destinationPath)
258             print(text["disp_conf_file"], self.configfile)
259             with open(self.configfile, 'r') as f:
```

```
                confs = json.load(f)
            if (confs["parameters"]["featureMeans"] and
                confs["parameters"]["featureStd"]):

                mean    =  numpy.array(confs["parameters"]["featureMeans"])
                std     =  numpy.array(confs["parameters"]["featureStd"])

            else:

                print(
                    "No mean and standard deviation exists in ",
                    self.configfile
                )
                return

            for i, x in enumerate(images):
                print("\nImage ", i+1, " of ", len(images))
                fe = FeatureExtraction(
                    image=path.join(
                        self.ImagePath,
                        images[i]
                    )
                )
                fe.computeFeatures().normalize(
                    mean=mean,
                    std=std
                ).exportCSV(
                    filename=path.join(self.destinationPath, images[i] + ".csv")
                )
```

A.4 Preprocessing class

```
"""
    Example run
    ```
        python3 preprocessing.py
    ```
"""
from PIL import Image
from scipy import ndimage
from skimage.filters import rank
from skimage.morphology import square
from skimage.morphology import disk
from skimage.morphology import white_tophat
import numpy
import cv2
import matplotlib.pyplot as plt
import PIL
from unbuffered import Unbuffered
import sys
# make print() not wait on the same buffer as the
# def it exists in:
sys.stdout = Unbuffered(sys.stdout)
class Preprocess:
    """
        Preprocess class is responsible for anything preprocessing. It is build
        for easy convolution of the preprocessing operations. Such that
        operations may be easily followed by each other in any order by dotting
        them out like so:
        ```
            obj =   Preprocess(
                        image="./STARE/im0255.ppm"
).meanFilter(
).show(
).greyOpening(
).show()
        ```
        Notice how `show()` can be called after any operation. `show()` uses the
        PIL Image debugger to show the image.

        The implemented methods are generally limited to the methods describedin
        Marin et al ITM 2011. However some methods allow for different
        parameters to be used in the operation where the ones described in Marin
        et al ITM 2011 are merely defaults.

        To run the methods described in Marin et al 2011 in the same order as
        described then the method `process` can be used:
        ```
            obj =   Preprocess(
                        image="./STARE/im0003.ppm"
).process(
).show(
).save(
```
```

```
                    path="./im0003_processed.png"
                )
        ```

        Non standard requesites for running are:
            - scipy   https://www.scipy.org/
            - cv2   http://opencv-python-tutroals.readthedocs.io/en/latest/
            - skimage   http://scikit-image.org/

        @class Preprocess
        @param   image {string} The path to the image to be preprocessed.
        @param   maskTh {int} The threshold value to create the mask from
        @property source {string} Image source
        @property image {PIL obj} PIL Image object
        @property mask {numpy array} The mask matrix which is 0 in the area
            outside FOV and 1's inside FOV
        @property threshold {int} The threshold value from which the mask is
            made from. Lower intensity than threshold and the pixel is
            considered outside FOV and inside otherwise.
    """
    def __init__(self, image, maskTh=50):
        self.initialized = False
        self.__printStatus(
            "Initialize preprocessing for: " + image,
            isEnd=True,
            initial=True
)
        self.source = image
        self.name = image.split("/")[-1].split(".")[0]
        self.image = Image.open(image)
        self.loaded = self.image.load()
        # self.threshold=50
        self.threshold = maskTh
        self.extractColorBands()
        self.mask = numpy.uint8(
            numpy.greater(
                self.red_array,
                self.threshold
).astype(int)
)

    def save(self, path, array=numpy.empty(0), useMask=False, rotate=True):
        """
            Saves the image array as png at the desired path.

            @method save
            @param path {string} the path where the image will be saved.
            @param array {numpy array} The array which the image is made from,
                default is self.image_array
            @param useMask {Bool} Wether to reset non FOV pixel using the mask.
                Default is False
        """
```
```

```
104             if not array.any():
105                 array = self.image_array
106             if useMask:
107                 array = array * self.mask
108             self._arrayToImage(array).save(path, "png", rotate=rotate)
109             self.__printStatus("saving to " + path + "...")
110             self.__printStatus("[done]", True)
111             return self
112 
113         def _arrayToImage(self, array=numpy.empty(0), rotate=True):
114             """
115                 @private
116                 @method arrayToImage
117                 @param array {numpy array} array which is converted to an image
118                 @param rotate {Bool} If true the image is transposed and rotated to
119                     counter the numpy conversion of arrays.
120             """
121             self.__printStatus("array to image...")
122             if not array.any():
123                 array = self.image_array
124             img = Image.fromarray(numpy.uint8(array))
125             self.__printStatus("[done]", True)
126             if rotate:
127                 return img.transpose(Image.FLIP_TOP_BOTTOM).rotate(-90)
128             else:
129                 return img
130 
131         def show(
132                 self,
133                 array=numpy.empty(0),
134                 rotate=True,
135                 invert=False,
136                 useMask=False,
137                 mark=None
138             ):
139             """
140                 @method show
141                 @param array {numpy array} image array to be shoum.
142                 @param rotate {Bool} Wether to rotate countering numpys array
143                     conversion, default True.
144                 @param invert {Bool} Invert the image, default False.
145                 @param useMask {Bool} Reset non FOV pixels using the mask, default
146                     is False.
147             """
148             if not array.any():
149                 array = self.image_array
150             im = self._arrayToImage(array, rotate=rotate)
151             self.__printStatus("show image...")
152             if useMask:
153                 array = array * self.mask
154             if mark:
155                 im = im.convert("RGB")
```

