Anders Bjorholm Dahl
Vedrana Andersen Dahl

Advanced Image Analysis

SELECTED TOPICS

DTU Compute, May 24, 2022

Contents

Preface

This lecture note is a collection of topics for the students taking the course 02506 Advanced Image Analysis at Technical University of Denmark. The note provides background material for the course together with practical guidelines and advice for carrying out tasks in image analysis. The topics are selected to represent problems that you typically meet as an engineer that specialize in image analysis.

Image analysis is a rapidly growing field of research with a wealth of methods constantly being developed and published. This note is not intended to give a complete overview of the field. Instead, we focus on general principles for image analysis with the aim of giving students the skill-set needed for exploring new methods. General principles relate to identifying relevant image analysis problems, finding suitable methods for quantification, implementing image analysis algorithms and verifying their performance. This requires programming skills and the ability to translate a mathematical description into an efficient functioning program.

Image analysis methods published in scientific articles can be challenging to implement as a computer programs, since they are described using mathematical notation, which my vary between articles. In some cases the notation can be very different from the code you need to write. One aim with this note is to guide the implementation of image analysis algorithms from descriptions in articles to the functioning programs. This is done through examples, practical tips, and advice on designing useful tests to ensure that the obtained implementation gives the expected output.

There are several papers and book chapters that describe the methods to be implemented during the course. These are integral parts of the course curriculum, and should be read when working with the examples in this note.

The structure of the note is as follows. First comes a general introduction to a few central aspects relevant for image analysis along with the first introductory exercise, which has the purpose of refreshing basic image analysis concepts. The main text includes three compulsory

exercises. Towards the end of the note we provide several examples for the final exercise in form of a mini-project.

During the course you may be implementing methods and algorithms that are already integrated in existing software libraries. These commercial or public implementations might be better than what you can achieve given the time available for the exercises. The reason to redo what other people have already done is to gain insight and understanding of how image analysis methods work, and to give you the skill-set needed for implementing or modifying advanced methods where there might not be an available implementation. It can however be a good idea to use existing implementations for evaluating your implementation.

1 Introduction

We refer to images as regularly sampled signals in 2D or 3D space that represent a measurement, typically a measure of light intensity. In image analysis, we use the image to obtain some information about the signals we have measured. Here we discuss aspects to consider when working with images regarding what images show, how images are represented, and how they were created.

We start with the mathematical notation of images. One way of representing an image using mathematical notation is as a function $I(x, y)$ with $I : \Omega \subset \mathbb{R}^2 \rightarrow \mathbb{R}$, which means that a scalar value $I(x, y)$ is assigned to each coordinate (x, y) in the image domain Ω. Sometimes we consider the image as digital signal sampled at integer values, i.e. running from 1, such that $x \in \{1, ..., X\}$ and $y \in \{1, ..., Y\}$. Still, many papers will ignore the discrete nature of the images, and treat I as a continuous function. In general, scientific articles show substantial variation on the notation of an image, e.g. it is often implicitly assumed that the image lives in 2D space, and the notation would simply be a symbol like I.

In the computer program, a gray-scale image is represented as a 2D array of numbers. Here, the indexing of the array elements is implicitly related to image space (x, y), however, one needs to consider program-specific details: 0 or 1 indexing, placement of the origin, and so on. For mathematical treatment of certain topics it is convenient to considered images as if centered around origin. For example, when we talk about filtering kernels. Also here it is important to be aware of the difference between the notation, and the actual representation of the kernel.

A 3D image is typically termed a volume, and again here we can model it as a function $I : \Omega \subset \mathbb{R}^3 \rightarrow \mathbb{R}$. Similar to before, I is a function that maps to a scalar value, but here from a 3D coordinate (x, y, z). Volumetric images are often reconstructed from projection data obtained using a scanner, e.g. a CT or an MRI scanner. In a volumetric image, the three spatial dimensions encode intensity information similar to a 2D image. This means that if we apply operators on a volumetric image, we would use a 3D operator, e.g. an averaging filter in three dimensions.

In some cases the sampling is anisotropic, which is typically seen in e.g. medical CT images, and this can influence the applied analysis methods.

Spectral images have multiple measures in each pixel. This means that each pixel value may be represented as a 1D array of values that encode the recorded spectral bands. A common example are the RGB images where $I : \Omega \subset \mathbb{R}^2 \to \mathbb{R}^3$ encodes the red, green, and blue band. If more spectral bands are recorded, we are typically talking about multispectral or hyper-spectral images where $I : \Omega \subset \mathbb{R}^2 \to \mathbb{R}^n$ normally with $n > 3$. For 2D spectral images we would often apply operators in each of the spectral bands independently. Using the smoothing example from before, but now in a RGB image, we would preform 2D smoothing in the R-band, G-band, and B-band individually.

Another common image-related representation is a movie. A movie is a set of consecutive images also called frames sampled over time. This can be modeled as $I(x, y, t)$ where $I : \Omega \subset \mathbb{R}^3 \to \mathbb{R}$ for a gray scale movie or $I : \Omega \subset \mathbb{R}^3 \to \mathbb{R}^3$ for an RGB movie. You can also have a multispectral movie ($I : \Omega \subset \mathbb{R}^3 \to \mathbb{R}^n$ with $I(x, y, t)$) or a volumetric movie ($I : \Omega \subset \mathbb{R}^4 \to \mathbb{R}$ with $I(x, y, z, t)$). For movies we would typically expect small changes between frames, and this can be utilized in the analysis.

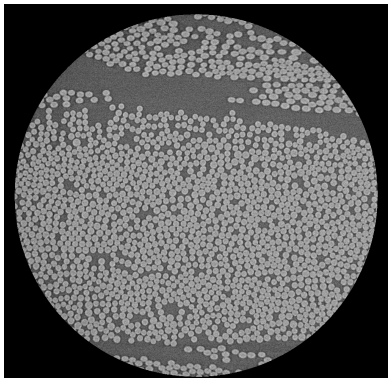

Figure 1.1: Slice of a CT image of glass fibers viewed orthogonal to the fiber direction.

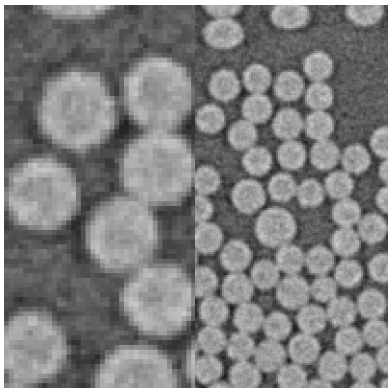

Figure 1.2: Two zoomed in images from image shown in Figure 1.1.

1.1 *Introductory exercise*

This exercise is aimed at refreshing or introducing concepts from basic image analysis curriculum and other related subjects. It contains some topics that will be useful at a later stage in the course. You are expected to carry out the first exercises, whereas the last exercises are not mandatory, but you are welcome to read or solve those exercises also.

1.1.1 *Image convolution*

Image convolution is a central tool in image analysis, and in this exercise you will investigate some properties of image convolution with a Gaussian kernel and its derivatives. You can read more about fundamentals of convolution and filtering in [1], Chapter 5.

For two continuous functions, convolution is defined as

$$(f * g)(x) = \int_{-\infty}^{\infty} f(x - \tau) g(\tau) \mathrm{d}\tau \, . \tag{1.1}$$

Convolution is commutative, but we usually distinguish between the signal and the kernel, and we say that the signal f is convolved with the kernel g.

[1] Wilhelm Burger, Mark James Burge, Mark James Burge, and Mark James Burge. *Principles of digital image processing*, volume 54. Springer, 2009

For a discrete sampled signal we get

$$(f * g)(x) = \sum_{i=-l}^{l} f(x-i)g(i) \ . \tag{1.2}$$

A convolution with a square kernel in 2D is given by

$$(f * g)(x,y) = \sum_{i=-l}^{l} \sum_{j=-l}^{l} f(x-i, y-j)g(i,j) \ . \tag{1.3}$$

In image analysis, a Gaussian kernel is often used used for image smoothing by filtering. The 1D Gaussian is defined by

$$g(x;t) = \frac{1}{\sqrt{2t\pi}} e^{-x^2/(2t)} \ , \tag{1.4}$$

where $t = \sigma^2$ is the variance of the Gaussian normal distribution. The 2D isotropic Gaussian is given by

$$g(x,y;t) = \frac{1}{2t\pi} e^{-(x^2+y^2)/(2t)} \ . \tag{1.5}$$

The Gaussian is separable ([2] Section 5.3.1), which means that we can convolve the image using two orthogonal 1D Gaussians, and obtain the same result as when convolving with 2D Gaussian of the same variance. This can speed up convolutions significantly, especially for large convolution kernels.

[2] Wilhelm Burger, Mark James Burge, Mark James Burge, and Mark James Burge. *Principles of digital image processing*, volume 54. Springer, 2009

Another property of the Gaussian convolution is the so-called *semi-group structure*, which means that we get the same convolution using a single large Gaussian as we get using several small ones

$$g(x,y;t_1+t_2) * I(x,y) = g(x,y;t_1) * g(x,y;t_2) * I(x,y) \ , \tag{1.6}$$

where I is an image. On the right part of equation, the order of convolution does not matter, as convolution is associative.

When computing various features for image $I(x,y)$, we often need to know a local change in intensity values for all positions (x,y). This can be achieved by taking the spatial derivative of the image. Since the image is a discretely sampled signal, we can only compute an approximation of the derivative. For example, we can take the difference between neighboring pixels.

It can, however, often be desirable to smooth the image in connection with taking the derivative, e.g. to remove nosie. It turns out that if we want to convolve the image with a Gaussian and then take the derivative, we can instead convolve the image with the derivative of the Gaussian

$$\frac{\partial}{\partial x}(I * g) = \frac{\partial I}{\partial x} * g = I * \frac{\partial g}{\partial x} \ . \tag{1.7}$$

Since we can compute the derivative of the Gaussian analytically, we get an efficient and elegant approach to computing a smoothed image derivative.

The analytic 1D Gaussian derivative is given by

$$\frac{\mathrm{d}}{\mathrm{d}x}g(x) = \frac{-x}{\sigma^3\sqrt{2\pi}}e^{-x^2/(2\sigma^2)} \ . \tag{1.8}$$

The semi-group structure also holds for image derivatives, such that we get

$$\frac{\partial}{\partial x}g(x,y;t_1+t_2) * I(x,y) = \frac{\partial}{\partial x}g(x,y;t_1) * (g(x,y;t_2) * I(x,y)) \ , \tag{1.9}$$

which implies that we can convolve with a large Gaussian derivative kernel or we can convolve with a smaller Gaussian and a smaller Gaussian derivative and get the same result.

Data For this exercise you will use an X-ray CT image of fibres `fibres_xcth.png`, shown in Figure 1.1 and Figure 1.2.

Tasks You should create your own kernel for convolving the image, but use an already implemented convolution function. You can use the following steps to create the kernel.

1. Create a variable σ for the standard deviation.
2. Compute an integer variable s that is 3-5 times σ.
3. Create an array containing the values for which the kernel should be computed. This is an array with the values $x = [-s, -s+1, ..., 0, 1, ..., s]^T$.
4. Compute a new array g with the value of the Gaussian function for each of the elements in x. $g = 1/(\sigma\sqrt{2\pi})\exp(-x^2/(2\sigma^2))$.

Once you have computed the kernel, you can verify that you got it right by plotting x against g as shown in Figure 1.3. A kernel with the derivative of the first order the Gaussian can be created in the same way, and you can see the plot in Figure 1.4.

Having a kernel, you are now ready to use it to try convolving the image, and experimentally verify the properties of convolution with Gaussians.

1. First, you should experimentally verify the separability of the Gaussian convolution kernel. You do this by convolving the test image with a 2D kernel, and convolving the same image with two orthogonal 1D kernels. The you can subtract the resulting two images and verify that the difference is very small. Note that you can get a 2D Gaussian kernel g_2 as the outer product of two 1D kernels: $g_2 = gg^T$ (g is a column vector).

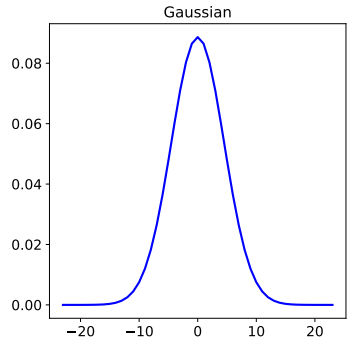

Figure 1.3: Plot of Gaussian with $\sigma = 4.5$.

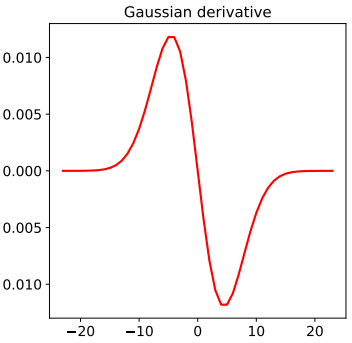

Figure 1.4: Plot of Gaussian derivative with $\sigma = 4.5$.

2. Investigate the difference between the derivative of the image convolved by a Gaussian and the image convolved with the derivative of the Gaussian as described in Eq. 1.7. When you convolve with the derivative of the Gaussian, you should be aware that you take the derivative in one direction (e.g. the x-direction), and you should smooth in the other direction with a Gaussian (not the derivative). You can compute the derivative of the image by convolving with the kernel `k = [0.5, 0, -0.5]` and then smooth with the Gaussian to get the same as above. Again, you can subtract the two images and show the difference to test if you get the same.

3. Test if a single large convolution with a Gaussian of $t = 20$ is equal to ten convolutions with a Gaussian of $t = 2$. Remember that $\sigma = \sqrt{t}$. Again you compare the two images by subtracting them and showing the difference.

4. Test if convolution with a large Gaussian derivative

$$I * \frac{\partial g(x, y; 20)}{\partial x} ,$$

is equal to convolving with a Gaussian with $t = 10$ and a Gaussian derivative with $t = 10$

$$I * g(x, y; 10) * \frac{\partial g(x, y; 10)}{\partial x} .$$

Again you compare the two images by subtracting them and showing the difference.

1.1.2 *Computing length of segmentation boundary*

Segmentation is one of the basic image analysis tasks, and we will also cover a few segmentation methods in the course. For an outcome of a segmentation, as for example shown in Figure 1.5, it may be important to measure some quality of the result. One relevant measurement is a length of the segmentation boundary.

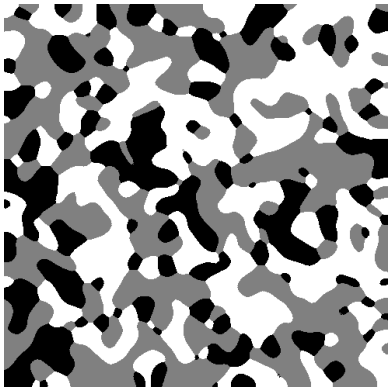

Figure 1.5: Image of a segmented fuel cell with three phases. Black represents air, grey is cathode, and white is anode.

Assume that segmentation is represented by an image $S(x, y)$ which takes n discrete values, i.e. $S : \Omega \to \{1, 2, \dots n\}$, where n is the number of segments. We define the length of the segmentation boundary as

$$L(S) = \sum_{(x,y)\sim(x',y')} d\left(S(x, y), S(x', y')\right) ,$$

where $(x, y) \sim (x', y')$ indicates two neighboring pixel locations, and d is a discrete metric

$$d(a, b) = \begin{cases} 0 & \text{if } a = b \\ 1 & \text{otherwise} \end{cases}$$

which in this case operates on pixel intensities. In other words, $L(S)$ counts all occurrences of two neighboring pixels having different labels.

Tasks

1. Compute the length of the segmentation boundary for provided segmentation images of a fuel cell, where one is shown in Figure 1.5. You can consider avoiding loops and instead using vectorization provided by Matlab or numpy, which will ensure an efficient and compact implementation.

2. Collect your code in a function which takes segmentation as an input, and returns the length of the segmentation boundary as an output. Your function will be useful when we will be working with Markov random fields later in the course.

Data In this exercise you should use the volume slice `fuel_cell_1.tif`, `fuel_cell_2.tif`, and `fuel_cell_3.tif` that you can find on Campusnet.

1.1.3 *Curve smoothing*

A segmentation boundary may be explicitly represented using a sequence of points connected by line segments, which typically delineates an object in the image. Assume that an N-times-2 matrix $\mathbf{X}$ contains x and y coordinates of N points which define a closed curve, a so-called snake [3].

[3] Michael Kass, Andrew Witkin, and Demetri Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, 1(4):321–331, 1988

To impose smoothness to this representation, we will need to smooth the curve. This can be achieved in a simple way by displacing every curve point towards the average of its two neighbors, possibly iteratively. Point displacement can be seen as a result of filtering the curve with a kernel $\lambda \begin{bmatrix} 1 & -2 & 1 \end{bmatrix}$, where λ is a parameter controlling the magnitude of the displacement. For efficiency, we want to implement the curve-smoothing step as

$$\mathbf{X}^{\text{new}} = (\mathbf{I} + \lambda \mathbf{L})\mathbf{X} \tag{1.10}$$

where $\mathbf{L}$ is a N-times-N matrix with elements 1, -2, and 1 in every row such that -2 is on the main diagonal, and 1 on its left and right (also circularly in the first and the last row), and zeros elsewhere. See Figure 1.6 for an example.

$$\mathbf{L} = \begin{bmatrix} -2 & 1 & 0 & 0 & 0 & 1 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 1 & 0 & 0 & 0 & 1 & -2 \end{bmatrix}$$

Figure 1.6: Matrix $\mathbf{L}$ for N=6. Notice value 1 in the upper right and lower left.

Using this step iteratively, where we use t for iteration number (i.e. not transpose or potential), gives

$$\mathbf{X}^{(t+1)} = (\mathbf{I} + \lambda \mathbf{L})\mathbf{X}^{(t)}\,. \tag{1.11}$$

Confirm that one step with $\lambda = 0.5$ displaces every curve point exactly to the average of its neighbors. Try smoothing one of the provided contours, also shown in Figure 1.7. We have included both original and noisy curves.

Maybe you noticed two important limitations of our simple approach. First, for larger values of λ the curve will start oscillating, but using a small λ requires many iterations of the smoothing step for a noticeable result. Second, smoothing leads to the shrinkage of the curve.

Stability issues can be avoided by evaluating the displacement on the new curve $\mathbf{X}^{\text{new}}$. In other words, we can use an implicit (backwards Euler) approach. Instead of Equation 1.10 where $\mathbf{X}^{\text{new}} = \mathbf{X} + \lambda \mathbf{L}\mathbf{X}$ we use $\mathbf{X}^{\text{new}} = \mathbf{X} + \lambda \mathbf{L}\mathbf{X}^{\text{new}}$ leading to

$$\mathbf{X}^{\text{new}} = (\mathbf{I} - \lambda \mathbf{L})^{-1}\mathbf{X}. \tag{1.12}$$

We can now choose an arbitrary large λ and obtain the desired smoothing in just one step. The price to pay is matrix inversion, but for many applications, this needs to be computed only once. While outside the scope of this course, an interested student may read on implicit smoothing of triangle meshes in an influential paper by Desbrun et al. [4].