```
156             pixels = im.load()
157             x, y = mark
158             for i in range(x-1, x+1):
159                 for j in range(y-1, y+1):
160                     # color an area around the mark
161                     # blue, for easilier visibility
162                     pixels[i, j] = (0, 0, 255)
163         if invert:
164             Image.eval(im, lambda x:255-x).show()
165         else:
166             print("#####", im.mode, "#####")
167             im.show()
168         self.__printStatus("[done]", True)
169         return self
170 
171     def extractColorBands(self):
172         """
173             Returns a greyscaled array from the green channel in
174             the original image.
175 
176             @method extractColorBands
177         """
178         self.__printStatus("Extract color bands...")
179         green_array = numpy.empty([self.image.size[0], self.image.size[1]], int)
180         red_array = numpy.empty([self.image.size[0], self.image.size[1]], int)
181         for x in range(self.image.size[0]):
182             for y in range(self.image.size[1]):
183                 red_array[x,y] = self.loaded[x,y][0]
184                 green_array[x,y] = self.loaded[x,y][1]
185         self.green_array = green_array
186         self.red_array = red_array
187         self.image_array = self.green_array
188         self.__printStatus("[done]", True)
189         return self
190 
191     def greyOpening(self, array=numpy.empty(0)):
192         """
193             Makes a 3x3 morphological grey opening
194 
195             @method greyOpening
196             @param array {numpy array} array to operate on.
197         """
198         self.__printStatus("Grey opening...")
199         if not array.any():
200             array = self.image_array
201         self.grey_opened = ndimage.morphology.grey_opening(array, [3,3])
202         self.image_array = self.grey_opened * self.mask
203         self.__printStatus("[done]", True)
204         return self
205 
206     def meanFilter(self, m=3, array=numpy.empty(0)):
207         """
```

```
208             Mean filtering, replaces the intensity value, by the average
209             intensity of a pixels neighbours including itself.
210             m is the size of the filter, default is 3x3
211 
212             @method meanFilter
213             @param m {int} The width and height of the m x m filtering matrix,
214                 default is 3.
215             @param array {numpy array} the array which the operation is carried
216                 out on.
217         """
218         self.__printStatus("Mean filtering " + str(m) + "x" + str(m) + "...")
219         if not array.any():
220             array = self.image_array
221         if array.dtype not in ["uint8", "uint16"]:
222             array = numpy.uint8(array)
223         mean3x3filter = rank.mean(array, square(m), mask=self.mask)
224         self.image_array = mean3x3filter * self.mask
225         self.__printStatus("[done]", True)
226         return self
227 
228     def gaussianFilter(self, array=numpy.empty(0), sigma=1.8, m=9):
229         """
230             @method gaussianFilter
231             @param array {numpy array} the array the operation is carried out
232                 on, default is the image_array.
233             @param sigma {Float} The value of sigma to be used with the gaussian
234                 filter operation
235             @param m {int} The size of the m x m matrix to filter with.
236         """
237         self.__printStatus(
238             "Gaussian filter sigma=" + str(sigma) + ", m=" + str(m) + "..."
239         )
240         if not array.any():
241             array = self.image_array
242         self.image_array = cv2.GaussianBlur(array, (m,m), sigma) * self.mask
243         self.__printStatus("[done]", True)
244         return self
245 
246     def _getBackground(self, array=numpy.empty(0), threshold=None):
247         """
248             _getBackground returns an image unbiased at the edge of the FOV
249 
250             @method _getBackground
251             @param array {numpy array} the array the operation is carried out
252                 on, default is the image_array.
253             @param threshold {int} Threshold that is used to compute a
254                 background image, default is self.threshold.
255         """
256         if not array.any():
257             array = self.red_array
258         if not threshold:
259             threshold = self.threshold
```

```
260         saved_image_array = self.image_array
261         background = self.meanFilter(m=69).image_array
262         self.__printStatus("Get background image...")
263         # reset self.image_array
264         self.image_array = saved_image_array
265         for x in range(len(background)):
266             for y in range(len(background[0])):
267                 if array[x,y] > threshold:
268                     if x-35 > 0:
269                         x_start = x-35
270                     else:
271                         x_start = 0
272                     if x+34 < len(background):
273                         x_end = x+34
274                     else:
275                         x_end = len(background) -1
276                     if y-35 > 0:
277                         y_start = y-35
278                     else:
279                         y_start = 0
280                     if y+35 < len(background[0]):
281                         y_end = y+35
282                     else:
283                         y_end = len(background[0]) -1
284                     # 1 is added to the right and bottom boundary because of
285                     # pythons way of indexing
286                     x_end += 1
287                     y_end += 1
288                     # mask is is the same subMatrix but taken from the original
289                     # image array
290                     mask    = array[x_start:x_end, y_start:y_end]
291                     # indexes of the non fov images
292                     nonFOVs = numpy.less(mask, threshold)
293                     # indexes of FOVs
294                     FOVs    = numpy.greater(mask, threshold)
295                     # subMat is a 69x69 matrix with x,y as center
296                     subMat  = background[x_start:x_end, y_start:y_end]
297                     # subMat must be a copy in order to not allocate values into
298                     # background directly
299                     subMat = numpy.array(subMat, copy=True)
300 
301                     subMat[nonFOVs] = subMat[FOVs].mean()
302                     # finding every element less than 10 from the original image
303                     # and using this as indices on the background subMatrix
304                     # is used to calculate the average from the 'remaining
305                     # pixels in the square'
306                     background[x,y] = subMat.mean()
307 
308         self.__printStatus("[done]", True)
309         return background
310 
311     def subtractBackground(self, array=numpy.empty(0)):
```

```
312             """
313                 @method subtractBackground
314                 @param array {numpy array} the array the operation is carried out
315                     on, default is the image_array.
316             """
317             if not array.any():
318                 array = self.image_array
319             background = self._getBackground() * self.mask
320             self.__printStatus("Subtract background...")
321             self.image_array = numpy.subtract(
322                 numpy.int16(array),
323                 numpy.int16(background)
324             ) * self.mask
325             self.__printStatus("[done]", True)
326             return self
327 
328         def linearTransform(self, array=numpy.empty(0)):
329             """
330                 Shade correction maps the background image into values
331                 that fits the grayscale 8 bit images [0-255]
332                 from: http://stackoverflow.com/a/1969274/2853237
333 
334                 @method linearTransform
335                 @param array {numpy array} the array the operation is carried out
336                     on, default is the image_array.
337             """
338             self.__printStatus("Linear transforming...")
339             if not array.any():
340                 array = self.image_array
341             # Figure out how 'wide' each range is
342             leftSpan = array.max() - array.min()
343             rightSpan = 255
344             array = ((array - array.min()) / leftSpan) * rightSpan
345             self.image_array = array * self.mask
346             self.__printStatus("[done]", True)
347             return self
348 
349         def transformIntensity(self, array=numpy.empty(0)):
350             """
351                 @method transformIntensity
352                 @param array {numpy array} the array the operation is carried out
353                     on, default is the image_array.
354             """
355             self.__printStatus("Scale intensity levels...")
356             if not array.any():
357                 array = self.image_array
358 
359             counts = numpy.bincount(array.astype(int).flat)
360             ginput_max = numpy.argmax(counts)
361             for x in range(len(array)):
362                 for y in range(len(array[0])):
363                     st = str(array[x,y]) + " ==> "
```

```
                st += str(array[x,y] + 128 - ginput_max) + " "
                array[x,y] + 128 - ginput_max
                if array[x,y] < 0:
                    array[x,y] = 0
                elif array[x,y] > 255:
                    array[x,y] = 255
        s = str(ginput_max)
        self.image_array = array * self.mask
        self.__printStatus("[done]", True)
        return self

    def vesselEnhance(self, array=numpy.empty(0)):
        """
            @method vesselEnhance
            @param array {numpy array} the array the operation is carried out
                on, default is the image_array.
        """
        self.__printStatus("Vessel enhancement...");
        if not array.any():
            array = self.image_array
        # disk shaped mask with radius 8
        disk_shape = disk(8)
        # the complimentary image is saved to hc:
        array = numpy.uint8(array)
        hc = 255 - array
        # Top Hat transform
        # https://en.wikipedia.org/wiki/Top-hat_transform
        # White top hat is defined as the difference between
        # the opened image and the original image.
        # in this case the starting image is the complimentary image `hc`
        self.image_array = white_tophat(hc, selem=disk_shape) * self.mask
        self.__printStatus("[done]", True)
        return self

    def __printStatus(self, status, isEnd=False, initial=False):
        """
            @private
            @method __printStatus
            @param status {string}
            @param isEnd {Bool} Wether to end with a newline or not, default is
                false.
            @param initial {Bool} Wether this is the first status message to be
                printed, default False.
        """
        if not initial and not isEnd:
            status = "\t" + status
        if initial:
            status = "\n" + status
        if isEnd:
            delim="\n"
        else:
            delim=""
```

```
416             # set tabs so status length is 48
417             tabs = ((48 - len(status)) // 8) * "\t"
418             status += tabs
419         print(status, end=delim, sep="")
420 
421     def process(self, enhance=True, onlyEnhance=False):
422         """
423             `process` starts the preprocess process described in
424             Marin et al ITM [2011]
425             The article works with two types of preprocessed images.
426             The first is the convoluted image obtained with all operations
427             except for `vesselEnhance` denoted as a homogenized image. And the
428             second is the vessel enhanced image which is the convolution of the
429             vessel enhancement operation on the homogenized image.
430 
431             This method supports both images. If `enhance` is False then
432             self.image_array will be of the homogenized image and afterwards the
433             vessel enhanced image can be computed without starting over by
434             setting `onlyEnhance` to True. So to compute both images one at a
435             time one could call:
436 
437             ```
438                 obj = Preprocess(
439                         "./im0075.ppm"
440                     )
441                     .process(
442                         enhance=False
443                     ).show(
444                     ).process(
445                         onlyEnhance=True
446                     ).show()
447             ```
448 
449             @method process
450             @method enhance {Bool} Wether to also process the vessel enhancement
451                 operation or not, default True.
452             @method onlyEnhance {Bool} Wether to only do the vessel enhancement
453                 operation, default False.
454         """
455         if not onlyEnhance:
456             self.greyOpening()
457             self.meanFilter()
458             self.gaussianFilter()
459             self.subtractBackground()
460             self.linearTransform()
461             self.transformIntensity()
462         if enhance or onlyEnhance:
463             self.vesselEnhance()
464         # returns the object where
465         # all described preprocess has taken place
466         # available on self.feature_array or self.show(), self.save(<path>)
467         return self
```

A.5 FeatureExtraction class