Shrinkage is caused by the kernel which minimizes curve length. Instead, we can use a kernel which minimizes the curvature, or even better, we can weight the elasticity (length minimizing) and rigidity (curvature minimizing) term. The kernel with the two contributions is

$$\alpha \begin{bmatrix} 0 & 1 & -2 & 1 & 0 \end{bmatrix} + \beta \begin{bmatrix} -1 & 4 & -6 & 4 & -1 \end{bmatrix}$$

with α and β weighting the two terms. See Hanbook of medical imaging [5], section 3.2.4, for the derivation of the kernels.

Smoothing is now obtained as

$$\mathbf{X}^{\text{new}} = (\mathbf{I} - \alpha \mathbf{A} - \beta \mathbf{B})^{-1}\mathbf{X}, \tag{1.13}$$

where $\mathbf{A}$ is identical to $\mathbf{L}$ we used before, and $\mathbf{B}$ is a very similar matrix, but having other values on the diagonals, e.g. -6 on the main diagonal. Note that $\alpha\mathbf{A} + \beta\mathbf{B}$ is also a (sparse) circulant matrix.

Tasks

1. Implement curve smoothing as in Equation 1.10 and test it for various values of λ. Try using smoothing iteratively to achieve a visible result for small λ.

2. Implement curve smoothing as in Equation 1.12 (implicit smoothing) and test it for various values of λ. Do you need an iterative approach of this smoothing?

3. Implement implicit curve smoothing but with the extended kernel. This means that your implementation instead of $\lambda\mathbf{L}$ uses a matrix that combines elasticity and rigidity, as in Equation 1.13. Test smoothing with various values of α and β. What do you achieve when choosing a large β and small α?

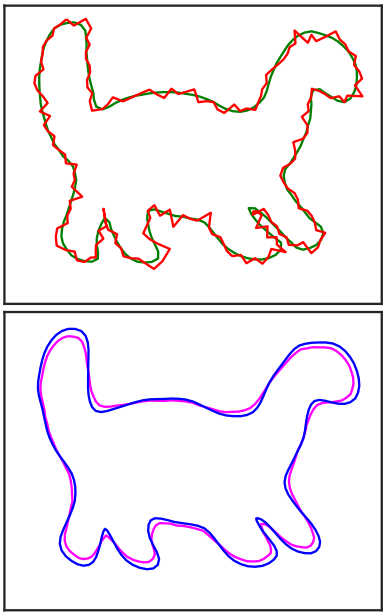

Figure 1.7: Top image shows the dinosaur curve in green, while red shows the curve with added noise. Bottom image shows two smoothing results with different α and β weights.

[4] Mathieu Desbrun, Mark Meyer, Peter Schröder, and Alan H Barr. Implicit fairing of irregular meshes using diffusion and curvature flow. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 317–324, 1999.

[5] Chenyang Xu, Dzung L Pham, and Jerry L Prince. Image segmentation using deformable models. *Handbook of medical imaging*, 2:129–174, 2000a

4. Implement a function which returns a smoothing matrix needed for implicit smoothing with the extended kernel. You will be using this when working with deformable models later in the course.

Data In this exercise you should use the curves given as text files containing point coordinates `dino.txt`, `dino_noisy.txt`, `hand.txt`, and `hand_noisy.txt`.

1.1.4 *Optional: Total variation*

In many image analysis applications, such as image denoising and image segmentation, we are interested in producing a result which has a quality that we loosely call *smoothness*. A common way of estimating smoothness is by considering a total variation defined for an image I as

$$V(I) = \sum_{x \sim x'} |I(x) - I(x')|,$$

where $x \sim x'$ indicates two neighboring pixel locations. We expect smooth images to have a low total variation.

Implement a function which computes the total variation of a 2D grayscale image and test it on the image shown in Figure 1.1 and Figure 1.2. Use Gaussian smoothing to remove some of the noise from the image, and confirm that the smoothed image has a smaller total variation.

Data In this exercise you should use the volume slice `fibres_xcth.png` that you can find on Campusnet.

1.1.5 *Optional: Unwrapping image*

A solution to image analysis problem may involve geometric transformations. When working with spherical or tubular objects, we sometimes want to represent an image in polar coordinate system. Implement a function which performs such *image unwrapping* using a desired angular and radial resolution. Use your function to unwrap one of the slices from the dental data set, an example is shown in Figure 1.8 and 1.9. Unwrapping will be useful when we will be working with deformable models later in the course.

Data In this exercise you should use one of the central slices from the `dental` folder that you can find on Campusnet.

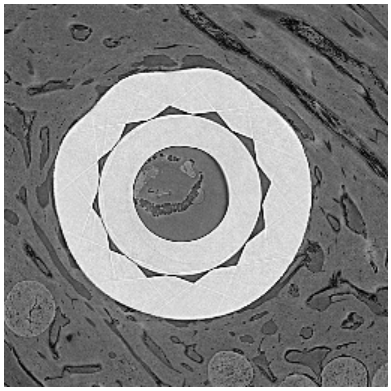

Figure 1.8: Image of a dental implant that should be unwrapped.

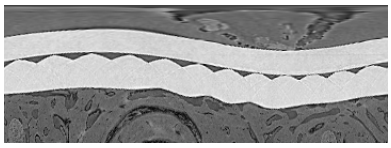

Figure 1.9: Unwrapped image of a dental implant.

1.1.6 *Optional: Working with volumetric image*

To give you a taste of working with 3D images, we have prepared a small data set containing slices from an X-ray CT scan. By convention

in X-ray imaging, dense structures (having a height X-ray attenuation) are shown bright compared to less dense structures. Furthermore, the direction given by image slices is most often denoted z. The volume you are given contains a metal (very bright) object. Show orthogonal cross sections of the object, see Figure 1.10. Can you determine an optimal threshold for segmenting the object from the background?

Optionally, show a volumetric 3D rendering of the thresholded object using any available software. An example is shown in Figure 1.11. If you are using MATLAB, check a function `isosurface`.

Data In this exercise you should use the volumetric image stored as individual slices in the folder called `dental` that you can find on Campusnet.

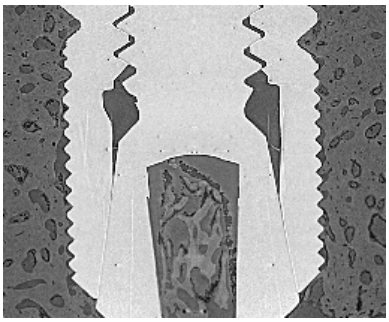

Figure 1.10: A longitudinal slice (an xz-plane) of the volumetric image of a dental implant.

Figure 1.11: 3D rendering of the thresholded volumetric image of a dental implant.

1.1.7 *Optional: PCA of multispectral image*

Principal component analysis (PCA) is a linear transform of multivariate data that maps data points to an orthogonal basis according to maximum variance. A basic introduction in PCA is given in [6], and here we will apply it on a multispectral image.

We provided an image acquired with the VideometerLab, which is a multispectral imaging device, that uses coloured LED's to illuminate a material, in this case samples in a petri dish. This gives an 18 channel image $I : \Omega \subset \mathbb{R}^2 \rightarrow \mathbb{R}^{18}$ where each channel corresponds to a wavelength, and channels cover the range from 410 nm to 955 nm, i.e. the visible and near-infrared spectrum. The image depicts vegetables on a dish and is shown in false colours in Figure 1.12.

The aim of this exercise is to carry out PCA and visualise the principal components as images. PCA can be done by eigenvalue decomposition of a data covariance matrix. In our analysis we view each pixel as an observation, so we rearrange I into a N-by-18 data matrix $\mathbf{X}$. Each row of $\mathbf{X}$ represents one pixel (observation), with the successive columns corresponding to wavelengths (variables).

Data covariance matrix $\mathbf{C}$ is defined as

$$\mathbf{C}_{ij} = \frac{1}{N-1} \sum_{n=1}^{N} (\mathbf{X}_{ni} - \mu_i)(\mathbf{X}_{nj} - \mu_j), \tag{1.14}$$

where $i, j \in \{1, ..., 18\}$, and μ is a 18 dimensional empirical mean vector computed for each variable.

Covariance matrix $\mathbf{C}$ can be computed as a matrix product

$$\mathbf{C} = \frac{1}{N-1} \overline{\mathbf{X}}^T \overline{\mathbf{X}}, \tag{1.15}$$

where $\overline{\mathbf{X}} = \mathbf{X} - \mathbf{1}_{n \times 1} \mu^T$ is a zero-mean matrix obtained by independently centering each row of $\mathbf{X}$ around its mean value. Convince yourself that is correct.

[6] Lindsay I Smith. A tutorial on principal components analysis. Technical report, 2002

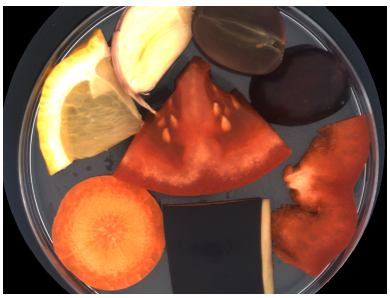

Figure 1.12: False color image obtained from an 18 band VideometerLab image.

Principal components are given by the eigenvectors of $\mathbf{C}$, e.i. vectors such that $\mathbf{C}\mathbf{v}_i = \lambda_i \mathbf{v}_i$. Eigenvector corresponding to the largest eigenvalue gives the direction of the largest variance in data, the eigenvector corresponding to the second largest eigenvalue is the direction of the largest variance orthogonal to the first principal direction, etc. The projections of the data points onto principal directions $\hat{\mathbf{X}}\mathbf{v}_i$ can be rearranged back into image grid, and viewed as images.

If $\mathbf{V}$ is a matrix containing eigenvectors in it columns, all principal components can be computed as

$$\mathbf{Q} = \overline{\mathbf{X}}\mathbf{V}. \tag{1.16}$$

Data In this exercise you should use the images in the folder called `mixed_green` which contains png-images. You find the file on Campusnet.

Suggested approach The following steps takes you through computing the principal components.

1. Write a script to read in the images and display them. Convince yourself that there is a difference between the spectral bands. Make sure to change the data type to float or double.
2. Rearrange the image into a matrix $\mathbf{X}$ as described above with one pixel in each row. Compute the column-wise mean μ and subtract this from $\mathbf{X}$ to get the zero mean $\overline{\mathbf{X}}$.
3. Compute the covariance matrix $\mathbf{C}$.
4. Compute the eigenvectors $\mathbf{V}$ and eigenvalues λ.
5. Compute the principal component loadings $\mathbf{Q}$.
6. Rearrange $\mathbf{Q}$ into images and display the result.

You can compare your implementation to an already implemented PCA function in e.g. MATLAB or Python.

1.1.8 *Optional: Bacterial growth from movie frames*

Image data with a temporal component can be stored in the form of a movie. The purpose of this exercise is to read in image frames from a movie and analyze them. The movie contains microscopic images of listeria bacteria growing in a petri-dish acquired at equal time steps. An example frame is shown in Figure 1.13. Your task is to make a small program that visualize bacterial growth by counting the cells.

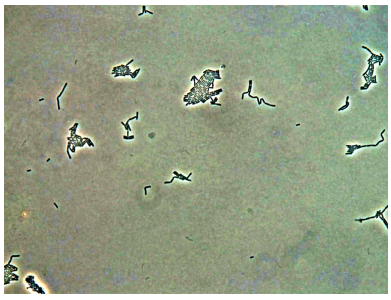

Figure 1.13: Example of microscopic image of listeria bacteria in a petri dish.

The image quality is however not very good due to the low resolution and compression artifacts, making it difficult to separate the individual

bacteria. So, we make a rough assumption that the number of pixels covered by bacteria is proportional to the number of bacteria. The task is therefore to make a plot of the number of pixels covered by bacteria as a function of time.

Data In this exercise you should use the movie `listeria_movie.mp4` that you can find on Campusnet.

Suggested approach You can for example first read in one representative frame from the movie and build an cell segmentation method. A simple threshold is not sufficient, but with a few processing steps the cells become distinguishable from the background. You can try the following steps:

1. Convert the image I to a grey scale image G.
2. Compute the gradient magnitude $M = \sqrt{(\partial G/\partial x)^2 + (\partial G/\partial y)^2}$ using an appropriate filter.
3. Smooth M using a Gaussian filter.
4. If the parameters have been chosen appropriately, the pixels covering bacteria can now be segmented by thresholding.

When you have made a functioning segmentation model, you can apply this to all images in the movie using the following steps for each image in the movie:

1. Apply the segmentation and sum the bacteria pixels.
2. Store this number in an array.
3. Plot the number of pixels as a function of time.

The obtained curve has a characteristic shape. Can you recognize the function that could describe this shape?

Part I

Feature-based image analysis

Analyzing images using feature-based representations is central to many applications. We have chosen three topics that we will cover. First we will work with scale-space for detecting image features independently of scale. Specifically, we will focus on scale-space blob detection. Then we will investigate features as a basis for pixel classification used for image segmentation. Finally, we will work with feature-based image registration.

2 Scale-space

Methods from scale-space allow scale invariant detection of image structures. This means that we can find features like blobs (binary large objects), corners, ridges, edges, and other structures at different scale. When we talk about image features like corners and edges, it is not corners or edges of the physical depicted objects, but corners and edges in the image intensities. To visualize this, you can think of a 2D image as a landscape, with pixel intensities corresponding to height measurements at regularly placed positions. In this landscape, an edge is a line where high abruptly changes. A corner will be a height-change point where two (more or less) orthogonal edges meet, and other types of features can be described in the same way.

Using a feature-based image representation is convenient, because we break the image up into manageable parts that are more descriptive than the individual pixels. Scale invariance, which means that we characterize (make a mathematical description) the same feature shown at different scale in two images, is also very convenient. In e.g. computer vision where images of the same object are often captured from difference distance, it is typically a desired property to be able to measure the features independent of its scale. But it also allows us to measure image structures that are different in size for example from microscope images, as we will be working with here.

Here we will base our work on the article of Lindeberg[1] that gives an introduction to scale-space theory. Scale-space representation has made the basis for a range of image analysis methods and is extensively used in computer vision. In the exercise you will implement scale-space blob detection and use it for detecting and measuring the size of fibres that are imaged using X-ray CT.

[1] Tony Lindeberg. Scale-space: A framework for handling image structures at multiple scales. 1996

The computation of scale-space is done by representing image features at all scales at once and detect features based on criteria that is independent of the scale. We will work with the Gaussian scale-space, and the analysis is in practice done by smoothing the image using a Gaussian filter. In Lindeberg[2] the scale-space representation is defined for a general N-dimensional signal $f : \mathbb{R}^N \to \mathbb{R}$, but we will use it

[2] Tony Lindeberg. Scale-space: A framework for handling image structures at multiple scales. 1996

for a 2D image $I : \mathbb{R}^2 \to \mathbb{R}$. For a 2D image, its Gaussian scale-space representation is $L : \mathbb{R}^2 \times \mathbb{R}_+ \to \mathbb{R}$, which in practice becomes a 3D object, with the two spatial image dimensions (x, y) and the scale in the third dimension. Since scale is obtained by smoothing with a Gaussian, the variable determining the degree of smoothing is the variance t. Also the standard deviation $\sigma = \sqrt{t}$ is used in the article, but here we have simplified the notation and use only the variance t.

The Gaussian scale-space L is defined for N-dimensional signals by

$$L(x;t) = \int_{\xi \in \mathbb{R}^N} f(x-\xi) g(\xi;t) d\xi \tag{2.1}$$

with $g : \mathbb{R}^N \times \mathbb{R}_+ \to \mathbb{R}$ being the N-dimensional Gaussian kernel

$$g(x;t) = \frac{1}{(2\pi t)^{N/2}} e^{-(x_1^2 + \cdots + x_N^2)/(2t)}. \tag{2.2}$$

In practice we will work with the Gaussian scale-space for 2D images on a discrete set of pixels. Therefore, we can write the Gaussian scale-space (ignoring boundary issues) as

$$L(x,y;t) = \sum_{-\gamma}^{\gamma} \sum_{-\delta}^{\delta} I(x-\gamma, y-\delta) g(\delta, \gamma; t) \tag{2.3}$$

where $g : \mathbb{R}^2 \times \mathbb{R}_+ \to \mathbb{R}$ is the 2D Gaussian kernel

$$g(x,y;t) = \frac{1}{2\pi t} e^{-(x^2+y^2)/(2t)}. \tag{2.4}$$

Computing the scale-space is done at a discrete set of steps, and we have the start condition with $t = 0$ defined as $L(x, y; 0) = I(x, y)$.

For feature detection, we need to compute the derivatives of a scale-space representation. Note that this is conveniently achieved by convolving an image with a kernel that is a derivative of a Gaussian. Blob detection uses second order derivatives, more precisely the Laplacian of a Gaussian $\nabla^2 L = L_{xx} + L_{yy}$ which gives a high response where there is a blob in the image. To detect blobs, we need to find local maxima and minima of the Laplacian. Some local maxima and minima will, however, be very weak and they should not be detected as a blob. Therefore, low responses of the Laplacian of the Gaussian should not be included. These low-response blobs are excluded by not including blobs that have an absolute Laplacian response lower than a certain threshold.

What we still need to do is to ensure that we can detect blobs across different scales. The image in scale-space representation is increasingly smoothed, and with increasing scale t, pixels will change their intensity value towards the average value of the image. Therefore, the absolute values of derivatives will become smaller when increasing t. For blob

detection, this means that the magnitude of the local maxima and minima in the scale-space of the Laplacian $\nabla^2 L$ will decrease and this smoothing must be compensated. The compensation factors for different features are given in Lindeberg[3] and for the blob feature it is t such that the scale normalized Laplacian of Gaussian is $t\nabla^2 L$.

[3] Tony Lindeberg. Scale-space: A framework for handling image structures at multiple scales. 1996

2.1 *Exercise 2 – part I, Scale-space blob detection*

In this part of the exercise you will implement scale-space blob detection with the purpose of detecting and measuring glass fibres from images of a glass fibre composite. An image example is given in Figure 2.1, that shows a polished surface of a glass fibre composite sample, where individual fibres can be seen. Since these fibres are relatively circular we will model them as circles. This means that we must find their position (center coordinate) and diameter, and for this we will use the scale-space blob detection. After having computed the fibres parameters, we will carry a statistical analysis of the results.

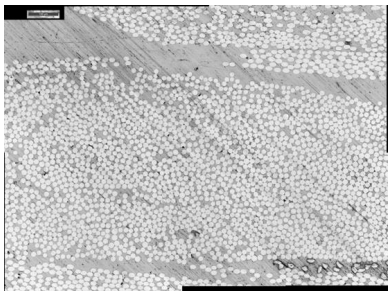

Figure 2.1: Example of fibre image acquired using an optical microscope.

2.1.1 *Computing Gaussian and its second order derivative*

We will approach this analysis in steps that lead to the final algorithm. First we will use synthetic data to develop and test our algorithm, and after that we will carry out the analysis on the real images.

Since we focus on blob detection, we must have a Gaussian kernel and its second order derivative. Some convolution libraries have already implemented the second order derivative of a Gaussian that you are welcome to use for the exercise. But we will anyhow start investigating the second order derivative of a Gaussian, which you will be using for blob detection.