```
import scipy
import numpy
from PIL import Image
from preprocessing import Preprocess
import cv2
from unbuffered import Unbuffered
import sys
from skimage.morphology import binary_erosion
from skimage.morphology import square
# make print() not wait on the same buffer as the
# def it exists in:
sys.stdout = Unbuffered(sys.stdout)

class FeatureExtraction:
    """
        @param image {string} Source of raw image
        @param images {list} Source of preprocessed images. Where images[0] is
        the homogenized image and images[1] is the vessel enhanced image.
        @param GTPath {string} Wether image has ground truth. If the image has
        ground truth image with correct labels then gt is a path to the
        groundtruth image.
    """
    def __init__(self, image=False, images=[], GTPath=""):
        if images:
            try:
                self.homogenized = numpy.array(Image.open(images[0]))
                self.vesselEnhanced = numpy.array(Image.open(images[1]))
                self.images = images
            except IndexError:
                print("""`images` parameter must include the homogenized image
                    at `images[0]` and vessel enhanced image at `images[1]`""")
                raise
        else:
            self.preprocess          = Preprocess(image)
            self.homogenized    = self.preprocess.process(enhance=False).image_array
            self.vesselEnhanced = self.preprocess.process(onlyEnhance=True).image_array
            self.mask           = self.preprocess.mask
            self.source         = image
            self.image          = Image.open(image)
            self.loaded         = self.image.load()
        if len(GTPath):
            self.gt             = True
            self.groundtruth    = Image.open(GTPath)
        else:
            self.gt             = False

        self.feature_array      = numpy.empty(0)

    def __getHomogenized(self, forceNew=False):
        raise NotImplementedError
    """
```

```
52          `exportCSV` exports `self.feature_array` to `filename` unless `array`
53          parameter is set if `balanced` then the exported features will have an
54          equal amount of class 0 and class 1.
55          The parameter `delim` can be used to change the seperator from commas to
56          some other character.
57
58          @method exportCSV
59          @param filename {string} Name of file including the path to it where
60              features will be exported to
61          @param array {numpy array} The feature array to export
62          @param delim {string} The delimeter
63          @default ","
64          @param balanced {bool} Wether to export the full feature array or a
65              balanced version with equal class representation
66          @default False
67      """
68      def exportCSV(
69              self,
70              filename="",
71              array=numpy.empty(0),
72              delim=",",
73              balanced=False
74          ):
75
76          if not array.any():
77              array = self.feature_array
78          if balanced:
79              zeros   = array[numpy.less(array[:,0], 1)]
80              ones    = array[numpy.greater(array[:,0], 0)]
81              if len(zeros) > len(ones):
82                  indices = numpy.random.choice(
83                      len(zeros),
84                      size=len(ones),
85                      replace=False
86                  )
87                  ones = numpy.concatenate(
88                      (ones, zeros[indices]),
89                      axis=0
90                  )
91                  array = ones
92              if len(ones) > len(zeros):
93                  indices = numpy.random.choice(
94                      len(ones),
95                      size=len(zeros),
96                      replace=False
97                  )
98                  zeros = numpy.concatenate(
99                      (zeros, ones[indices]),
100                     axis=0
101                 )
102                 array = zeros
103         if not len(filename):
```

```
            if hasattr(self, "source"):
                filename = "extracted_" + self.source
            else:
                filename = "extracted_" + self.images[1]
        if self.gt:
            formatting = ['%d', '%.0f', '%.0f', '%f', '%f', '%.0f', '%f', '%f']
            header = """label,\tfeat. 1,\tfeat. 2,\tfeat. 3,\tfeat. 4,\tfeat. 5,
                    \tHu mom. 1,\tHu mom. 2"""
        else:
            formatting = ['%.0f', '%.0f', '%f', '%f', '%.0f', '%f', '%f']
            header = """feat. 1,\tfeat. 2,\tfeat. 3,\tfeat. 4,\tfeat. 5,
                    \tHu mom. 1,\tHu mom. 2"""
        numpy.savetxt(
            filename,
            array,
            fmt=formatting,         # formatting
            delimiter=',\t',        # column delimiter
            newline='\n',           # new line character
            footer='end of file',   # file footer
            comments='# ',          # character to use for comments
            header=header)          # file header
    """
        `normalize` is used to normalize the feature_array. If comp_only
        (compute only) is set to `True` then only `self.std_vector` and
        `self.mean_vector` will be set but the value of `self.feature_array`
        will not be set. This can be useful if computing an accumulated mean and
        standard deviation and then using the `mean` and `std` parameter later
        to normalize with the accumulated mean and average standard deviation
        vectors.

        @method normalize
        @param array {numpy array} The feature array if not set then
            `self.feature_array`.
        @param mean {numpy array} The mean to use in the normalization. If not
            set then it will be computed over the inside FOV pixels of the
            `array` using the `self.mask`.
        @param std {numpy array} The standard deviation to be used in
            normalization.
        @param comp_only {bool} If true then mean, sample variance and standard
            deviation will be computed and saved to `self.var_vector`,
            `self.std_vector` and `self.mean_vector` respectively. But they wont
            be used to normalize the feature array.
        @default False
    """
    def normalize(
            self,
            array=numpy.empty(0),
            mean=numpy.empty(0),
            std=numpy.empty(0),
            comp_only=False
        ):
        if not array.any():
```

```
            array = self.feature_array
        # preserve label column
        # compute mean and std excluding out of FOV pixels
        indices = numpy.greater(self.mask.flatten(), 0)
        FOV = array[indices]

        # Since mean should only be computed on the training set
        # the assumption of ignoring the first column is made, since
        # this is the label column.
        if not mean.any():
            mean    = FOV.mean(axis=0)[1:]
        if not std.any():
            std     = FOV.std(axis=0)[1:]
            var     = FOV.var(axis=0)[1:]
        if comp_only:
            self.var_vector     = var
            self.std_vector     = std
            self.mean_vector    = mean
        else:
            if self.gt:
                labels = array[:,0]
                array[:,1:] = (array[:,1:] - mean) / std
            else:
                array = (array - mean) / std
            if self.gt:
                array[:,0] = labels
                # since there is a groundtruth then the first column
                # will be the label column, the rest are the actual features.d
            self.feature_array = array
        return self

    def computeFeatures(self, forceNew=False):
        if forceNew:
            return self._extract()

        elif self.feature_array.any():
            return self
        else:
            return self._extract()

    """
        `_extract` is responsible of extracting the feature array for every
        pixel in the preprocessed image. If optional parameters
        `homogenized_array` and `ve_array` are not provided then

        @method _extract
        @param homogenized_array {numpy array} The homogenized image from
            preprocessing
        @param ve_array {numpy array} The vessel enhanced image from
            preprocessing
    """
    def _extract(
```

```
208             self,
209             homogenized_array=numpy.empty(0),
210             ve_array=numpy.empty(0)
211         ):
212         if not homogenized_array.any():
213             homogenized_array = self.homogenized
214         if not ve_array.any():
215             ve_array = self.vesselEnhanced
216         # erode image using an eroded mask
217         mask = binary_erosion(self.mask, square(10))
218         homogenized_array = homogenized_array * mask
219         # # # # # # # # # # # # # # # # # # # # # #
220         print("Extracting features ", end=" ")
221         print("\t\t[", end="")
222         self.feature_array = []
223         for x in range(len(homogenized_array)):
224             for y in range(len(homogenized_array[0])):
225                 if self.mask[x,y] or True: # disabled for now
226                     ########################################
227                     xstart  = x - 8 if x-8 >= 0 else 0
228                     ystart  = y - 8 if y-8 >= 0 else 0
229 
230                     xend    = x + 8 if x+8 < len(ve_array) else len(ve_array) -1
231                     yend    = y + 8 if y+8 < len(ve_array[0]) else len(ve_array[0]) -1
232                     # 1 is added to the right and bottom boundary because of
233                     # pythons way of indexing
234                     xend += 1
235                     yend += 1
236 
237                     subarea = ve_array[xstart:xend, ystart:yend]
238 
239                     if subarea.max() != 0:
240 
241 
242                         Hu0, Hu1 = self.__moments(subarea)
243 
244                         ########################################
245                         xstart  = x-4 if x-4 >= 0 else 0
246                         ystart  = y-4 if y-4 >= 0 else 0
247 
248                         xend    = (x+4
249                             if x+4 < len(homogenized_array)
250                             else len(homogenized_array) -1)
251                         yend    = (y+4
252                             if y+4 < len(homogenized_array[0])
253                             else len(homogenized_array[0]) -1)
254                         # 1 is added to the right and bottom boundary because of
255                         # pythons way of indexing
256                         xend += 1
257                         yend += 1
258 
259                         subarea = homogenized_array[xstart:xend, ystart:yend]
```

```
                        FOV     = numpy.greater(subarea, 0)
                        subarea = (subarea[FOV]
                            if FOV.any() and homogenized_array[x,y] > 0
                            else numpy.array([0]))
                        # equation 5 from Marin et al.
                        f1      = homogenized_array[x,y] - subarea.min()
                        # equation 6 from Marin et al.
                        f2      = subarea.max() - homogenized_array[x,y]
                        # equation 7 from Marin et al.
                        f3      = homogenized_array[x,y] - subarea.mean()
                        # equation 8 from Marin et al.
                        f4      = subarea.std()
                        # equation 9 from Marin et al.
                        # inverting the background, so setting zero to 255
                        f5      = homogenized_array[x,y]
                        ########################################