The Gaussian is separable and we can employ 1D filters for our analysis, which you will compute now. The 1D Gaussian is given by

$$g(x) = \frac{1}{\sqrt{2\pi t}} e^{\frac{-x^2}{2t}} \ . \tag{2.5}$$

Suggested procedure

1. Derive (analytically) the second order derivative of the Gaussian

$$\frac{\mathrm{d}^2 g}{\mathrm{d}x^2} \ .$$

2. Implement a function that takes the variance t as input and outputs a filter kernel of g and $\mathrm{d}^2 g/\mathrm{d}x^2$. You should use a filter kernel with a size of at least $\pm 3\sqrt{t}$. Why? (*Hint:* Set a variable $\nu = \lceil 3\sqrt{t} \rceil$, make an array with the integer values $[-\nu, -\nu+1, ..., \nu-1, \nu]$, and compute the Gaussian on these values.)

3. Try the filter kernel on the synthetic test image `test_blob_uniform.png` and inspect the result.

2.1.2 *Detecting blobs at one scale*

Here you going to implement a function to detect blobs at a single scale. Blobs can be found as spatial maxima (dark blobs) or minima (bright blobs) of the scale-space Laplacian

$$\nabla^2 L = L_{xx} + L_{yy} \ . \tag{2.6}$$

Suggested procedure

1. Compute the Laplacian at one scale using the synthetic test image `test_blob_uniform.png`.
2. Create a function that detects the coordinates of maxima and minima in the Laplacian image (detect blobs), and that has an absolute value of the Laplacian larger than some threshold.
3. Plot the center coordinates and circles outlining the detected blobs. The radius of the circles should be $\sqrt{2t}$.
4. Try varying t such that the blobs in `test_blob_uniform.png` are exactly outlined.

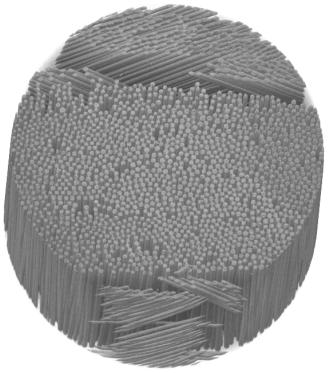

Figure 2.2: Visualization the 3D fibers scanned with the high resolution X-ray CT-scanner.

2.1.3 *Detecting blobs on multiple scales*

Now you should extend the blob detection at a single scale to multiple scales. To find blobs at multiple scales, we must use the scale-space representation. This can conveniently be done by representing $\nabla^2 L$ as a 3D array (volumetric image).

Suggested procedure

1. Decide on scales at which the Laplacian must be computed. A good idea is to make it equal steps in t. Remember that the radius of the blobs are $\sqrt{2t}$, so you can look at the size of the structures that you want to detect to decide a good range of scales.
2. Compute the scale normalized scale-space Laplacian $t\nabla^2 L$ for the test image `test_blob_uniform.png`. It is very important that you remember to scale normalize, i.e. that you multiply the Laplacian $\nabla^2 L$ by t to be sure to detect the correct scales.
3. Find coordinates and scales of maxima and minima in this scale-space and plot the detected blobs on top of the image. What are the detected scales and what is the diameter of the blobs?

4. Detect blobs in the test image `test_blob_varying.png`.

5. Verify that you detected the blobs at the correct scale by showing an image where you plot circles with a diameter of $\sqrt{2t}$ on top of the detected blobs.

2.1.4 *Detecting blobs in real data*

We will now continue with the real images of fibers. The fibre data is obtained using different scanning methods including scanning electron microscopy (`SEM.png`), optical microscopy (`Optical.png`), synchrotron X-ray CT (`CT_synchrotron.png`), and three resolutions of laboratory X-ray CT (`CT_lab_high_res.png`, `CT_lab_med_res.png`, `CT_lab_low_res.png`). The CT data is a single slice very close to the top, so we assume the data to be from the same part of the sample, and this allows us to directly compare the fibers. We will do this comparison in next exercise, but in this we will compute the fiber location and their diameter. In Figure 2.2 you can see a visualization of the fibre data from the high resolution X-ray CT scan.

We start by testing the blob-detection on this real data.

Suggested procedure

1. Run your blob-detection function from above on a cut-out example of one of the images. It is important that you tune your parameters to get the best possible results.

2.1.5 *Localize blobs*

It turns out, that it is difficult to detect blobs in the Laplacian scale-space in the fiber image, such that all fibers are found. To overcome this, we will detect the fibers as maxima in a Gaussian smoothed image. Since the fibers are almost the same size, we can use a single scale of the Gaussian to detect the fiber centers.

Suggested procedure

1. Smooth an image of fibers with a Gaussian and visualize the result.

2. Find locations of maxima in this image and plot the positions on top of the original image.

3. Compute the Laplacian scale-space for the image.

4. Find the scale of each fibre as the minimum over scales at the fiber locations.

5. Plot circles according to the found scale on top of the original image.

6. Detect fibers in all six fiber images. Save the locations and diameters.

In the exercise in Week 4, where you will work with feature-based image matching, you will use the results obtained in this exercise. So, it will be possible to continue working on the parts that you did not finish here in Week 4.

3 Feature-based segmentation

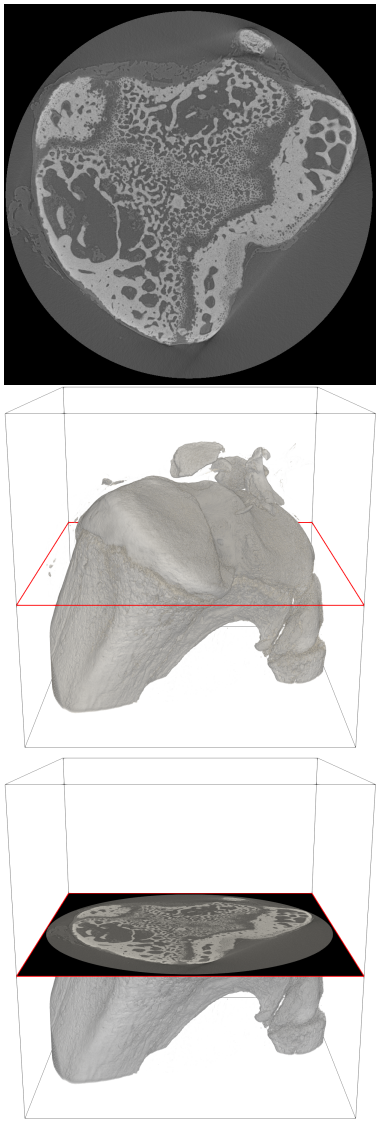

Figure 3.1: CT-scan showing the trabecular structure of a mouse bone. Top is the image slice, middle shows the bone in 3D and bottom shows where the slice is taken in the image volume.

Image segmentation is the process of partitioning an image into regions. The result of segmentation can be represented as a function $g(x, y) \rightarrow \ell$ that maps each position (x, y) in the image I to a label $\ell \in 1, \ldots, n_l$. In practice this means that we need a way to compute a label for every image pixel. As for representing the result of the segmentation using labels, this is just one choice. In some other approaches, the segmentation may be represented differently, for example using curves delineating boundaries of segments. Before talking about the segmentation method, we will discuss the reasons for segmenting an image.

Segmentation is often done to visualize certain image structures, to measure some quantities in the image, or both. Let's consider an example in Figure 3.1 that shows a cross section of a tibia bone from a mouse, acquired using CT-scanning. If we were given the task to segment this image, we would first need to consider what the desired outcome is, i.e. to ask the question: 'What would be the optimal segmentation?' The bone image can be segmented in many ways, and to decide on the desired segmentation outcome you need to consider what you want to visualize, quantify, or otherwise use in your analysis.

Here, the mouse bone has been imaged as a part of a project set to investigate how osteoporosis affects bone growth. To visualize and quantify effects of the osteoporosis, tibia bones from mice with and without osteoporosis were imaged. We know that osteoporosis makes bones weaker, and now we can try to translate this to something we can measure by segmenting the image. We could e.g. measure the area of the image depicting bone by counting the pixels that are labeled as bone. This is illustrated in Figure 3.2 in the first segmentation where the image is segmented into bone and background. Total bone area can here be obtained by counting the white pixels, and bone fraction is the ratio between the number of bone pixels divided by all pixels. Yet another measure we could obtain is the boundary length of the interface between background and bone. The finer the bone structures are, the longer this interface would be and therefore this is a good

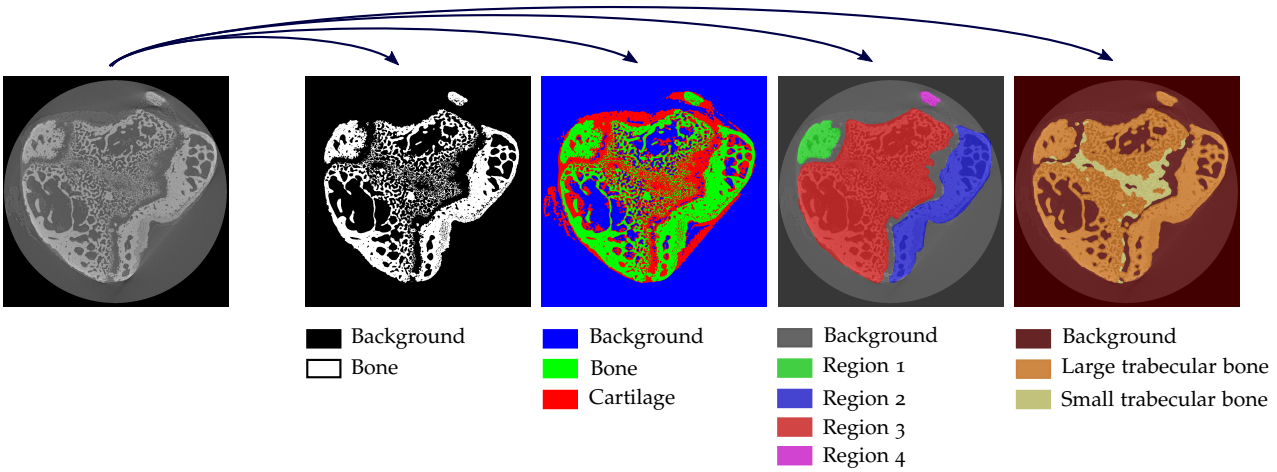

Figure 3.2: Slice of a CT-scan of a trabecular mouse bone that has been segmented based on three different criteria. The left image is the original. The next image in black and white is a segmentation into bone and background. The third image in red, green, and blue is a segmentation into bone, cartilage, and background. The fourth image is a segmentation into four separate bone regions, and the last image in brown colors is a segmentation based on the size of the trabecular structures in the bone (size of bone and holes).

measure for the bone structure. We could also compute the distribution of bone thickness or other measures that are related to the problem we are investigating namely the effect of osteoporosis.

The second segmentation is similar to the first, but here also the areas of the image with cartilage has been labeled. In the third segmentation example, the image of the bone has been separated into four regions and background. And in the final example, the bone has been segmented based on the coarseness of the bone, such that the finer structures are in one region and coarser structures are in another. In all these examples, the segmentation will allow computation of areas, shapes, lengths, thicknesses, and other measures that could be relevant for the investigation. But to emphasize the important point; the aim of segmenting an image is a choice that is determined by the problem we try to solve.

With this in mind, we can consider the segmentation method, i.e. how to compute pixel labels given intensities of all pixels. A good rule is to choose the simplest method that gives a satisfying result. In some cases you might want to do manual segmentation, i.e. labeling image regions by drawing them using a paint program or by choosing a semi-manual segmentation method that gives some level of automation. In this chapter, we will focus on automated segmentation methods, but it is important to keep in mind that manual labeling can be part of a segmentation process, e.g. to create a mask over a region of interest or to correct areas of faulty segmentations. Having the human as part of the segmentation loop are used in research areas such as *interactive segmentation* or *active segmentation*, which addresses the issue on deciding what should be segmented.

For automated segmentation, the simplest labeling is obtained by classifying each pixel according to only its own intensity. In a gray-scale image, this will typically be one or more threshold values that

separate the pixel intensities into groups. For many image segmentation problems, this is not sufficient for a satisfactory segmentation. In the bone example in Figure 3.2, the first image is segmented using an intensity threshold. But to automatically segment the image based on e.g. the trabecular structures as in the last image requires more than just the pixel intensity, namely one needs to also account for contextual image information. Contextual information means that we want to account for image patterns of the depicted structures.

Instead of using only the intensity of the pixel, when computing the label, we can use the information from the neighborhood around a pixel. Hereby, we capture the local appearance of the image, and we will try to use that information for labeling each pixel in the image. The local appearance is also known as image texture.

3.1 *Supervised feature-based segmentation*

We will approach the segmentation as a supervised labeling problem, where we learn the parameters of the segmentation model from training data. The training data consists of one or more labeled training images, that are created by manually annotating the images. You can think of this segmentation approach as of transferring the labels from the training image to an unknown test image. Here, the training and test images are of the same type, meaning having a similar appearance. The choice of what we want to segment is now made, since we are given the labeled training data.

Supervised labeling is also typically what you would do when working with deep learning using convolutional neural networks, which we cover later in this note. But for now, we will work with feature classification, where we have designed the features ourselves, and only classification of the features is using the training data. Using chosen features, as we will do now, has some advantages, since the features are easy to compute and do not require training. As it turns out, we can typically obtain good segmentation results with limited training data. The principle of supervised feature-based segmentation is what is used in tools like Ilastik[1] and the Trainable WEKA Segmentation Tool[2].

[1] Stuart Berg, Dominik Kutra, Thorben Kroeger, Christoph N Straehle, Bernhard X Kausler, Carsten Haubold, Martin Schiegg, Janez Ales, Thorsten Beier, Markus Rudy, et al. Ilastik: interactive machine learning for (bio) image analysis. *Nature Methods*, 16(12):1226–1232, 2019

[2] Ignacio Arganda-Carreras, Verena Kaynig, Curtis Rueden, Kevin W Eliceiri, Johannes Schindelin, Albert Cardona, and H Sebastian Seung. Trainable weka segmentation: a machine learning tool for microscopy pixel classification. *Bioinformatics*, 33(15):2424–2426, 2017

3.1.1 *Image features*

We will start with image features. In this context, we consider so-called dense image features, which means that features are computed for every pixel in the image. A feature for every pixel is a vector of a certain length, say k. The idea is that pixels originating from a similar texture also have similar feature vectors. To store features for all image pixels, we can choose to construct an array of size $r \times c \times k$, where r

and c are dimensions of the image (number of rows and columns).

In this exercise we will compute two types of image features. The first type of feature is obtained by computing a set of image derivatives by convolving the image with Gaussian kernel and its derivatives. The second type of features is obtained by collecting pixel intensities from the patches centered on a pixel. There are many other features that we could choose, e.g. SIFT[3], SURF[4] or ORB[5] features. These features successfully characterize the contextual appearance of the image, and would therefore be useful for segmentation. But they have been developed for detecting and matching interest points, and therefore they are relatively complex to compute. Instead, for this exercise we chose relatively simple features that are easy to compute, and that still give good performance.

[3] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2): 91–110, 2004

[4] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *European conference on computer vision*, pages 404–417. Springer, 2006

[5] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *2011 International conference on computer vision*, pages 2564–2571. Ieee, 2011

Features from a Gaussian and its derivatives Building on what we worked with in the previous chapter, we use image smoothing by convolving with a Gaussian kernel and its derivatives for computing the segmentation features.

The Gaussian features are obtained as a stack of Gaussian derivatives. Let us use the same notation of Gaussian derivatives as we did in the scale-space exercise, such that we have

$$L = (I * g(x,t)) * g(x,t)^T \,,$$

where L is an image I convolved with the 1D Gaussian

$$g(x,t) = \frac{1}{\sqrt{t2\pi}} e^{\frac{-x^2}{2t}}$$

at scale t, where $t = \sigma^2$. We will use a short notation for the Gaussian derivative

$$g_x = \frac{\partial g}{\partial x} \,,\; g_{xx} = \frac{\partial^2 g}{\partial x^2} \,,$$

etc. Then we get the images by convolving with Gaussian derivatives as

$$L = I * g * g^T \,,\; L_x = I * g_x * g^T \,,\; L_y = I * g * g_x^T \,,$$
$$L_{xx} = I * g_{xx} * g^T \,,\; L_{xy} = I * g_x * g_x^T \,, \ldots$$

etc. In this exercise, we will compute the higher order derivatives until the fourth order, which will result in a 15-dimensional descriptor that captures the local appearance of the image. The descriptor is formed by stacking the Gaussian convolved images, such that each pixel position has an associated 15-dimensional feature vector. That is, we have a $r \times c \times 15$ feature image

$$\begin{aligned} F \;=\; & [L,\, L_x,\, L_y,\, L_{xx},\, L_{xy},\, L_{yy},\, L_{xxx},\, L_{xxy},\, L_{xyy},\, L_{yyy}, \\ & L_{xxxx},\, L_{xxxy},\, L_{xxyy},\, L_{xyyy},\, L_{yyyy}] \,. \end{aligned}$$

Since the response of the Gaussian derivatives becomes smaller with increasing order, we will normalize the layers in the feature image with the standard deviation of the feature image

$$\dot{L}(x,y) = \frac{L(x,y)}{\text{std}(L)} ,$$

where $\text{std}(L)$ is the standard deviation of the image L. We get the feature descriptor as

$$\begin{aligned} \dot{F} \quad = \quad & [\dot{L},\, \dot{L}_x,\, \dot{L}_y,\, \dot{L}_{xx},\, \dot{L}_{xy},\, \dot{L}_{yy},\, \dot{L}_{xxx},\, \dot{L}_{xxy},\, \dot{L}_{xyy},\, \dot{L}_{yyy}, \\ & \dot{L}_{xxxx},\, \dot{L}_{xxxy},\, \dot{L}_{xxyy},\, \dot{L}_{xyyy},\, \dot{L}_{yyyy}] . \end{aligned}$$

Multi-scale features Segmentation results will improve by computing features at more than one scale and combining them. Therefore, you may consider stacking features computed at multiple scales for better segmentation. If we call a feature computed at a scale t_1 for $\dot{F}_{t_1}$, we can compute a multi-scale feature as

$$\dot{F}_{\text{multi}} = [\dot{F}_{t_1}, \dot{F}_{t_2}, \dot{F}_{t_3}] ,$$

here with the three sclaes t_1, t_2, t_3, which will result in a 45-dimensional feature vector for each image pixel.

Patch-based features Another way of computing image features is by extracting small patches centered on a pixel and concatenating the pixels into a feature vector. If we e.g. have a 9×9 image patch, this will result in an 81 dimensional feature vector for each pixel position. Similar to the Gaussian features, this can be computed in a multi-scale fashion.

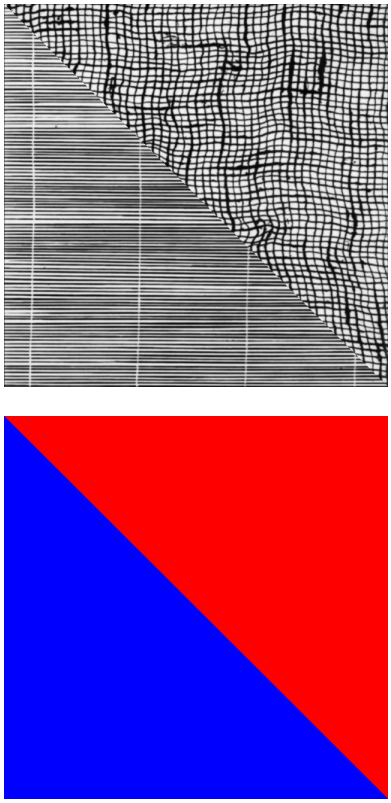

Figure 3.3: A training image with ground truth labeling. Red is one label and blue is another. Typically, this will be two scalar values, e.g. $\{0, 1\}$.