                        if self.gt:
                            # values in groundtruth are either 255 or 0
                            gtval = self.groundtruth.getpixel((x,y))
                            label = gtval if gtval == 0 else 1
                            features = [label, f1, f2, f3, f4, f5, Hu0, Hu1]
                        else:
                            features = [f1, f2, f3, f4, f5, Hu0, Hu1]

                    elif not self.gt:
                        features = [0, 0, 0.0, 0.0, 0, 0.0, 0.0]

                    else:
                        # values in groundtruth are either 255 or 0
                        gtval = self.groundtruth.getpixel((x,y))
                        label = gtval if gtval == 0 else 1
                        features = [label, 0, 0, 0.0, 0.0, 0, 0.0, 0.0]

                    self.feature_array.append(features)
                if x % (len(homogenized_array) * 0.05) < 1:
                    print("#", end="")

        self.feature_array = numpy.array(self.feature_array)
        print("]")
        return self

    """
        `__moments` computes the first two Hu moment over some array given by
        the parameter `subarray`.

        @private
        @method __moments
        @param subarray {numpy array} The area which the Hu moments are computed
            over.
    """
    def __moments(self, subarray):
```

```
312             """
313                 I_HU(x,y) = subarray(x,y) * gaussian_matrix(x,y)
314 
315                 returns absolute value of the log of the first two Hu moments
316             """
317             I_HU = self.__gausMatrix(subarray)
318             h1, h2 = cv2.HuMoments(cv2.moments(I_HU))[0:2]
319             h1 = numpy.log(h1) if h1 != 0 else h1
320             h2 = numpy.log(h2) if h2 != 0 else h2
321             return numpy.absolute( [h1[0], h2[0]] )
322 
323         def __gausMatrix(self, array, mu=0.0, sigma=1.7):
324             x, y = array.shape
325             return scipy.ndimage.filters.gaussian_filter(array, 1.7)
```

A.6 Configuration file