3.1.2 *Probabilistic clustering-based segmentation*

The basic idea in our feature-based segmentation model is that parts of the image that have similar appearance should have the same label. Since the features encode the local appearance of the image, it means that we want to give the same label to features that are similar.

Using an already labeled image, we can learn the desired labels of the features. In practice, we typically have labels given as a separate image. This is shown in Figure 3.3 for one of the two-label images that you will work with in this exercise. Since the training image and the ground-truth label image are of the same size, it is easy to look up the label at a given position in the image.

By computing image features in both a training and a test image, we can transfer the labels from training image to the test image. This is

done by using a similarity measure of the image features, and here we will use Euclidean distance

$$d(f_p, f_q) = \sqrt{\sum_{i=1}^{n} (f_p(i) - f_q(i))^2} \,,$$

where f_p and f_q are two n-dimensional feature vectors.

A direct approach for labeling the image would be to compute the Euclidean distance from each feature vector in the test image to each feature vectors in the training image (or a random subset of the features in the training image). By selecting the label of one or more of the nearest feature vectors, we could obtain a labeling. If we select more than one feature, we can e.g. label according to majority vote. This would be using k-nearest neighbor classifier, which might not always be robust, because outliers can introduce noise.

A more robust approach is using k-means clustering of feature vectors, where we use cluster centers as representative feature vectors. This is sometimes referred to as a dictionary-based approach, and each cluster center is referred to as a *visual word*, while the collection of cluster centers make up a dictionary. Each of the clusters is made up of a number of feature vectors from the training image. This allows us to compute the probability of a given label λ for each feature cluster C as

$$p_C(\lambda) = \frac{\text{\# elements from } C \text{ with label } \lambda}{\text{\# elements in } C} \,, \tag{3.1}$$

which can be written as

$$p_C(\lambda) = \frac{1}{|C|} \sum_{f \in C} \delta\left(\ell(f) - \lambda\right) \,, \tag{3.2}$$

where $\ell(f)$ is the label of feature element f, and

$$\delta(x) = \begin{cases} 1 & \text{if} \quad x = 0 \\ 0 & \text{otherwise} \end{cases} \,. \tag{3.3}$$

A visual dictionary typically has a much larger number of elements than labels in the segmentation problem. In the exercise we e.g. suggest having 100-1000 elements. The local appearance of the image belonging to the same segment may vary significantly, and despite this variation, we want to assign the same label to pixels in the same region. By having many 'visual words', i.e. cluster centers of the features, we can assign the same label to features even though they might be far from each other in feature space.

After having assigned a label probability to each 'visual word' (feature cluster C), we can compute the pixel-wise label of a new image by the following steps

- create an empty label image, P, of size $r \times c \times n_l$,

- compute features, $\dot{F}_{\text{multi}}$, for each pixel i,
- for each pixel position, (x_i, y_i), find the nearest cluster center C (in feature space),
- set the label probabilities of the nearest cluster center into P at pixel position (x_i, y_i).

Hereby, you obtain an image of label-wise probabilities.

3.2 *Exercise*

You should implement a probabilistic dictionary-based segmentation using Gaussian features and k-means clustering. If time allows, you can try the same approach for segmentation, but using image patches instead of Gaussian features. The exercise consists of the following steps (explained in details below):

(A) Compute features

(B) Prepare labels for clustering

(C) Build dictionary

(D) Assign dictionary to test image

(E) Compute probability image and segmentation

There are two sets of artificially composed textured images available for training and testing in the two folders `2labels` and `3labels` that you can use for testing your implementation. Furthermore, there is a set of images of a bone with corresponding labels. Finally, there is a set of electron microscopy images of cell membranes with 30 images with ground truth. Only the training images have corresponding labels because the data set was prepared for a competition (ISBI Cell Tracking Challenge, 2012[6]). But you can train on one of these and test on one or more of the other training images.

[6] Ignacio Arganda-Carreras, Srinivas C Turaga, Daniel R Berger, Dan Cireşan, Alessandro Giusti, Luca M Gambardella, Jürgen Schmidhuber, Dmitry Laptev, Sarvesh Dwivedi, Joachim M Buhmann, et al. Crowdsourcing the creation of image segmentation algorithms for connectomics. *Frontiers in neuroanatomy*, 9:142, 2015

3.2.1 *(A) Compute features*

To ensure that you have time to complete the exercise, we have prepared the function for computing the Gaussian feature image. It is available as the functions `get_gauss_feat_multi.m` for MATLAB and `get_gauss_feat_multi` function in the `feature_based_segmentation.py` file for Python. Its functionality is described in the help text of the functions. You should start by computing these features and visually inspect what they look like. You will start working with the training image.

Suggested procedure

1. Read in the image and display it.
2. Compute the feature image.
3. Inspect the feature image by displaying the layers. You can sample a few and look at them one at a time or you can display multiple images at once.
4. Since we will be clustering the features, you can transform the feature image of size $r \times c \times 15n$, where r is the rows, c is the columns, and n is the number of scales, into a 2D array of size $rc \times 15n$ where each row is a feature vector.

3.2.2 *(B) Prepare labels for clustering*

The label image stores the label information as unique intensity values. Since we want to use the labels for computing label probabilities, we must create a representation of the label image that can be used for this. In the exercise you will work with two and three labels, but let the number of labels be n_l. n_l is the number of unique labels in the label image, but let us assume that the values in the label image is $[0, ..., n_l]$. Then we construct a new image that we call $\mathcal{L} \in \mathbb{R}^{r \times c \times n_l}$, that stores label probabilities. This means that we in each pixel of $\mathcal{L}$ has the value one in the dimension of the label and zero in the rest. You should transform the training label image to a label probability image.

3.2.3 *(C) Build dictionary*

You should use k-means clustering for building the dictionary. Using all feature vectors for building the dictionary is very time consuming, and it is sufficient to select a random subset of features. It is important that the subset is chosen randomly to be representative for the training image. You should both sample features and the corresponding image labels, i.e. labels should be sampled from the same pixel positions as the features. The labels are used for computing the label probabilities of the clusters according to Eq. 3.2 and 3.3.

Suggested procedure

1. Select a random subset of feature vectors with corresponding labels (you can use random permutation). If you choose e.g. 5000-10000 vectors, it should be sufficient for clustering.
2. Use k-means to cluster the feature vectors into a number of clusters. You can choose e.g. 100-1000 clusters.

3. Make an $n_l \times n_c$ array to store label probabilities, where n_c is the number of cluster centers. Compute the probability of a cluster belonging to each of the labels using Eq. 3.2 and 3.3, and store the probabilities in the array.

3.2.4 *(D) Assign dictionary to test image*

You now have a dictionary that can be used for segmentation in the form of cluster centers with label probabilities. You can now assign each pixel in your test image to the nearest dictionary element and use the label probabilities of these dictionary elements for your segmentation. We will do this, but first getting the index of the dictionary element for each pixel in the test image.

Suggested procedure

1. Compute a feature image from the test image.
2. Use a nearest neighbor algorithm (`knnsearch` in MATLAB or `Nearest Neighbors` from Scikit Learn in Python) and find the nearest cluster for each feature in the image.
3. Store the index of the nearest cluster center in an image of size $r \times c$.

3.2.5 *(E) Compute probability image and segmentation*

Based on the assignment image you should now compute the probability image and the final segmentation. The probability image will be of size $r \times c \times n_l$, where each pixel has a probability of belonging to one of the n_l labels. From the probability image you can compute the segmentation as the pixel-wise most probable label. The reason we go via a probability image, and not directly to a label image, is that we can regularize the segmentation by e.g. smoothing the probability image prior to choosing the most probable label.

Suggested procedure

1. Create an $r \times c \times n_l$ probability image.
2. In each pixel you insert the probability of the cluster center (index is stored in the assignment image from before).
3. Obtain a segmentation by selecting the most probable label in each pixel.
4. Try smoothing the probability image before selecting the most probable label.

3.2.6 *Segmentation with patch-based features*

Do the same as above using image patches instead of Gaussian features. This is a little more difficult because of boundary effects. You can extract image patches using the `im2col` function in MATLAB, and we have provided an `im2col` function found in the `feature_based_segmentation.py` file for Python, that has the same functionality.

4 Feature-based registration

Image registration is the process of transforming one image (the moving image) to the coordinate system of another image (the target image). We will compute this transformation by matching image features. Here we will use SIFT features (Scale invariant feature transform) that are described in detail in[1]. The reason for using SIFT is that OpenCV[2] has a good implementation of the SIFT features, which makes it easy to carry out the exercise on feature-based registration. But recent developments on learning-based methods for feature-matching will generally give higher accuracy and faster computation[3]. Therefore, learning-based methods will generally be a preferred choice. Learning-based methods roughly use the same components as traditional methods like SIFT, which are important for image registration. Traditional interest point features that do not involve learning such as SIFT are referred to as hand-crafted features.

[1] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2): 91–110, 2004

[2] OpenCV. Open source computer vision library, 2015

[3] Jiayi Ma, Xingyu Jiang, Aoxiang Fan, Junjun Jiang, and Junchi Yan. Image matching from handcrafted to deep features: A survey. *International Journal of Computer Vision*, 129(1):23–79, 2021

The concept of interest point image features is essential in image analysis. The basic idea is to identify salient positions in an image (unique key points) and use the surrounding image context to describe the image locally in the form of a descriptor vector. This is the same concept as we used for feature-based segmentation, where we wanted to group parts of the image with similar appearance or texture. But in image matching, we are interested in finding positions in two images of the same object or scene that can uniquely be matched.

This principle has been used for solving many problems including object recognition, image retrieval (image search), image stitching, geometric reconstruction, etc. Solutions to many of these problems have however been improved by machine learning methods and especially convolutional neural networks (CNNs) have replaced feature-based analysis in a range of applications. But interest point features are the best performing technique for finding correspondence between images with large structural variation, which is typical for photographs of the same scene or images from a microscope.

Image registration based on interest point features is based on detecting interest points in an image and finding unique correspondence

between these interest points. Interest point features are composed of two elements, namely an interest point or a key point, which is a position in the images (an (x, y)-coordinate) and a descriptor, which is a vector that encodes the appearance (pixel intensities) in a local neighborhood around the interest point. Typically, hundreds to thousands of features are detected in an image, and correspondence is found by finding the descriptors with the highest similarity between two images.

The desired properties for interest point features that they are

- unique such that they encode just one position in an image,
- invariant to change in view-point, overall intensity change, and change in noise level,
- robust such that they will be detected despite image changes, and
- efficient with respect to memory and computational time.

SIFT has been designed to have exactly these properties, and later developments are modifications that strive at optimizing these properties either through design or machine learning.

Learning-based features Some deep learning-based alternatives to hand-crafted features have been suggested[4], showing superior performance. High-performance examples include SuperPoint[5] that learns both interest point detection and description and Key.Net[6] that only learns interest point detection.

The problem of learning interest point features is that there is no ground truth for what a good interest point is. The SuperPoint method solves this by generating synthetic data, where specific image features such as corners are created and used for training an interest point detector. This detector is then used for detecting interest points in a set of real images. A set of corresponding images are created by using homographies to transform one image and detecting interest points in each of these. Since the homography gives ground truth correspondence, the sum of all detected interest points is used as ground truth. A second model is then trained for detecting these interest points, and the same network is used to simultaneously learn descriptors at the interest point locations. The resulting interest points have the same properties as hand-crafted features, namely a location and a vector that describes the local image appearance, which can then be matched by comparing interest points. The advantage is faster computation and higher accuracy in matching.

Besides learning models for computing interest point features, there are also methods for learning the matching between interest point features, e.g. the SuperGlue method[7]. Also here the advantage is higher performance compared to hand-crafted features.

[4] Jiayi Ma, Xingyu Jiang, Aoxiang Fan, Junjun Jiang, and Junchi Yan. Image matching from handcrafted to deep features: A survey. *International Journal of Computer Vision*, 129(1):23–79, 2021

[5] Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. Superpoint: Self-supervised interest point detection and description. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 224–236, 2018

[6] Axel Barroso-Laguna, Edgar Riba, Daniel Ponsa, and Krystian Mikolajczyk. Key. net: Keypoint detection by handcrafted and learned cnn filters. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5836–5844, 2019

[7] Paul-Edouard Sarlin, Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. Superglue: Learning feature matching with graph neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 4938–4947, 2020

Hand-crafted features – SIFT The advantage of hand-crafted features is that there is no learning involved, and therefore they will be easy to use for any type of images. Despite the lower performance compared to learning-based features, hand-crafted features will often perform well for many registration problems.

Scale invariant feature transform (SIFT) described in[8] is a widely used interest point feature. Here we focus on SIFT. But bear in mind that many other interest point features have similar properties[9].

We will use SIFT for feature-based image registration. The problem we address is that we are given two images that depict the same object and we want to find an image transformation that aligns the two images. This allows us to compare structures found in one image with structures found in the other image. In some cases, the difference between images can be modeled by a rotation $\mathbf{R}$, translation $\mathbf{t}$ and scale s. This is e.g. the case in microscopy or CT, where pixels have a fixed physical size. A more general affine transformation can be computed using a homography, but in cases where the imaging system can only transform the image by a rotation, translation, and scale, these will be the appropriate parameters to compute. Here we will work with sets of corresponding 2D points, where the correspondence is found by matching SIFT features.

[8] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2): 91–110, 2004

[9] Tinne Tuytelaars, Krystian Mikolajczyk, et al. Local invariant feature detectors: a survey. *Foundations and trends® in computer graphics and vision*, 3(3):177–280, 2008

Fitting two sets of 2D points in least squares sense. For two 2D point sets $\mathbf{P}$ and $\mathbf{Q}$ with corresponding elements $\mathbf{p}_i$ and $\mathbf{q}_i$ (2×1 column vectors), where $i = 1, \ldots, n$, we are interested in finding a rotation $\mathbf{R}$, a translation $\mathbf{t}$ and a scale s that minimizes the squared distance between the two point sets. The transformation from one point set to the other is given by

$$\mathbf{q}_i = s\mathbf{R}\mathbf{p}_i + \mathbf{t}\,. \tag{4.1}$$

The scale s can be found e.g. by computing the ratio between average distance to the centroid of each point set

$$s = \frac{\sum_{i=1}^{n} ||\mathbf{q}_i - \boldsymbol{\mu}_q||}{\sum_{i=1}^{n} ||\mathbf{p}_i - \boldsymbol{\mu}_p||}\,, \tag{4.2}$$

where $\boldsymbol{\mu}_p = \frac{1}{n}\sum_{i=1}^{n} \mathbf{p}_i$ and $\boldsymbol{\mu}_q = \frac{1}{n}\sum_{i=1}^{n} \mathbf{q}_i$ are the centroids of $\mathbf{p}_i$ and $\mathbf{q}_i$ respectively.

Now we want to find the rotation and translation that minimize

$$\sum_{i=1}^{n} (\mathbf{q}_i - s\mathbf{R}\mathbf{p}_i - \mathbf{t})^2\,.$$

One way of least-squares fitting 2D point sets involves 2-by-2 covariance matrix (normalization is not needed)

$$\mathbf{C} = \sum_{i=1}^{n} (\mathbf{q}_i - \boldsymbol{\mu}_q)(\mathbf{p}_i - \boldsymbol{\mu}_p)^T\,,$$

and its singular value decomposition

$$\mathbf{U}\Sigma\mathbf{V}^T = \mathbf{C}\,.$$

Here, $\mathbf{U}$ and $\mathbf{V}$ are the left and right singular matrices respectively and Σ is a diagonal 2-by-2 matrix containing the singular values. From this, we obtain the rotation

$$\hat{\mathbf{R}} = \mathbf{U}\mathbf{V}^T\,. \tag{4.3}$$

In rare cases, this computation can result in a reflection instead of a rotation. If the determinant of $\det(\hat{\mathbf{R}}) = 1$ it is a rotation and if $\det(\hat{\mathbf{R}}) = -1$ it is a reflection. This computation will typically only result in a reflection e.g. if there are only two points for determining the rotation. Therefore, we can compute the rotation taking this into account by

$$\mathbf{R} = \hat{\mathbf{R}}\mathbf{D}\,, \tag{4.4}$$

where

$$\mathbf{D} = \begin{bmatrix} 1 & 0 \\ 0 & \det(\hat{\mathbf{R}}) \end{bmatrix}\,. \tag{4.5}$$

Finally, we find the translation as the average vector from points in $\mathbf{q}$ to the rotated points in $\mathbf{p}$

$$\mathbf{t} = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{q}_i - s\mathbf{R}\mathbf{p}_i) = \boldsymbol{\mu}_q - s\mathbf{R}\boldsymbol{\mu}_p\,. \tag{4.6}$$

See e.g. Arun et al. [10] that covers a more general 3D case.

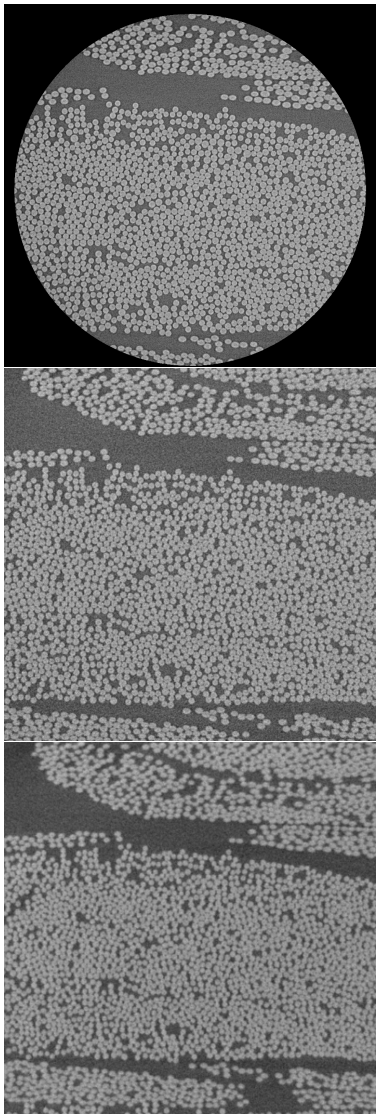

Figure 4.1: Example of an image of the same fiber sample acquired using a CT scanner at three resolutions.

[10] K Somani Arun, Thomas S Huang, and Steven D Blostein. Least-squares fitting of two 3-D point sets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (5):698–700, 1987

4.1 *Exercise on feature-based registration*

This exercise aims at finding correspondence between images by matching SIFT features and computing the transformation, i.e. the rotation, translation, and scale between the two images. For computing the SIFT features you can use `vlFeat` for MATLAB or `OpenCV` for Python (you will need a newer version of `OpenCV`). In the extra exercise, you can use the transformation obtained here to compare the fiber diameters that you got from using blob detection for fibers. But for now, the purpose is to compute the transformation.

4.1.1 *Rotation, translation and scale*

You should implement a function that takes two point sets as input and returns the rotation, translation, and scale. To ensure that you have the correct implementation, you can make a set of random 2D points and ensure that you get the correct numbers out. You can follow this procedure

1. Generate a random 2D point set $\mathbf{P}$.
2. Define variables for translation $\mathbf{t}$, rotation $\mathbf{R}$, and scale s, and decide on their values.
3. Transform $\mathbf{P}$ using these parameters to obtain the point set $\mathbf{Q}$.
4. Plot these point sets using two different colors.
5. Implement a function that computes the parameters $\mathbf{t}'$, rotation $\mathbf{R}'$, and scale s' from $\mathbf{P}$ and $\mathbf{Q}$. Make sure that you get the exact same values.
6. Add some noise to $\mathbf{Q}$ and recompute the parameters and test how much noise you can add and still get reasonably good parameter estimates.

4.1.2 *Compute and match SIFT*

Here you should compute SIFT features in two images and match them using Euclidean distance between their descriptor vectors. You should use the criterion by Lowe where a correct match is found, if the fraction between the closest feature vector and the second closest feature vector is less than a certain value, e.g. 0.6. There are functionality for matching features in both `vlFeat` and `OpenCV` that you can use. You can also implement your own matching function where you take e.g. scale and rotation into the matching criterion. You can follow this procedure

1. Create a transformed image by rotating, scaling, and cropping an image. We call the original `image1` and the transformed `image2`.
2. Compute SIFT features in the two images.
3. Match the SIFT features. You can use the functionality for matching SIFT from `vlFeat` for MATLAB and `OpenCV` for Python.
4. Display the match to see if the matching criterion is correct.
5. Extract the coordinates of the matching keypoints.
6. Use the function for computing the rotation, translation, and scale from before to transform the set of key points found in `image1` to the set of keypoints found in `image2`.
7. Display `image2` and plot the key points found in `image2` and the transformed key points from `image1`.
8. When you have confirmed that the points match, you can try to match the CT-images of fibers at the three resolutions shown in Figure 4.1. Visualize the matching feature points by drawing lines between them e.g. as shown in Figure 4.2. This allows you to visually evaluate the matching.

4.1.3 *Transform the matched features*

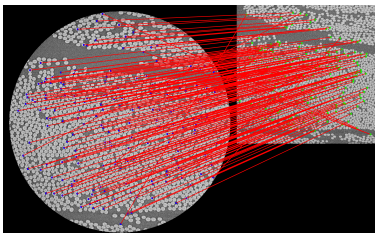

Figure 4.2: Matching SIFT features illustrated by red lines.

Now you should combine the function for computing the transformation parameters and the SIFT feature matching. The matching will not be perfect and you will most likely see some wrongly matched features. The larger the difference in appearance or scale of the image that is being matched, the more wrongly matched features can be expected. If the majority of the correspondences are correct then the least-squares fit will give a relatively good result.

Since we are computing the transformation by a least-squares fit, the outliers will affect the result to some extent. Outliers can however be removed relatively easily. You can compute the Euclidean distance between the two point sets after you have aligned them. There you will see that most of the distances are relatively small. And if you remove matching points with a distance larger than a certain threshold, you can repeat the computation of the transformation and obtain higher precision in the matching. You should implement a function that makes this two-step computation of the transformation and choose a good criterion for a threshold.

Illustrate your transformed feature points by plotting the two point sets on top of each other in the image e.g. as illustrated in Figure 4.3. You should be able to see a difference in the precision of the matching after removing the outliers.

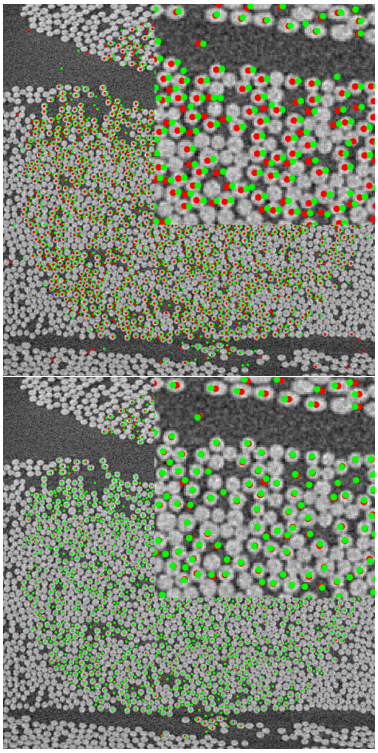

Figure 4.3: Matching features shown in red and green (zoom in upper right corner). Top is after computing least squares of all matching features and bottom is a recomputed match after removing outliers.

4.1.4 *Transform detected fibers (optional)*

You have now established a correspondence between the fiber images, and you can now compare the detected fibers from the exercise on scale-space blob detection. You can do this by

1. Compute transformation between two fiber images.
2. Detect blobs in the two images
3. Transform blob parameters (location and size) from one image to the other.
4. Match blobs and compare their individual sizes.

Part II

Image analysis with geometric priors

In image analysis, the terms *prior knowledge* and *contextual information* refer to all the information about the problem that is available *in addition* to the image data. This additional information may refer to some local appearance (how bright, dark, smooth, textured... something is) or to some geometric property (position, size, orientation, shape... of this something).

There are numerous ways of using prior information when solving image analysis problems, and, in this part of the lecture note, we look into three well-established approaches. First, we cover Markov random fields (MRF) where local contextual information is a part of a probabilistic framework that can be efficiently optimized using graph cuts. Then we present a segmentation model based on a parametric deformable curve, and for this, we introduce Mumford-Shah functional, Chan-Vese algorithm, and snakes. Lastly, we cover layered surface detection, a model useful when dealing with distinctive geometry.

The three approaches covered in this part of the note have many common points and may be combined in different ways. For example, layered surfaces use the same graph-cut solver as used in MRF. And curve-based segmentation can be driven by layered surfaces.

5 Markov random fields

Markov random fields (MRF) is a probabilistic model that can be used for various tasks in computer vision and image analysis. All MRF formulations involve labeling, i.e. assigning labels to some image entities, called *sites*, typically assigning labels to pixels. MRF are characterized by the Markov property, i.e. that the probability of a site (pixel) being assigned a certain label is only dependent on the neighborhood of the site.

In exercises on MRF, we will use MRF model for image segmentation. A segmentation can be formulated as assigning a (discrete) label to each pixel in the image. Often, we would like the segmentation to be smooth, which is the local contextual information we wan to incorporate in our model. In MRF we do this by giving a low probability for a configuration where many neighboring pixels have different labels. To do so we add a term, *a prior*, next to the usual *data-term*, which in this context is also called *a likelihood term*. Provided an image, we aim at finding a label configuration that maximizes the *a posterior* (MAP) probability which is a combination of a likelihood (data) term and the term modelling a smoothness prior.

One characteristics of MRF, *Markov-Gibbs equivalence* following directly from Markov property, is that the probability of the MRF configuration is an exponential of the negative configuration energy. It is more practical to work with energies, as energy contributions add up, while probabilities multiply. Therefore, instead of maximizing the posterior probability we will minimize the posterior energy of the configuration f given by

$$E(f) = U(f|d) = U(d|f) + U(f)\,, \tag{5.1}$$

where $E(f)$ is a (segmentation) energy of the configuration f for a certain data (image) d. Here, $U(f|d)$ is a posterior energy and $U(d|f)$ is a likelihood energy.

Another important property of MRF is that both likelihood and prior may be expressed as sums of local contributions called *potentials*. In the exercise, we will work with so-called *one-clique potentials* (single pixels) and *two-clique potentials* (pixel pairs). In terms of clique potentials (5.1)

becomes

$$E(f) = \sum_{\{i\} \in \mathcal{C}_1} V_1(f_i) + \sum_{\{i,j\} \in \mathcal{C}_2} V_2(f_i, f_j)$$

where $\mathcal{C}_1$ is the set of one-cliques (for example single pixels), and V_1 is a one-clique potential used for modeling the likelihood term, $\mathcal{C}_2$ is the set of two-cliques (for example pixel pairs), and V_2 is a two-clique potential used for modeling the prior term.

As such MRF framework allows us to compute a (global) probability of a whole configuration when some (local) property is modeled as a sum of clique potentials.

As discussed in the book by Li[1], Chapter 1, Introduction, first paragraph, the main concerns of the MRF framework are *how to define clique potentials* (modelling part), and *how to find the optimal solution* for a given energy function (optimization part).

[1] Stan Z Li. *Markov random field modeling in image analysis*. Springer Science & Business Media, 2009

5.1 *Gender determination, an easy introduction to MRF*

We start with the extremely small 1D example with the aim of introducing MRF terminology, demonstrating the modelling possibilities provided by MRF, and the use of the terms *likelihood* or *data term*, *prior* and *posterior*.

Imagine entering a bar and observing 6 persons standing along the counter. You estimate persons heights (in cm) and record this data as

$$d = \begin{bmatrix} 179 & 174 & 182 & 162 & 175 & 165 \end{bmatrix}.$$

You want to estimate the persons gender, i.e. you want to assign either a label M or F to each person. You may consider each person individually according to only its own height, but you also want to utilize your knowledge of the contextual information. You decide to pose the problem as a MRF with the neighbourhood given by the first neighbor (person to the left and person to the right).

But first, let's consider the likelihood (data) term. You know that the average male height is 181 cm, the average female height is 165 cm, and that the height for each gender may be described as following a normal distribution where you assume the same standard deviation for both genders. For this reason you define the likelihood terms as one clique potentials

$$V_1(f_i) = (\mu(f_i) - d_i)^2$$

where d_i is the height of the person i, f_i is a label assigned to the person i (i.e. either M or F), and $\mu(f_i)$ is either $\mu_M = 181$ or $\mu_F = 165$. The likelihood energy of a configuration $f = [f_1 \dots f_6]$ is the sum of all one-clique potentials

$$U(d|f) = \sum_{i=1}^{6} V_1(f_i)\,.$$

For example, configuration $\begin{bmatrix} M & M & M & M & M & F \end{bmatrix}$ has likelihood energy of 451.

To find the configuration which minimizes the likelihood energy you can consider the one-clique potentials for all i and both labels

$$\begin{array}{lcccccc} (\mu_M - d_i)^2 & : & 4 & 49 & 1 & 361 & 36 & 256 \\ (\mu_F - d_i)^2 & : & 196 & 81 & 289 & 9 & 100 & 0 \end{array}$$

Obviously, the minimal likelihood energy is obtained if we choose a label which minimizes the cost for each i, resulting in a labeling

$$f^D = \begin{bmatrix} M & M & M & F & M & F \end{bmatrix}, \tag{5.2}$$

and giving $U(d|f^D) = 99$. Another thing to notice is that additional cost for deviating from this labeling varies, depending on which label we change. For example, it costs additional 352 to label the forth person as male, while it only costs additional 32 to label the second person as female.

Now we want to incorporate the contextual (prior) information about the gender of the people standing along the bar counter. To make this example similar to modeling smoothness, let's say that you expect a configuration with men standing next to women to occur less frequently then configurations where genders group. For this reason, you may decide to incorporate a cost which penalizes a less-frequent configuration. For prior energy you therefore define 2-clique potentials as

$$V_2(f_i, f_{i'}) = \begin{cases} 0 & \text{if } f_i = f_{i'} \\ 100 & \text{otherwise} \end{cases}.$$

The prior energy is the sum of all 2-clique potentials for all 2-cliques (all pairs of neighbors) in a configuration. Utilizing the fact that two-clique contains five neighbour pairs $(i, i+1)$ for $i = 1, ..., 5$, we write

$$U(f) = \sum_{i=1}^{5} V_2(f_i, f_{i+1}).$$

Obviously, this prior energy is zero (i.e. minimal) for a configuration with all labels being equal, while a configuration alternating between a male and a female yields a maximal prior energy of 500. The prior energy for the configuration f^D which we earlier showed to minimize the likelihood energy is $U(f^D) = 300$, somewhere in between the smallest and the largest prior.

According to (5.1), the posterior energy of configuration f^D is

$$U(f^D|d) = U(d|f^D) + U(f^D) = 99 + 300 = 399.$$

The question is, can we find a configuration which yields a better (smaller) posterior energy? And finally, which configuration minimizes posterior energy?

For our small problem, we can simply try all different configurations (there are $2^6 = 64$ in total). Relatively easy we can confirm that a configuration

$$f^O = \begin{bmatrix} M & M & M & F & F & F \end{bmatrix}$$

with $U(f^O|d) = 163 + 100 = 263$ is an optimal configuration.

Note how smoothness cost of 100 (later we call this parameter β) influences which configuration is optimal. In general, the choice of the smoothness parameter depends on your confidence in the prior, compared to the data. Note also that by incorporating the prior information we made sure to find what we expected to find in the first place.

Finally, note the distinction between modeling (setting up the problem by defining a likelihood term and a prior term) and optimization (finding the configuration which minimizes the posterior energy) which in this case involved trying all configurations.

5.2 *MRF modelling for image segmentation*

In this exercise we define an energy function for segmenting a noisy image, similar to the problem in Li Section 3.2.2. Here, we will compute the energy of different configurations to confirm that minimizing the segmentation energy leads towards the desired solution. The model we use is very similar to the model used for gender determination. In this exercise we use synthetic data (i.e. we produce the input image by adding noise to a ground truth image) shown if Figure 5.1. This allows us to evaluate the quality of our energy function. In the text the input image is denoted D (data) and ground truth segmentation S_{GT} where elements of S_{GT} are from the set $\{1, 2, 3\}$ corresponding to the darkest, medium gray, and brightest class.

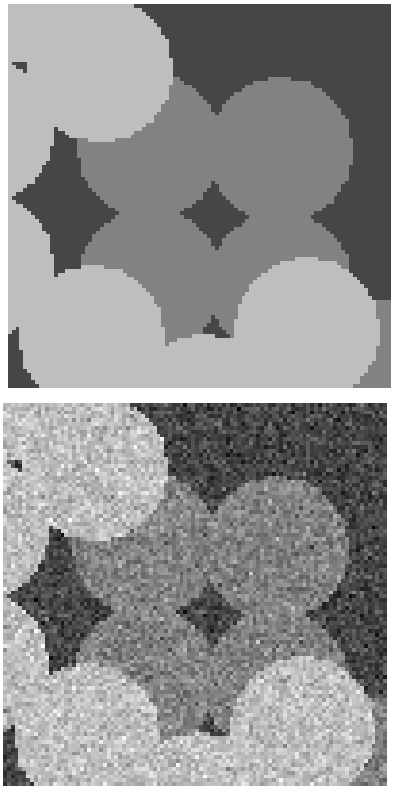

Figure 5.1: A ground truth (the desired segmentation should resemble ground truth) and a noisy image (input data).

Looking at the histogram of the pixel intensities and the intensities divided into the ground-truth classes, Figure 5.2, we observe overlapping distributions, so we can not expect a good segmentation if considering only individual pixel intensities.

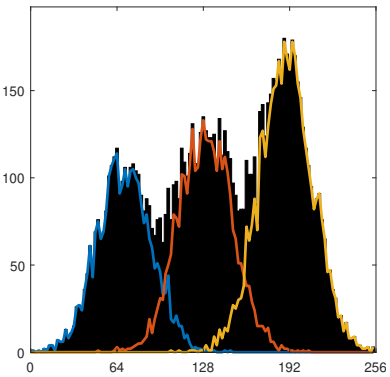

Figure 5.2: Intensity histogram of the noisy image and the histograms for the three segments.

Now we pose image segmentation as a MRF. Sites are pixels, labels are from $\{1, 2, 3\}$, and we choose a first-order neighborhood (four closest pixels). As in the previous example, we define the one-clique potentials for the likelihood energy as the squared distance from the class mean

$$V_1(f_i) = (\mu(f_i) - d_i)^2$$

where d_i are intensities of the (noisy) image, f_i are pixel labelings given by the configuration, and values μ are estimated from the histogram and set to $\mu_1 = 70$, $\mu_2 = 130$, $\mu_3 = 190$. As before, the likelihood energy is

$$U(d|f) = \sum_i V_1(f_i)\,,$$

where summation now covers all image pixels. Similarly to the previous example, we define 2-clique potentials for discrete labels which penalizes neighbouring labels being different

$$V_2(f_i, f_{i'}) = \begin{cases} 0 & f_i = f_{i'} \\ \beta & \text{otherwise} \end{cases},$$

and prior energy

$$U(f) = \sum_{i \sim i'} V_2(f_i, f_{i'})$$

where summation runs over all pairs of neighbouring pixels and β is a smoothness weight, which for `uint8` pixel intensities you may set to 100. The posterior energy is now given by (5.1).

We want to check that our optimal function leads to the desired result. That is, we want to make sure that posterior energy gets smaller when we approach the desired result. Therefore we want to compute the likelihood, prior and posterior for some reasonable segmentations (MRF configurations). For the purpose of this testing, we produce at least two segmentations of the noisy image D. This can be a segmentation obtained by thresholding D at intensity levels 100 and 160 (valleys of the histogram). The second segmentation may be computed by median filtering S_T using an appropriate kernel. You are welcome to produce additional configurations, e.g. by applying a Gaussian filter to D prior to thresholding, or by using morphological operations. For all candidate configurations you should take a look at the intensity histograms of three classes, similarly as for the S_{GT} earlier.

To observe how likelihood (V_1), prior (V_2) and posterior ($V_1 + V_2$) change for different configuration, you need the functions which compute these energies for a given image D, a configuration S and the MRF parameters μ (intensities for segmentation classes) and β (smoothness term). You may chose to write one function which returns V_1 and V_2, or two separate functions.

Tasks

1. Get hold of the gray-scale image D and one configuration for the segmentation S. To begin with, this may be a ground truth segmentation. Notice that a ground truth segmentation (labeling) is not the same as a noise-free image: elements of the segmentation are labels $1, 2, \ldots$ while elements of the noise-free image are pixel intensities $\mu_1, \mu_2, \ldots$.

2. For computing V_1 you need D, S and μ. First, compute an intensity-realization of S, that is an image where each occurrence of label f_i is replaced by $\mu(f_i)$. It is then easy to compute V_1 as a sum of squared differences.

3. For computing V_2 you need S and β. Recall that an almost identical problem was solved in week 1.

4. Produce some other configurations S, by any means you find appropriate: thresholding, manually drawing, modifying ground truth... Apply your two functions to all configurations, and display the likelihood, prior and posterior for every configuration.

5. If we consider only the likelihood, which configuration is the most probable? If we consider only the prior energy, which configuration is the most probable? What if we consider the posterior energy?

6. Would you expect that minimizing the posterior energy leads to a good segmentation? If not, try adjusting β.

5.3 *Graph cuts for optimizing MRF*

The interactions modelled by MRF prior make optimization (finding an optimal configuration) of the MRF very difficult. General MRF optimization methods may be very slow, but efficient graph cut algorithms can be used for a subset of problems.

A binary (two label) MRF problem with submodular second order energy (loosely speaking an energy favoring smoothness and having only one-clique and two-clique potentials) can be exactly solved by finding a minimum s-t cut of a graph constructed from the energy function [2,3,4]. A minimum s-t graph cut can be found e.g. using the Ford and Fulkerson algorithm, or an efficient freely available graph cut implementation by Boykov and Kolmogorov. A multiple-label discrete MRF problem can also utilize graph cuts via iteratively solving multiple two-label graph cuts, e.g. by using α expansion.