```
{
    "generalSettings": {
        "DestinationFolder": "quickfeats",
        "ImagePath": "../STARE",
        "LabelPath": "../labels",
        "extract_balanced": false,
        "override_params": true
    },
    "parameters": {
        "featureMeans": [
            9.55860709190124,
            8.383207004324243,
            -0.016580067798572018,
            4.795640306869291,
            147.52288315138432,
            2.778818816145008,
            7.241800345905508
        ],
        "featureStd": [
            11.13983602214089,
            10.68084747300321,
            3.528459958094444,
            5.071280187477676,
            26.6519371231476,
            1.2742274005193075,
            3.070712498583245
        ]
    }
}
```

A.7 Synopsis

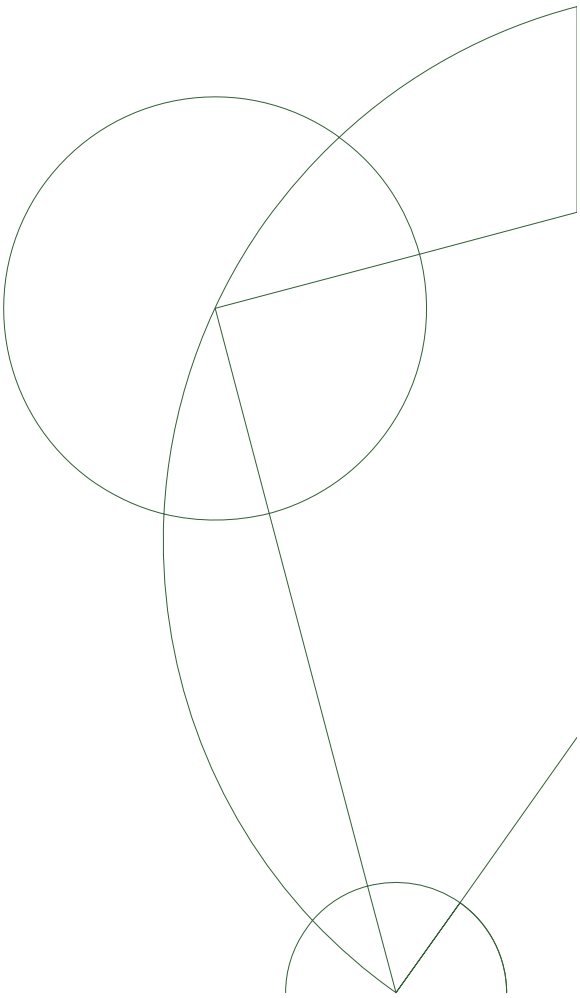
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Synopsis

Jesper Henrichsen — TGW831

Vessel segmentation in retinal images

An implementation in Python using feed-forward neural networks via Shark library



Aasa Feragen & Christian Igel

March 14, 2016

1 Motivation

Image analysis is playing a larger and larger role in medicare. Therefore research in image analysis constantly tries to push forward the reliability of state of the art image analysis techniques, so that doctors may confidently outsource part of their responsibility to computers. Having machines learning the statistical differences between healthy and sick on thousands of scans and screenings helps reduce the time from scanning to making the correct diagnosis. The potential for computers to assists doctors in diagnosing patients is an important reason for the increased focus on this research field in recent years.

There is evidence suggesting that analysing retinal bloodvessels can aid in making early diagnosis of Alzheimer's disease [1]. Furtheremore retinal bloodvessels is used when determining the risk of diabetes patients to develop decreased visibility and in some cases blindness. Another benefit of analysing retinal images are that they are comparatively cheap and non-invasive to make. It is therefore important to optimize segmentation processes for retinal images in a push to maximize the amount of valuable information that can be acquired from these images and the confidence that can be put into that information.

2 Problem statement

How well can a neural network algorithm segment retinal blood vessels? (1)

In this project the focus will be on a state of the art technique to segment bloodvessel pixels in retinal images. The technique is developed by Marin et al [2] and was published in 2011. The purpose of this project is to make a program that implements this technique and to attempt to answer the question in (1) from the perspective of this technique.

3 Scope delimitation

In general, the scope of the project in terms of segmentation will be shaped after the documented approach of the article by Marin et al. [2]. Becuase of the complexity and time consumption of a neural network implementation, it is beyond the scope of this project to implement the algorithm used in the classification step of the project. Instead, the Shark data analysis library developed at Datalogisk Institut ved Københavns Universitet (DIKU) will be relied upon [3].

4 Method

Table 1: Breakdown of overall project methodology

Preprocessing	
	— Even lightning variations
	— Increase contrasts between vessel and nonvessel
	— reduce noise, such as nonconnected vessel-looking pixels
	— Vessel enhancement
Feature extraction	
	— Gray-level-based features
	— Moment invariants-based features
Classification	
	— Neural network setup in Shark
	— Neural network training and testing

The methodology of this project is an approach to retinal image analysis that is described in Marin et al. [2]. The overall breakdown of this approach can be seen in Table 1, which shows that there are three main descriptors each with its own set of methods.

4.1 Preprocessing

In the preprocessing step the desire is to prepare the images for the feature extraction step. This is done by evening out the lightning variation in the images to be as uniform as possible, and getting rid of pixels that have a higher intensity due to light reflexion. There is also general noise in the images that may cause false positives or false negatives if left in the images. In the end the desired final product of the preprocessing step is an image where bloodvessel pixels have been enhanced and put in the foreground while everything else is put in the background. Foreground and background is to be understood in terms of the intensity of pixels, where background pixels has low intensity, and foreground pixels has high intensity. The image preprocessing step produces a preprocessed image through convolution with the functions involved during this step. See Figure 1b for an example of vessel enhancement, the last applied function during preprocessing.