In the following exercises we are using graph cuts to optimize discrete MRF. Students using python may use `PyMaxflow` package documented at `http://pmneila.github.io/PyMaxflow/index.html`. MATLAB users may use the provided code, in particular the `GraphCutMex` function. This is a slightly modified version of the older Boykov implementation, the newest version can be found at `http://pub.ist.ac.at/~vnk/software.html`. Furthermore, MATLAB has a built-in functionmaxflow which also implements Boykov's algorithm and may be used instead.

To begin with, we look back at the small example with gender labeling. Recall that the heights (in cm) of 6 persons are

$$d = \begin{bmatrix} 179 & 174 & 182 & 162 & 175 & 165 \end{bmatrix}$$

and we want to estimate the persons gender. For likelihood we use squared distance from the means $\mu_M = 181$, $\mu_F = 165$. For the prior we use $\beta = 100$ as a penalty for neighbouring labels being different.

[2] Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(11):1222–1239, 2001

[3] V Kolmogorov and R Zabih. What energy functions can be minimized via graph cuts? *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(2): 147–159, 2004

[4] Y Boykov and V Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(9):1124–1137, 2004

We want to construct a *s-t* graph corresponding to this problem. The construction is not unique. When choosing an approach, the focus is often on constructing a graph with fewest edges, as suggested in Li book Section 10.4.2. However, you might prefer constructing a more intuitive graph despite having a higher number of edges. This approach is sketched in Figure 5.3. Terminal edges (linking to source and sink) are used for the likelihood energy terms, while internal edges model the prior energy terms. Confirm that a cost of an *s-t* cut in this graph equals to the posterior energy of the corresponding configuration.

To be able to compute the optimal configuration using the MATLAB `GraphCut` function, we need to create two matrices which contain edge weights to be passed to the function. The matrix containing terminal weights and the matrix containing weights between internal nodes for gender assignment example are shown in Figure 5.4. Wrapper in python has a slightly different manner of passing graph weights to the function, as explained in the package documentation (look at *A first example*).

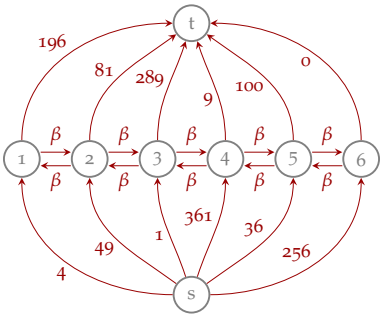

Figure 5.3: A sketch of a *s-t* graph for a gender labeling problem.

$$E_{\text{terminal}} = \begin{bmatrix} 1 & 4 & 196 \\ 2 & 49 & 81 \\ 3 & 1 & 289 \\ 4 & 361 & 9 \\ 5 & 36 & 100 \\ 6 & 256 & 0 \end{bmatrix}$$

$$E_{\text{internal}} = \begin{bmatrix} 1 & 2 & \beta & \beta \\ 2 & 3 & \beta & \beta \\ 3 & 4 & \beta & \beta \\ 4 & 5 & \beta & \beta \\ 5 & 6 & \beta & \beta \end{bmatrix}$$

Figure 5.4: Representing a *s-t* graph using two matrices, one containing weights of terminal edges and one matrix for internal edges.

Tasks

1. Install the required `maxflow` package (python) for graph cuts and/or download the provided software (MATLAB).

2. Get a small scripts which solves the gender labeling problem.

3. Run the script and get familiar with the functionality of your graph-cut solver.

4. Which configuration is optimal? Change $\beta = 10$ and solve again. Which configuration is optimal now? Try also $\beta = 1000$.

5.4 *Binary segmentation using MRF*

Now we can set up and solve a binary segmentation using MRF. When using a method for the first time, it is always a good idea to start with an easy example, where it is clear what the desired output is. This may be a simple image corrupted by noise, for example a noisy image of the DTU logo.

After that, we will segment the bone image `V12_10X_x502.png` shown in Figure 5.5. The image is a slice from a CT scan of a mouse tibia. You can visually distinguish air (very dark), bone (very bright) and cartilage (dark). The task here is to segment the image in two segments: air and bone. Cartilage should be segmented together with air. In the next exercise we look at multilabel segmentation, allowing us to distinguish all three materials as in Figure 5.6.

The model we use is still the same as in the previous exercises, with the likelihood as the sum of squared distances, and the prior penalizing neighboring labels being different.

Tasks

1. The DTU logo is of type `uint8` and should be converted into double precision (float) before any computation. You also want to divide image intensities with 255, as this will simplify the weighting between the likelihood and the prior term.
2. For the DTU logo use $\mu_1 = 90/255$ and $\mu_2 = 170/255$.
3. To pass the weights of terminal and internal edges to graph solver, python users may use the functionality provided by `maxflow`, see the example *Binary image restoration* but notice that it uses a different energy formulation. MATLAB users need to construct matrices with indexes and weights, and here it may help to start by creating an index matrix `X = reshape(1:r*c, [r, c])`. You can than use appropriate parts of `X` to create index part of matrices passed to `GraphCutMex`.
4. For the DTU logo, choose parameter β which yields in a good segmentation.
5. The bone image is of type `uint16` and after converting it into double precision you may also want to divide image intensities with $2^{16} - 1$.
6. For bone image, you need to determine the mean intensities of the air and bone (μ_1 and μ_2) by inspecting the histogram. Look at maximum likelihood configuration, to confirm that your μ_1 and μ_2 are suitable. Choose a (small) parameter β and compute the optimal configuration for this β using your graph cut solver. Then, adjust β to obtain a visually pleasing segmentation with reduced noise in air and bone choose. Observe how changing β affects the segmentation.
7. For your results, you may want to produce a figure showing histogram of the entire image, and on top of that the intensity histograms of the air and bone classes, similar to how it was done in the modelling exercise.

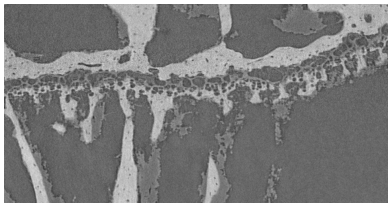

Figure 5.5: A bone image.

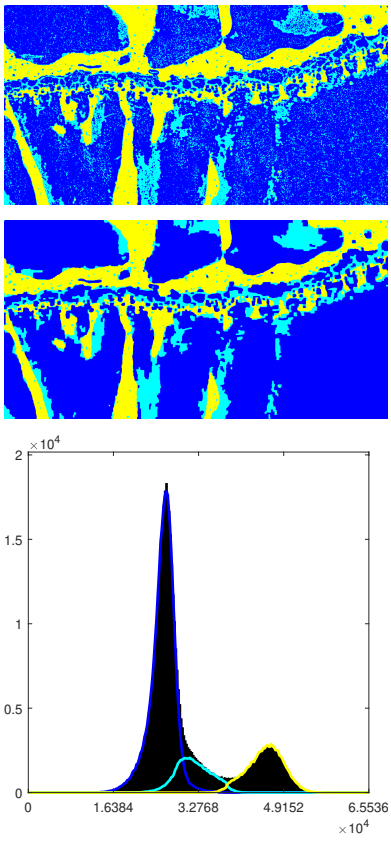

Figure 5.6: A segmentation of bone using maximum likelihood (top) and maximum posterior (middle). Histograms show intensity distributions for the three segmented classes.

5.5 *Multilabel segmentation usign MRF (optional)*

Multilabel segmentation is obtained using an iterative α expansion algorithm. In python, use the `maxflow.fastmin.aexpansion_grid`function which is a part of `maxflow.fastmin`. For those using MATLAB we provide a function `multilabel_MRF` which implements α expansion. Read

the help text of the function for explanation on input and output variables.

You should first verify the quality of the solution provided by the α expansion algorithm by segmenting the synthetic image, for example circles. How does the energy of the graph cut solution compare to the energies of the configurations found in the modeling exercise? Try changing β to see how it affects the result.

Use the α expansion algorithm to segment the bone image into air, cartilage and bone class. The challenge here is to distinguish between air and cartilage. You should aim at producing a visually pleasing result with cartilage as solid as possible (without noisy pixels segmented as air) and air as clean as possible (without noisy pixels segmented as cartilage). A good result can be obtained by tweaking two parameters: the mean value for the cartilage class and the smoothness weight β. The mean intensity for air and bone class can be estimated from the histogram.When adjusting a mean value for cartilage, choose first no smoothing ($\beta = 0$) and try to obtain a reasonable (but noisy) segmentation. Then increase β to remove the noise.

After you have tuned the parameters and obtained a nice segmentation, try segmenting the other bone image (`V8_10X_x502.png`). Can you use the same parameters? Why?

6 Deformable models

Often, image segmentation involves a combination of two terms: one dealing with the image data and the other describing a desirable segmentation. For example, we have used Markov random fields to impose smoothness on the segmentation. The topic of this chapter, deformable models for image segmentation, is another strategy which combines two contributions: the first originating from the image, and the second imposing smoothness.

With deformable models, image segmentation is performed by evolving (moving, displacing) a curve in an image. The curve moves under the influence of *external forces*, which are computed from the image data, and *internal forces* which have to do with the curve itself.

Deformable models are generally classified as either *parametric* (also called explicit) or *implicit* (in the context of image segmentation also called geometric), depending on the method used for representing the curve, see Figure 6.1. Despite this fundamental difference in curve representation, the underlying principles of both methods are the same [1].

In the exercise for this chapter, we use parametric curve representation, often called a *snake* [2], $C(s) = (x(s), y(s))$ where parameter $s \in [0,1]$ is an arclength. In a discrete setting, this becomes a sequence of points (x_s, y_s), and parameter s becomes a discrete index $s = \{1, \ldots, n\}$ indicating ordering of the points. It is convinient to represent such a curve using an $n \times 2$ array (matrix) of numbers (coordinates).

In the exercise we consider an image where the task is to separate the foreground from the background. At any time, a curve C divides the image into inside and outside region. We want to deform the curve such that it, eventually, delineates the foreground. In the notation, we will use subscripts in and out for inner and the outer region of the curve.

The curve is guided by the segmentation energy E, which should be defined such that the desired segmentation has a minimal energy. The desired segmentation should depend on the image data, but should also be relatively smooth. To incorporate those two factors, there are

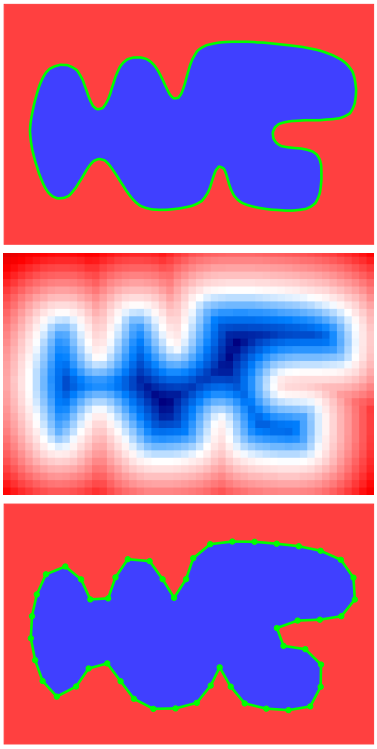

Figure 6.1: A curve (top) and its two discrete representations: implicit (middle) and parametric (bottom).

[1] Chenyang Xu, Anthony Yezzi Jr, and Jerry L Prince. On the relationship between parametric and geometric active contours. In *The Asilomar Conference on Signals, Systems,and Computers*, volume 1, pages 483–489. IEEE, 2000b

[2] Michael Kass, Andrew Witkin, and Demetri Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, 1(4):321–331, 1988

two energy contributions $E = E_{\text{ext}} + E_{\text{int}}$. Here, we use E_{ext} to denote external energy, which is a contribution to the segmentation energy determined by image data. And we use E_{int} for internal energy, which has to do with the curve itself.

A segmentation is obtained by iteratively moving the curve to minimize the energy, and the most challenging part of the approach is deriving energy-minimizing curve deformation forces $F = -\nabla E$. Since energy may be broken in two contributions, the forces also have two terms, $F = F_{\text{ext}} + F_{\text{int}}$.

To allow deformation, the curve is made dynamic (time-dependable), and its change in time, often denoted *evolution*, is given by

$$\frac{\partial C}{\partial t} = F(C)\,.$$

This exercise is inspired by the Chan-Vese algorithm [3], a deformable model for image segmentation which minimizes a piecewise-constant Mumford-Shah functional. In the original formulation, Chan-Vese uses a implicit (level-set) curve representation and a two-step optimization. We will use the solution of the Chan-Vese approach, but, instead of using level-sets, we will combine it with a parametric curve representation. For this reason, even though our model has an external energy identical to the one used by the Chan-Vese algorithm, the internal energy we use is the same as what is used for snakes.

[3] Tony F Chan and Luminita A Vese. Active contours without edges. *IEEE Transactions on image processing*, 10(2):266–277, 2001

In the following two sections, we briefly cover first the external, and then the internal energy. The detailed explanation on how external forces are derived from external energy can be found in the Chan-Vese article. How internal forces are derived is explained in Chapter 3 of the Handbook of Medical Imaging, Image Segmentation Using Deformable Models [4], subsection 3.2.1 and 3.2.4. Recall that you already implemented curve smoothing as one of the introductory exercises during the first week of the course.

[4] Chenyang Xu, Dzung L Pham, and Jerry L Prince. Image segmentation using deformable models. *Handbook of medical imaging*, 2:129–174, 2000a

6.1 *External energy, Chan-Vese*

An external energy closely related to the two-phase piecewise constant Mumford-Shah model is

$$E_{\text{ext}} = \int_{\Omega_{\text{in}}} (I - m_{\text{in}})^2 \, d\omega + \int_{\Omega_{\text{out}}} (I - m_{\text{out}})^2 \, d\omega$$

where I is an image intensity as a function of the pixel position, while m_{in} and m_{out} are mean intensities of the inside and the outside region. This energy seeks the best (in a squared-error sense) piecewise constant approximation of I. An evolution that will deform a curve toward an energy minimum is derived as

$$F_{\text{ext}} = (m_{\text{in}} - m_{\text{out}})\,(2I - m_{\text{in}} - m_{\text{out}})\,N\,. \tag{6.1}$$

where N denotes an outward unit normal of the curve.

In other words, the curve deforms in the normal direction, and for every point on the curve we only need to compute the (signed) length of the displacement. We will denote the scalar components of the force as $f_{\text{ext}} = (m_{\text{in}} - m_{\text{out}})\,(2I - m_{\text{in}} - m_{\text{out}})$. Note that this can be written as $f_{\text{ext}} = 2(m_{\text{in}} - m_{\text{out}})\left(I - \frac{1}{2}(m_{\text{in}} + m_{\text{out}})\right)$, i.e. the signed length of displacement is proportional to the difference between the image intensities and the mean of m_{in} and m_{out}.

6.2 *Internal forces, snakes*

The internal energy is determined solely by the shape of the curve. In the classical snakes formulation internal forces discourage stretching and bending of the curve

$$E_{\text{int}} = \frac{1}{2}\int_0^1 \alpha \left|\frac{\partial C}{\partial s}\right|^2 + \beta \left|\frac{\partial^2 C}{\partial s^2}\right|^2 ds\,,$$

with weights α and β controlling the elasticity (first-order derivative) and the rigidity (second-order derivative) term. Corresponding deformation forces are

$$F_{\text{int}} = \frac{\partial}{\partial s}\left(\alpha \frac{\partial C}{\partial s}\right) - \frac{\partial^2}{\partial s^2}\left(\beta \frac{\partial^2 C}{\partial s^2}\right)\,. \tag{6.2}$$

Those internal (regulatory) forces are the key to success of deformable models, as they provide robustness to noise.

Since our snake is discrete, the derivatives should be approximated by finite differences. Applying internal forces (regularization) now corresponds to filtering (smoothing) the curve with filters for the second and the (negative) fourth derivative, i.e. the filter $\begin{bmatrix} 1 & -2 & 1 \end{bmatrix}$ and the filter $\begin{bmatrix} -1 & 4 & -6 & 4 & -1 \end{bmatrix}$. Those contributions, weighted by parameters α and β are now used to regularize (smooth) the curve. In efficient implementation this is done by a matrix multiplication, and for better stability we use a backward Euler scheme. For slightly more detail, you can revise the introductory exercise on curve smoothing 1.1.3.

6.3 *Final model*

For a snake consisting of n points and represented using an $n \times 2$ matrix $\mathbf{C}$, a final discrete update step is, adapted from Handbook of Medical Imaging, Eq. (3.22),

$$\mathbf{C}^t = \mathbf{B}_{\text{int}}\left(\mathbf{C}^{t-1} + \tau\,\text{diag}(\mathbf{f}_{\text{ext}})\,\mathbf{N}^{t-1}\right)\,. \tag{6.3}$$

In this expression τ is the time step for displacement, while $\mathbf{B}_{\text{int}}$ is the $n \times n$ matrix used for regularizing the curve and taking the role of the internal forces. Curve normals are represented as an $n \times 2$ matrix $\mathbf{N}$, and pointwise displacement is obtained by multiplying $\mathbf{N}$ with a $n \times n$ diagonal matrix containing the displacement lengths. Note that the multiplication with matrix $\text{diag}(\mathbf{f}_{\text{ext}})$ is simpler to implement as a row-wise multiplication with a vector $\mathbf{f}_{\text{ext}}$.

6.4 *Exercise: Segmentation and tracking*

We will use a deformable model to segment and track a simple organism in a sequence of images. You are provided with two image sequences: crawling amoeba [5] and water bear [6]. The same code can be used for both sequences, with only a minor adjustment in a pre-processing step.

While tracking is the goal of the exercise, it is always advisable to use very simple problems while writing the first version of the code. You are therefore advised to first implement curve deformation (steps 3 to 9 in the approach sketched below) for a simple image, for example the provided `plusplus.png`. Once you can segment plus-plus, move on to segmenting and tracking image sequences.

Steps for solving the whole tracking problem are listed below, with the hints for MATLAB and python users. For step 8, we provide a couple of helping functions.

[5] The video of crawling amoeba is from Essential Cell Biology, 3rd Edition Alberts, Bray, Hopkin, Johnson, Lewis, Raff, Roberts, & Walter, https://www.dnatube.com/video/4163/Crawling-Amoeba

[6] The video of water bear is from Olympus microscopy resources, https://www.olympus-lifescience.com/ru/microscope-resource/moviegallery/pondscum/tardigrada/echiniscus

Tasks

1. Read in and inspect the movie data. In MATLAB you may use `VideoReader`. You may save the image sequence as a multi-dimensional array, or as a movie object using `im2frame` conversion. In python you may use function `get_reader` from `imageio` package.

2. Process movie frames. For our segmentation method to work, movie frames need to be transformed in grayscale images with a significant difference in intensities of the foreground and a background. For the movie showing the crawling amoeba (which is white on a dark background), it is enough to convert movie frames to grayscale. Transforming intensities to doubles between 0 and 1 is advisable, as it might prevent issues in subsequent processing. For the movie of the echinicsus, we want to utilize the fact that foreground is yellow while background is blue. A example of suitable transformation is $(2b - (r + g) + 2)/4$, with r, g, b being color channels (with values between 0 and 1).

3. Choose a starting frame and initialize a snake so that it roughly delineates the foreground object. You may define a circular snake

with points $(x_0 + r\cos\alpha, y_0 + r\sin\alpha)$, where (x_0, y_0) is a circle center, r is a radius and angular parameter α takes n values from $[0, 2\pi\rangle$. See Figure 6.2 for example, but use approximately 100 points along the curve.

4. Compute mean intensities inside and outside the snake. In MATLAB you can use `poly2mask` function. In python use `polygon2mask` from package `skimage.draw` introduced in version 0.16.

5. Compute the magnitude of the snake displacement given by Eq. (6.1). That is, for each snake point, compute the scalar value giving the (signed) length of the deformation in the normal direction. This depends on image data under the snake and estimated mean intensities, as shown in Figure 6.3. A simple approach evaluates the image intensities under the snake by rounding the coordinates of the snake points. A more advanced approach involves interpolating the image at the positions of snake points for example using bilinear interpolation, which is in MATLAB implemented in function `interp2`, and is in python available under the same name in `scipy.interpolate` package.

6. Write a function which takes snake points $\mathbf{C}$ as an input and returns snake normals $\mathbf{N}$. A normal to point c_i can be approximated by a unit vector orthogonal to $c_{i+1} - c_{i-1}$. (Alternatively, and slightly better, you may average the normals of two line segments meeting at c_i.) Displace the snake. Estimate a reasonable value for the size of the update step by visualizing the displacement. You should later fine-tune this value so that the segmentation runs sufficiently fast, but without introducing exaggerated oscillations. This step corresponds to computing the expression in the parentheses in the Eq. (6.3).

7. Write a function which given α, β and n constructs a regularization matrix $\mathbf{B}_{\text{int}}$. Your code from the introductory exercise could be used. Apply regularization to a snake. Estimate a reasonable values for the regularization parameters α and β by visualizing the effect of regularization. You should later fine-tune these values to obtain a segmentation with the boundary which is both smooth and sufficiently detailed. This step corresponds to matrix multiplication on the right hand side of the Eq. (6.3).

8. The quality of the curve representation may deteriorate during evolution, especially if you use a large time step τ and/or weak regularization, i.e. small α and β. To allow faster evolution without curve deterioration, you may choose to apply a number of substeps (implemented as subfunctions) which ensure the quality of the snake:

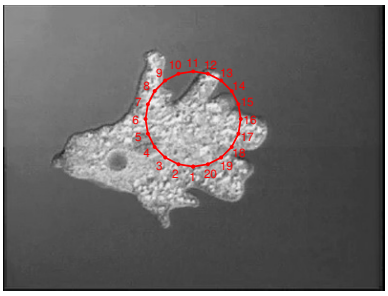

Figure 6.2: The first frame of a crawling amoeba and a circular a 20-point snake.

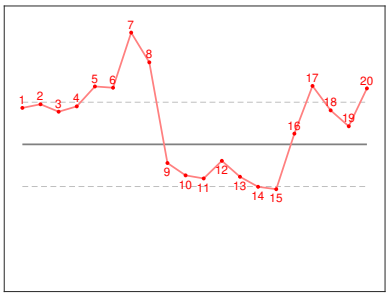

Figure 6.3: Red curve shows image intensities along the snake in Figure 6.2. Dashed gray lines indicate m_{in} and m_{out}, while gray line indicates the mean of m_{in} and m_{out}. Signed length of the curve displacement f_{ext} is computed from the difference between the red curve and the gray line.

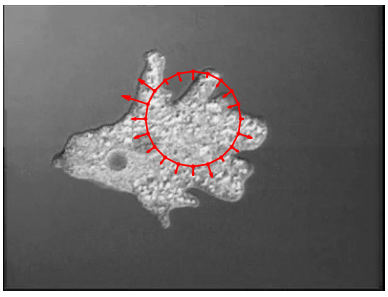

Figure 6.4: External force on the curve indicated by arrows. Displacement is in the normal direction and the length of the displacement is given by the values shown in Figure 6.3.

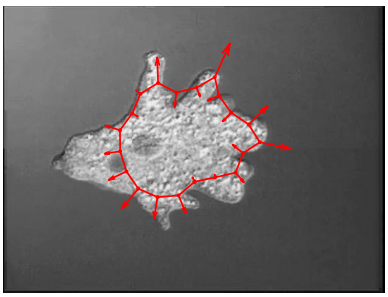

Figure 6.5: The curve and the external forces after 20 iterations.

- Constrain snake to image domain.
- Distribute points equidistantly along the snake. This can be obtained using 1D interpolation (function `interp1`).
- Apply heuristics for removing crossings from the snake. For example, if you detect self-intersection, identify two curve segments separated by the intersection and reverse the ordering of the smallest segment.

We provide functions for distributing points and removing crossings, in MATLAB and in python.

9. Repeat steps 4–8 until a desirable segmentation is achieved. Note that the regularization matrix only depends on regularization parameters and a number of snake points. This is constant when the size of the snake and the regularization are fixed, which is a typical case. It is therefore sufficient to precompute $\mathbf{B}_{\text{int}}$ prior to looping. Figure 6.5 shows our 20-point snake during evolution.

10. Read in the next frame of the movie, and use the results of the previous frame as an initialization. Evolve the curve a few times by repeating steps 4–8.

11. Process additional frames of the image sequence.

7 Layered surfaces

A SEGMENTATION PROBLEM can sometimes be geometrically constrained. We may for example be interested in segmenting a roughly horizontal layer or a roughly circular object. In 3D we may be interested in terrain-like surface, a tubular object, or a spherical object. Such topological constraints strongly reduce the solution space for the segmentation, and may turn an otherwise challenging problem into an easily solvable problem.

An example of segmentation problem which may benefit from constraining the solution, such that it consists of layers, is shown in Figure 7.1.

In this exercise, we focus on optimal net surface detection via graph search originally suggested by Wu and Chen[1] and popularized by Li, Wu, Chen and Sonka[2]. To segment terrain-like surfaces, they construct a graph on a set of sample points from a volume, such that the roughness of possible solutions is constrained. The optimality of the solution is defined in terms of a volumetric cost function derived from the data. The algorithm can find multiple interrelated layered terrain-like and tubular surfaces, which made it applicable for medical image segmentation and led to numerous extensions. One important extension involves a cost function which determines an optimal placement of the surface: a cost function, originally defined only in terms of on-surface appearance, has been extended to incorporate appearance of the regions between surfaces [3].

We will here review an algorithm for finding optimal layered surfaces in 3D, with focus on the inputs and the outputs. For details on *how* this algorithm works, the reader is referred to article by Li et al. Note that while the theory given in 7.1 covers the 3D case (surfaces in the volume), the exercise in 7.4 is on 2D case (curves in the image).

We also very briefly cover the principle of transforming the data into a volumetric cost, which is the input to the layered surface detection algorithm. Also this aspect of the layered surface detection is simplified for the exercise, where we only consider cost functions derived directly from pixel intensities.

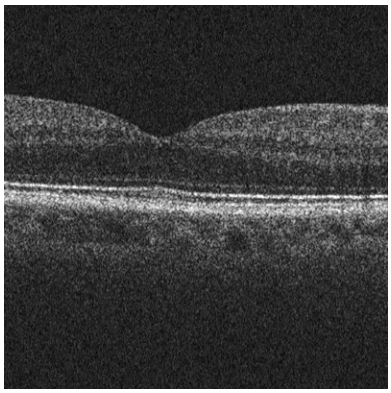

Figure 7.1: OCT (optical coherence tomography) image of retina. Quantifying the thickness of retinal layers informs about eye disease and is therefore of clinical importance.

[1] Xiaodong Wu and Danny Z Chen. Optimal net surface problems with applications. In *Automata, Languages and Programming*, pages 1029–1042. Springer, 2002

[2] Kang Li, Xiaodong Wu, Danny Z Chen, and Milan Sonka. Optimal surface segmentation in volumetric images – a graph-theoretic approach. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(1):119–134, 2006

[3] Mona Haeker, Xiaodong Wu, Michael Abràmoff, Randy Kardon, and Milan Sonka. Incorporation of regional information in optimal 3-d graph search with application for intraretinal layer segmentation of optical coherence tomography images. In *Information Processing in Medical Imaging*, pages 607–618. Springer, 2007

7.1 Layered surface detection

In a discrete volume $x \in \{1,\dots,X\}$, $y \in \{1,\dots,Y\}$, $z \in \{1,\dots,Z\}$, a terrain-like surface s defined by $z = s(x,y)$ satisfies a smoothness constraint (Δ_x, Δ_y) if

$$|s(x,y) - s(x-1,y)| \leq \Delta_x \quad \text{and} \quad |s(x,y) - s(x,y-1)| \leq \Delta_y \,. \quad (7.1)$$

For a cost volume $c(x,y,z)$, an *on-surface* cost of s is defined as

$$C_{\text{on}}(s,c) = \sum_{x=1}^{X} \sum_{y=1}^{Y} c(x,y,s(x,y)) \,. \quad (7.2)$$

The *optimal net surface problem* is concerned with finding a terrain-like surface with a minimum cost among all surfaces satisfying the smoothness constraint.

The polynomial time solution presented in the work by Wu and Chen transforms the optimal net surface problem into a problem of finding a *minimum-cost closed set* in a node-weighted directed graph with nodes representing volume voxels. This is further transformed into a problem of finding a *minimum-cost s-t cut* in a related arc-weighted directed graph. Minimum-cost *s-t* cut can be solved in polynomial time and efficiently found using the algorithm of Boykov and Kolmogorov [4], a well known tool for many image segmentation tasks. While the optimal net surface problem is ultimately solved using the minimum-cost *s-t* cut algorithm, it should be noted that the graph constructed for surface detection is different from the graph used for Markov random fields.

[4] Y Boykov and V Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(9):1124–1137, 2004

The solution to the optimal net problem gives a practical tool for detecting a surface in a volume, or a curve in an image. To use the tool, we need to (somehow) transform the image data into a cost function with the property of having small values in regions where we expect to find the surface. The practical value of the solution to the optical net problem is further increased by two very useful extensions which we describe next.

7.1.1 Multiple surfaces

The extension to multiple surfaces developed in [5] may be exemplified by considering two terrain-like surfaces s_1 and s_2. The surfaces are said to meet an overlap constraint $(\delta_{\text{low}}, \delta_{\text{high}})$ if

$$\delta_{\text{low}} \leq s_2(x,y) - s_1(x,y) \leq \delta_{\text{high}} \,. \quad (7.3)$$

[5] Kang Li, Xiaodong Wu, Danny Z Chen, and Milan Sonka. Optimal surface segmentation in volumetric images – a graph-theoretic approach. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(1):119–134, 2006

Given two cost volumes c_1 and c_2 the total cost associated with surfaces s_1 and s_2 is

$$C_{\text{on}}(s_1,c_1) + C_{\text{on}}(s_2,c_2)$$

and the optimal surface detection will return a pair of surfaces with a minimum cost among all surfaces satisfying overlap and smoothness constraint. Depending on the problem at hand c_1 and c_2 may be different or identical, and likewise smoothness constrains may vary or be the same for the two surfaces.

7.1.2 *In-region cost*

When detecting only one surface s, instead of on-surface cost as in 7.2 we may define a cost for the two regions: the one below the surface and the one above the surface. The cost of the surface s is then given by

$$C_{\text{in}}(s, c_{\text{below}}, c_{\text{above}}) = \\ = \sum_{x=1}^{X} \sum_{y=1}^{Y} \left(\sum_{z=1}^{s(x,y)} c_{\text{below}}(x,y,z) + \sum_{z=s(x,y)+1}^{Z} c_{\text{above}}(x,y,z) \right) . \quad (7.4)$$

That is *one* surface may be found by defining *two* cost volumes: one with small values (dark) below and on the surface, and the other with small values above the surface. In-region cost is evaluated over larger area, making this approach very robust to noise. It is also practical in cases where the boundary between two regions is blurry.

Finally, two set of *layered* (i.e. non-intersecting and ordered) surfaces give rise to an *in-region* cost corresponding to the regions between two neighboring surfaces. So for two surfaces s_1 and s_2 we have

$$C_{\text{in}}(s_1, s_2, c_{1,2}) = \sum_{x=1}^{X} \sum_{y=1}^{Y} \sum_{z=s_1(x,y)+1}^{s_2(x,y)} c_{1,2}(x,y,z) . \quad (7.5)$$

This cost, together with the cost for the region under the surface s_1 and the region over the surface s_2, can be incorporated into the minimization problem [6]. In the text below, we use notation $c_{0,1}$ and $c_{k,k+1}$ for the cost volumes below the first and above the last (k-th) surface.

[6] Mona Haeker, Xiaodong Wu, Michael Abràmoff, Randy Kardon, and Milan Sonka. Incorporation of regional information in optimal 3-d graph search with application for intraretinal layer segmentation of optical coherence tomography images. In *Information Processing in Medical Imaging*, pages 607–618. Springer, 2007

7.2 *Summary*

To summarize, for finding K cost-optimal layered surfaces we need to define

- K on-surface cost volumes c_k, $k = 1, \ldots, K$, and/or
- $K+1$ in-region cost volumes $c_{k,k+1}$, $k = 0, \ldots, K$.

The set of feasible surfaces is given by

- K smoothness constraints $\left(\Delta_x^k, \Delta_y^k\right)$, $k = 1, \ldots, K$ and
- $K-1$ overlap constraints $\left(\delta_{\text{low}}^{k,k+1}, \delta_{\text{hight}}^{k,k+1}\right)$, $k = 1, \ldots, K-1$.

Layered surface detection has found an immediate use for detecting tubular surfaces. The main principle is the fact that a circle $x^2 + y^2 =$

ρ^2 appears as a straight line $r = \rho$ when represented in polar (r, θ) coordinates. Detecting a tubular surface is achieved by representing the volumetric data in a cylindrical coordinate system (r, θ, z) with the longitudinal axis $r = 0$ roughly aligned with the center of the tube. We call this transformation *unwrapping* the volume, and we also say that the volume is sampled along the radial rays. An important practical parameters for unwrapping are the radial and the angular resolution. In the unwrapped representation, the tubular surface is terrain-like and can be defined as $r = s(\theta, z)$. When using layered surface detection for detection of tubular surfaces, additional constraints are added to ensure a smooth transition over $\theta = 0$.

7.3 Constructing cost volumes

The surfaces returned by the layered surface detection algorithm are optimal in terms of the volumetric cost. Therefore, as mentioned, to detect a surface we need to define a cost volume which takes small values where the data $V(x, y, z)$ supports the surface k. This modelling step, crucial for the performance of the algorithm, is fully dependent on the data.

The transformation from image intensities to cost function often involves filtering, computing gradients or cumulative sums, truncating values etc. Finding a suitable cost function may require some expertise.

If the surface to be detected is characterized by a certain voxel intensity v_s, then the cost volume may be defined as $(V - v_s)^2$. More often, the surface divides two regions of different intensities, so cost volume needs to be defined in terms of change of intensity. When computing intensity changes for tubular surfaces, the best approach is to first unwrap the volume, and then compute the change in the r direction.

7.3.1 Examples

Figure 7.2 demonstrates the use of the layered surface detection for detecting a terrain-like curve in an image.

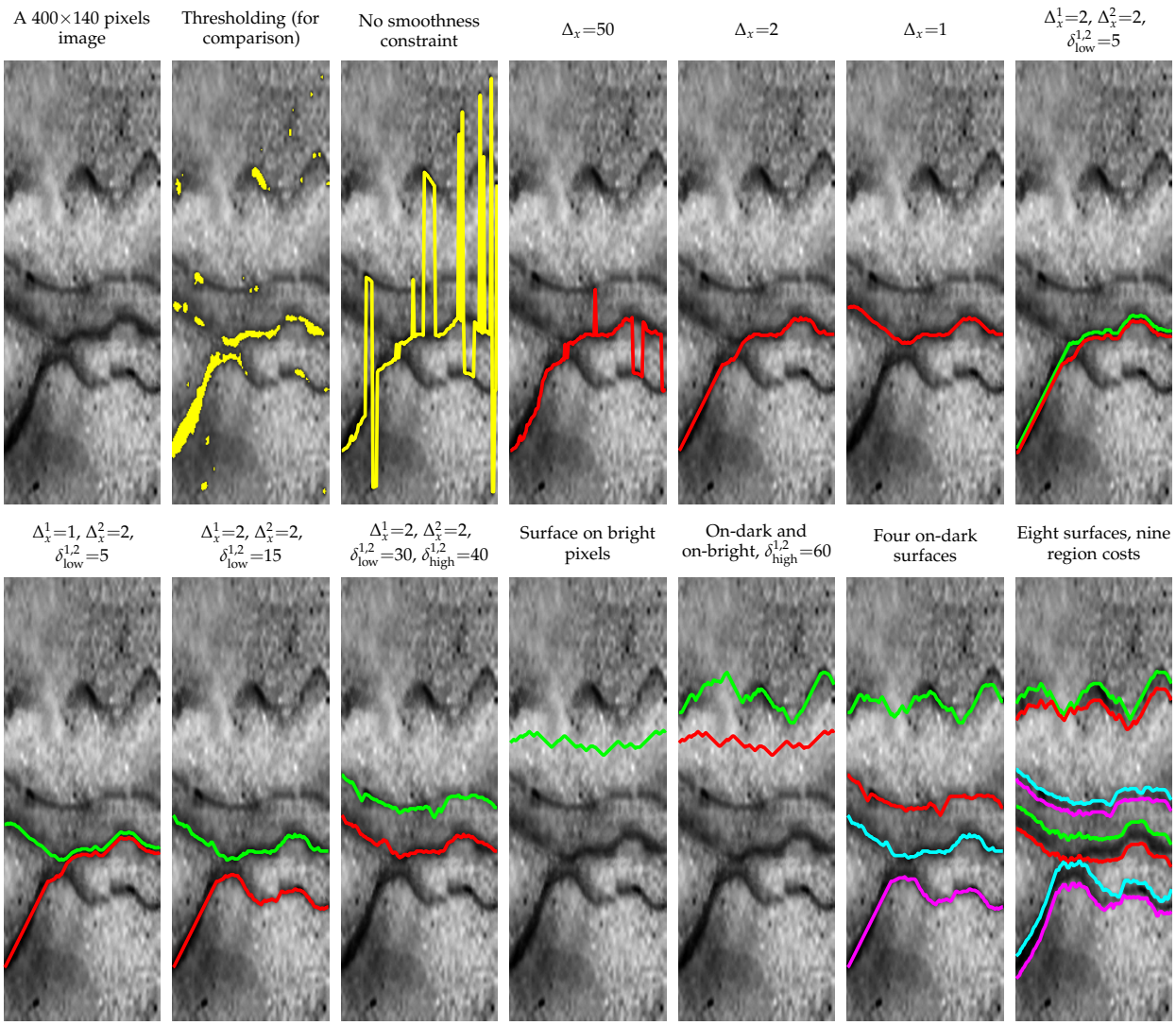

Figure 7.2: Output of layered surface detection. First three images serve to illustrate the problem). Images 4–6 show how changing the smoothness constraint influences the result. Images 7–10 demonstrate the use of the overlap constraint. Images 11–12 demonstrate the use of different cost functions. Image 13 is a four-surface detection, while image 14 uses region costs.