4.2 Feature extraction

In the feature extraction step the concern is to get the set of features which describe the difference between nonvessel and vessel pixels the best. In this step a 7 dimensional feature vector is extracted for every pixel in the image. These features are all as described in Marin et al. [2]. Similar for all features, except one, is that a neighborhood around the considered pixel is used to determine the feature. For the first 5 features a neighborhood of 9×9, with the considered pixel in the center, is used. From Marin et al. [2] the first 5 extracted features are:

$$f_1(x,y) = I_H(x,y) - min_{(s,t)\in S^9_{x,y}}\{I_H(s,t)\} \tag{2}$$

$$f_2(x,y) = max_{(x,y)\in S^9_{x,y}}\{I_H(s,t)\} - I_H(x,y) \tag{3}$$

$$f_3(x,y) = I_H(x,y) - mean_{(s,t)\in S^9_{x,y}}\{I_H(s,t)\} \tag{4}$$

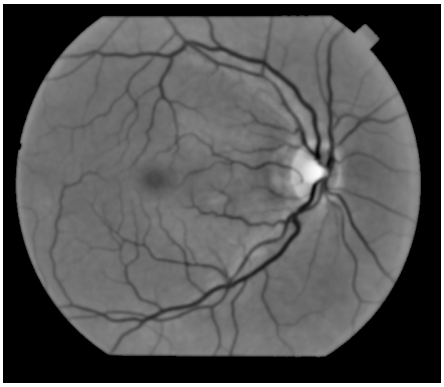
$$f_4(x,y) = std_{(s,t)\in S^9_{x,y}}\{I_H(s,t)\} \tag{5}$$

$$f_5(x,y) = I_H(x,y) \tag{6}$$

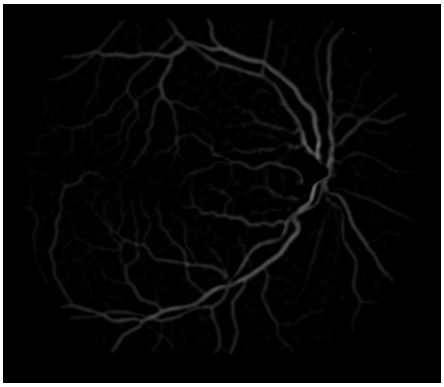
Here I_H is a homogenized image, that is all purposes of the preprossesing step described above have been achieved except for the vessel enhancement part. An example of a homogenized image is shown in Figure 1a.

Figure 1: Examples of images after vesselenhancement and homogenization respectively

(a) Homogenized image I_H



(b) An example of a vessel enhanced image



The last two extracted features are the absolute log value of the first two Hu moment invariants on a neighborhood of 17×17 pixels. These two features are extracted from the vessel enhanced image in Figure 1b which is the result of the vessel enhanced image on I_H from Figure 1a.

4.3 Classification

Classification is done using the Shark library's feed-forward neural network. The topology of the network is a 3 layer network, besides the input and output layer. There are 15 neurons in each hidden layer. The input layer has 7 input neurons for each feature and the network has just one output neuron since the classification is between vesselpixel or non vesselpixel.

When it comes to metrics then the stated results in Marin et al. TMI 2011 [2] will be used as a first indicator of the completeness of the implementation. The results will not be exactly the same, but they will be close.
A second metric will be a comparison between the extracted features performed in the described neural network and then the same features used in a support vector machine (SVN) algorithm, also via the Shark library.

5 Timeplan

Figure 2: Project timeline illustrated in Gantt graph

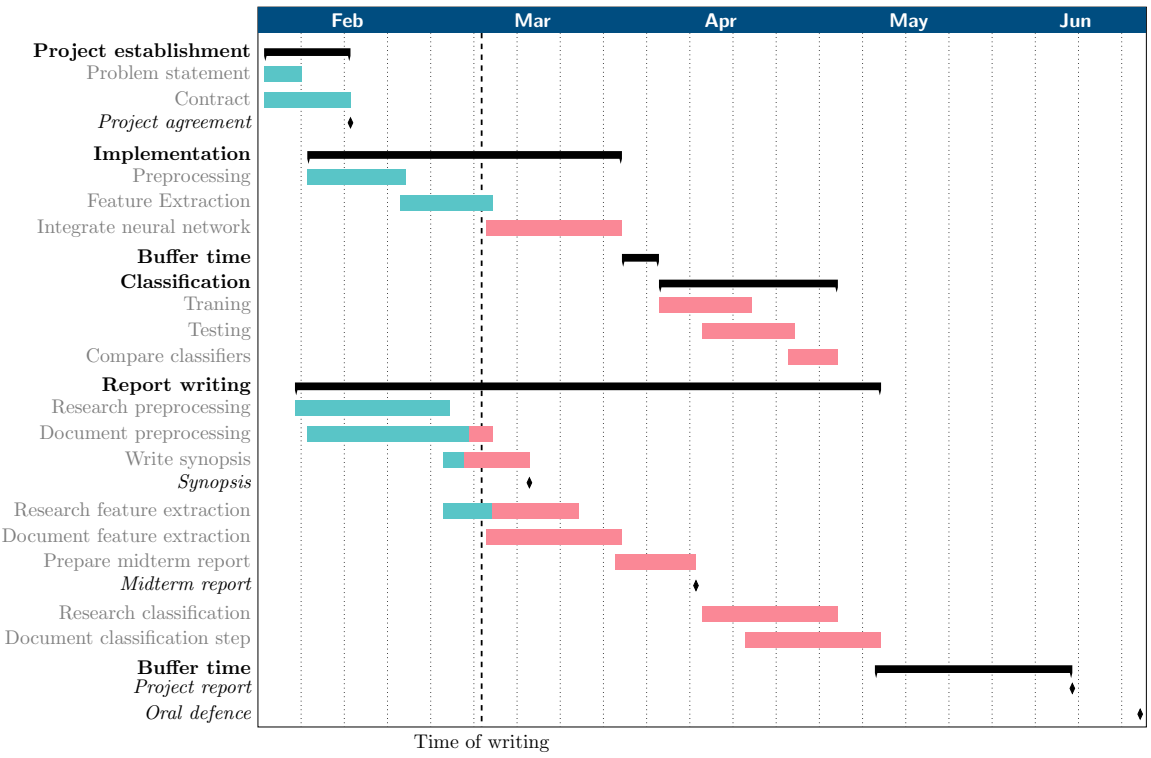


Figure 2 shows a rough breakdown of the overall deliverables that make up this project both in terms of report writing, implementation and classification work. All the following deadlines are marked: the *project agreement*, *synopsis*, *midterm report* hand-in, *project report* hand-in and the earliest estimate for an oral exam date as *oral defence*.

Black bars outlines the total length of a certain project delivery and all deliverables are marked with green and red. A green color indicates completed, while red is a to-be-done indicator. The status as of the time of writing is indicated by a dashed line in the seccond week of March.

There are two time buffers that allow flexibility for unanticipated obstacles during the project. The main time buffer is put at the end of the project but may be consumed by deliverables at any step of the project.

The research deliverables are named so because they are thought of as a period of in-depth reading of a relevant topic in correspondance with report writing of that topic. These periods are more difficult to pin down but they coexists with the writing and generally have a lifetime of the same length as the time set aside for the equivalent writing deliverable.

If the time buffers ends up not being used then that time, depending on the amount left, will be used to explore interesting questions that may have presented themselves during the project. This could be in terms of making small changes during the preprocessing step and see how they affect the performance of the classification. Another area of the project that could be explored further if any time is left is the training phase of the nerual network classification.

References

[1] Clement L. Trempe J. Wallace McMeel Fatmire Berisha, Gilbert T. Feke and Charles L. Schepens. Retinal Abnormalities in Early Alzheimer's Disease. *Investigative Ophthalmology & Visual Science (IOVS)*, 48, May 2007.

[2] Diego Marìn, Arturo Aquino, Manuel Emilio Gegùndez-Arias, and Josè Manuel Bravo. A new supervised method for blood vessel segmentation in retinal images by using gray-level and moment invariants-based features, 2011.

[3] Christian Igel, Verena Heidrich-Meisner, and Tobias Glasmachers. Shark. *Journal of Machine Learning Research*, 9:993–996, 2008.