7.4 *Exercise: Layered surfaces in 2D*

In this exercise you will get familiar with layered surfaces. We will only consider layered surfaces in 2D.

Tasks

1. Run the script `on_surface_cost_example` which demonstrates the use of *on-surface* cost. In this case we want to detect dark lines, so for on-surface cost we use image intensities. Get familiar with the functions for computing the optimal solution.

2. Run the script `in_region_cost_example` which demonstrates the use of *in-region* cost. In this case we want to separate the dark and the brigth regions, so for in-regions cost of dark region we use image intensities I, while for bright region we use $255 - I$. Get familiar with the functions for computing the optimal solution and passing the region costs to the solver.

3. Inspect the image `rammed-earth-layers-limestone.jpg`. Use the layered surface detction to detect the darkest line in the image, shown red in Figure 7.3. Then, detect two dark lines (blue and red line in the figure). Finally, detect the lines partitioning the dark regions, as shown in green in Figure 7.3.

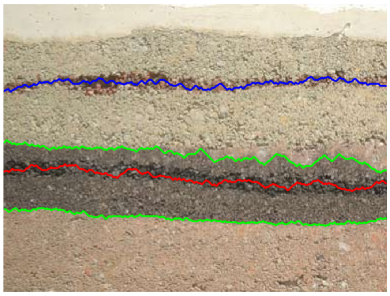

Figure 7.3: Layers in limestone.

7.5 *Exercise: Quantifying dental tomograms*

In this exercise, you will use layered surface detection to solve a concrete image analysis problem. The problem may be solved using different approaches, and you are encouraged to find your own solution. You can therefore interpret tasks below as hints, and you don't need to solve all the tasks.

We will address the problem of quantifying dental tomograms. The success of the dental implants depends on osseointegration, the formation of a direct interface between an implant and bone. An experiment was conducted to experiment how different conditions and treatments affect ossointegration. To assess the outcome of the experiment, we want to measure the interface between an implant and the bone.

For example consider the two slices in Figure 7.4.

Tasks/hints

1. Inspect the data.

2. Use unwapping to handle tubular (circular) data.

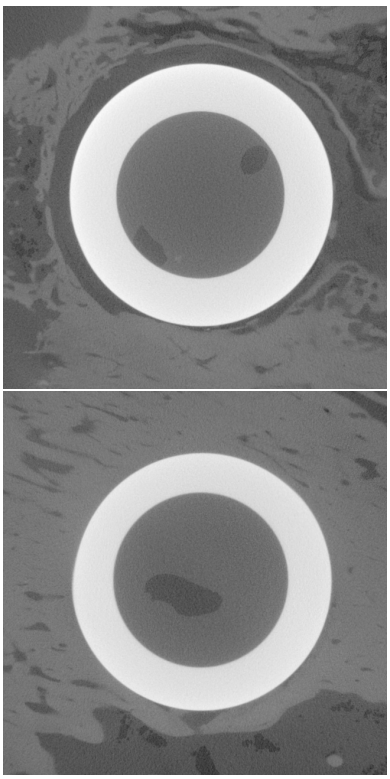

Figure 7.4: Two slices from a dental tomogram with different measure of osseointegration.

3. Detect the layer corresponding to the surface of the dental implant. You may treat the slices individually or as a 3D volume.

4. For one measure of osseointegration consider a curve displaced 20 pixels from the surface of the implant. Express the measure of osseointegration as the percentage of this curve which is passing trough bone. You may use thresholding to divide bone from air (threshold is around 110).

5. For another measue of osseointegration detect a surface (curve) defining the transition from bone to air in vicinity of the implant surface. Quantify the distance between the bone and the implant, for example as the mean distance. You may also produce a histogram of distances.

Part III

Image analysis with neural networks

NEURAL NETWORKS are very useful for a range of image analysis tasks including segmentation, detection, classification, etc. Neural networks are often easy to adapt to a specific problem and they allow approximating an unknown function f^* that, based on some input $\mathbf{x}$, can predict the output $\mathbf{y}$ even without *a priori* knowing the relation between $\mathbf{x}$ and $\mathbf{y}$. This is done by learning a set of parameters θ from a training set, i.e. of corresponding input values $\mathbf{X}$ and predictions $\mathbf{Y}$. In image analysis problems, the input will typically be an image or a part of an image, and the output is a scalar vector or an image.

A range of high-performance libraries for neural networks exists that are very well suited for solving a number of problems also in image analysis. The aim here is, however, to give an understanding of the basic elements of neural networks and get experience with their functionality. This will be done by implementing a feed forward neural network, a Multilayer Perceptron (MLP). The first task is to separate simple point sets. This is not an image analysis task, but it is chosen as a simple and easy to visualize task that can help verifying that the implementation is correct. Furthermore, it will allow some experience with various model parameters. This implementation will later be applied to image classification and image segmentation.

8 Feed forward neural network

A NEURAL NETWORK is often drawn as a directed graph as shown in Figure 8.1. The input layer is shown on the left, hidden layers are in the middle, and the output layer is to the right. This exercise is based on the description in the Deep Learning book[1]. Chapter 5 in the book Pattern Recognition and Machine Learning[2] also gives a good introduction to neural networks.

[1] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016

[2] CM Bishop. Pattern recognition and machine learning., 2006

8.1 Concept of neural network

Deep Learning book (Goodfellow et al.) covers both the fundamentals and the details of the deep learning method. Here, we will give a brief introduction to a simple feed forward network. You will later implement a more general version of a feed forward network.

Conceptually we want to construct a function f that takes a vector $\mathbf{x}$ as input and predicts a vector $\mathbf{y}$

$$f(\mathbf{x}) = \mathbf{y}.$$

In image analysis, the input $\mathbf{x}$ will often be an image reshaped into a vector (elements of the vector are all pixel values). The output depends on the problem to be solved. If we deal with a classification problem, the output will typically be a vector of class probabilities. E.g. if there are k classes, $\mathbf{y}$ will be a k-dimensional vector, where each element in the vector is the probability for belonging to one of the k classes. If we are doing image segmentation into L labels, then the output will be an $D \times L$ matrix where each element is a probability of the i-th pixel belonging to the j-th label where $i = 1, ..., D$ and $j = 1, ..., L$.

There are steps in deep learning that require design choices. This includes choices such as number of layers, number of nodes in these layers and decisions regarding regularization parameters. Such settings are referred to as hyper-parameters of the model (opposed to the edge weights that are the model parameters and will be updated during training).

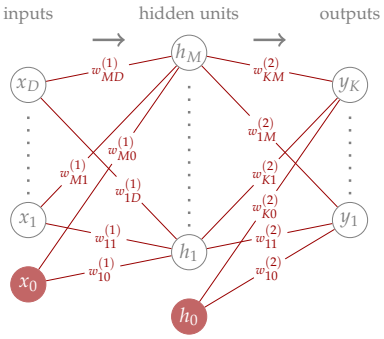

Figure 8.1: Example of a neural network with an input layer, one hidden layer, and an output layer. This is termed a one layer network, since it has one hidden layer. Typically, there will be many hidden layers.

We will start by explaining the *forward model* of the feed forward neural network. Through the forward pass we get the model prediction, which solves our image analysis problem. But in order to construct a model which gives us the desired results, we must first tune the parameters of the model. This is done by changing the edge weights such that the model can correctly predict the values of a training set. When tuning the model parameters we use the *backpropagation algorithm*, which will be explained after the forward model.

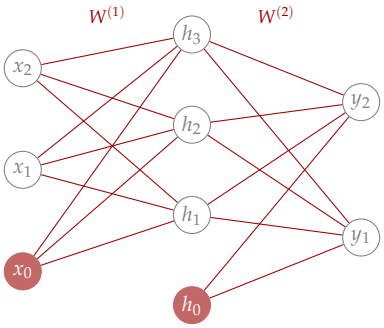

Figure 8.2: Simple three layer neural network.

8.2 *Forward model*

For simplicity, we start by describing the network shown in Figure 8.2. This network will solve a classification problem. The network takes a two dimensional input vector (x_1, x_2) and outputs a probability for two classes. The network contains an input layer of three nodes (also called neurons), where two take the values of the input x_1 and x_2 (independent variables) and one node, called the bias node, is set to $x_0 = 1$. The hidden layer contains four nodes including three nodes connected to the input layer (h_1, h_2, h_3) and the bias node $h_0 = 1$. The output layer contains the two predicted values (y_1, y_2) (dependent variables). The weights of the edges connecting the nodes are termed $w_{ij}^{(l)}$ for the edge connecting node j from layer $l-1$ with node i in layer l.

The values of the nodes in the hidden layers are computed by first computing a weighted linear combination z_i of the node values and the edge weights followed by a non-linear activation function. Here we use the max function $a(z_i) = \max(z_i, 0)$, which in deep learning is called the rectified linear units function (ReLU) to obtain h_i. We have

$$z_i = \sum_{d=0}^{D} w_{id}^{(1)} x_d \,, \tag{8.1}$$

$$h_i = a(z_i) = \max\{0, z_i\} \,, \tag{8.2}$$

$$\hat{y}_j = \sum_{m=0}^{M} w_{jm}^{(2)} h_m \,. \tag{8.3}$$

Since the output of the network should be used for classification, we want to interpret the output values as probabilities, i.e. the element y_j should be seen as the probability of the input belonging to the j-th class. Therefore, we must transform the values $\hat{y}_j$ into positive values summing to 1. For this we use the softmax function

$$y_j = \frac{\exp \hat{y}_j}{\sum_{k=1}^{K} \exp \hat{y}_k} \,. \tag{8.4}$$

When solving classification problem, each input will be put in the output class j^* with the highest value of all y_j.

Note that the output of the network is defined by the set of network weights $w_{ij}^{(l)}$.

8.3 *Backpropagation*

We now want to train the network, i.e. adjust the weights of the network such that the output f gives the desired output. For this, we define a loss function, which is a measure of how different is the current output from the desired output. To measure the loss we need some data for which we know the desired values, also called the target values.

During training, we minimize the loss over a training set of inputs and the associated target values. In training we use the backpropagation algorithm to compute the gradient of the loss function with respect to each weight. Then we use an iterative optimization method called stochastic gradient descent. Just as in gradient descent the loss is minimized by taking steps proportional to the negative gradient.

For a classification problem, we use the cross entropy loss function computed from the predicted value $\mathbf{y}$ and the target $\mathbf{t}$ as

$$L = -\sum_{k=1}^{K} t_k \ln y_k \,, \tag{8.5}$$

where t_k is the target value of the prediction where

$$t_k = \begin{cases} 1 & \text{if class label is } k \\ 0 & \text{otherwise} \end{cases} \,. \tag{8.6}$$

The cross entropy loss gives zero if $\mathbf{y} = \mathbf{t}$ and otherwise a positive value $-\ln y_{k^*}$, where k^* is the target class.

When training the network we use the predictions and targets for all data in our training set. However, when updating weights we do not consider the loss function for all inputs and targets at once. Instead, we consider one input (or a smaller sample of inputs called a minibatch) in a random order (therefore the term stochastic in the name of the optimization). One cycle through the full training dataset is called one epoch.

In each iteration of the stochastic gradient descent we evaluate partial derivatives for a certain (fixed) input in order to determine how change in each $w_{ij}^{(l)}$ affects L. Then we use an update

$$w_{ij}^{(l)\text{new}} = w_{ij}^{(l)} - \eta \frac{\partial L}{\partial w_{ij}^{(l)}} \,,$$

where η is a user-chosen learning rate.

In the derivation below we explain the computation of the partial derivatives using backpropagation. Change in each $w_{ij}^{(l)}$ contributes to

the change in L only trough $z_i^{(l)}$ and using the chain rule the derivative may be separated into two elements

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \overbrace{\frac{\partial L}{\partial z_i^{(l)}}}^{\text{difficult}} \overbrace{\frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}}}^{\text{easy}} .$$

Since $z_i^{(l)}$ is a linear function of $w_{ij}^{(l)}$ the second (easy) partial derivative evaluates to $h_j^{(l-1)}$ (or, in the case of the first layer, input values x_j). So when implementing backpropagation, the node values need to be stored during the forward pass.

The first (more difficult) partial derivative needs to be evaluated for each $z_i^{(l)}$. We denote these values by $\delta_i^{(l)} = \frac{\partial L}{\partial z_i^{(l)}}$, such that we have

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} h_j^{(l-1)} . \tag{8.7}$$

Notice here that the update for weight $w_{ij}^{(l)}$ is a product of two values, the first value depends only on the *to*-node and the second value depends only on the *from*-node.

We still need to evaluate $\delta_i^{(l)}$, i.e. establish how a change of $z_i^{(l)}$ affects L. This depends only on what happens in the layers further down the pipeline, and on the choice of the non-linear activation used on $z_i^{(l)}$. We distinguish between the last layer (where we use the softmax function) and the internal layers.

For the last layer we express L as a function of $z_k^{(l^*)}$

$$\begin{aligned} L &= -\sum_k t_k \ln \frac{\exp z_k^{(l^*)}}{\sum_j \exp z_j^{(l^*)}} = \quad \text{using the properties of ln and the distributive rule} \\ &= -\sum_k t_k z_k^{(l^*)} + \overbrace{\sum_k t_k}^{=1} \ln \overbrace{\sum_j \exp z_j^{(l^*)}}^{\text{equal for all } k} . \end{aligned}$$

The derivative of L with respect to $z_i^{(l^*)}$ is therefore

$$\delta_i^{(l^*)} = -t_i + \frac{1}{\sum_j \exp z_k^{(l^*)}} \exp z_i^{(l^*)} = y_i - t_i . \tag{8.8}$$

Now consider the internal layers. The change in $z_i^{(l)}$ may change all z_k^{l+1}, and any of these changes may affect L. The chain rule gives

$$\frac{\partial L}{\partial z_i^{(l)}} = \sum_k \overbrace{\frac{\partial L}{\partial z_k^{(l+1)}}}^{\text{we have}} \overbrace{\frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}}}^{\text{we need}} .$$

The first set of derivatives are $\delta_k^{(l+1)}$ for the layer further down the pipeline. We already evaluated those for the last layer in (8.8), and this is why we compute the update backwards trough the network. The only remaining is to determine how the change of $z_i^{(l)}$ affects $z_k^{(l+1)}$. From definition (8.1) we see that $z_k^{(l+1)}$ is a linear function of $a\left(z_i^{(l)}\right)$ which gives

$$\frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}} = w_{ki}^{(l+1)} a'\left(z_i^{(l)}\right),$$

where a' denotes the derivative of the activation function, which for ReLU function takes a value zero for arguments smaller than zero, and one otherwise. This is easy to determine by assessing whether $h_i^{(l)}$ is zero or larger. The final expression for internal layers is

$$\delta_i^{(l)} = a'\left(z_i^{(l)}\right) \sum_k w_{ki}^{(l+1)} \delta_k^{(l+1)}. \tag{8.9}$$

8.4 Implementation

You are now ready to implement your neural network based on the following steps.

8.4.1 Setting up the problem

You should implement the network shown in Figure 8.2 using the description given above. This network contains a single hidden layer, it takes a two dimensional input and classifies the input to a two dimensional output.

You also need some test data for running your method. We have provided a function for MATLAB called `make_data.m` and a script for running it called `data_example.m`. In python it is all in the file `make_data.py`. This code will create point-sets similar to the ones shown in Figure 8.3. In this experiment you will visualize the output of the network on the regular grid, so you can use all of the generated points for training. This first experiment is to ensure that you build the neural network in a correct way.

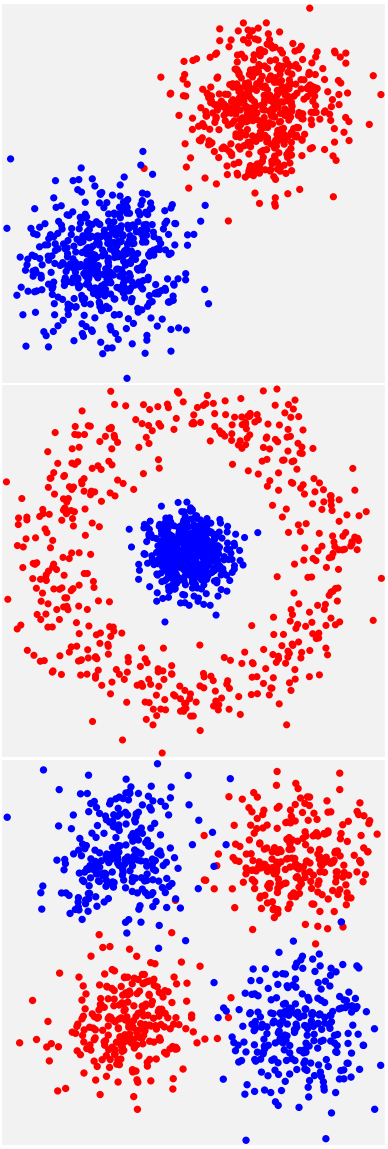

Figure 8.3: Scatter plots of point sets to test neural network

8.4.2 Simple three layer network

You should start with a three layer network with a single hidden layer with five neurons, i.e. four neurons where three are connected to the input layer and one bias neuron. It is a good idea to start making a hand drawing of the network you should implement with the nodes, edges and notation for the parts of the network.

Below is an elaborate description of how you can implement the simple three layer network.

Standardize data To ensure numerical stability of the MLP, it is a good idea to standardize the input data to have zero mean and standard deviation of one in each dimension. Store the mean values and standard deviations of the original points, such that you can transform new input points using these values.

Initialize the weight as arrays Edge weights are best stored as 2D arrays that allow computing node values between layers using vector-matrix multiplications. For the simple three layer network, the weight arrays $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ will be of dimension 3-by-3 for mapping from layer 1 to 2 (input to hidden layers without bias) and 4-by-2 for mapping from layer 2 to 3 (hidden layer to outputs).

It is important that you initialize the weight arrays with random numbers that are scaled adequately. Initializing with zeros is not a good idea, as it introduces no asymmetry between neurons. If the weights are too large or too small, you might risk numerical instability where your network will not converge. The current recommendation in the case of neural networks with ReLU neurons is to initialize with normally distributed random numbers scaled with a factor $\sqrt{\frac{2}{n}}$, where n is the number of inputs to the neuron. I.e., use $n = 3$ when initializing $\mathbf{W}^{(1)}$ and $n = 4$ for $\mathbf{W}^{(2)}$.

Forward model The forward model includes a mapping from the input $\mathbf{x}$ to the hidden nodes with values $\mathbf{z}$. This can be computed as a vector-matrix multiplication

$$\mathbf{z} = [\mathbf{x}, 1]\mathbf{W}^{(1)} , \tag{8.10}$$

resulting in the 1-by-3 vector $\mathbf{z}$. Then the nodes are activated using ReLU, that is $h_i = a(z_i) = \max(0, z_i)$.

From the hidden layer, the value $\hat{\mathbf{y}}$ is computed using a second vector-matrix multiplication

$$\hat{\mathbf{y}} = [\mathbf{h}, 1]\mathbf{W}^{(2)} , \tag{8.11}$$

resulting in a 1-by-2 vector. This is now activated using soft-max

$$y_j = \frac{\exp \hat{y}_j}{\sum_{k=1}^{2} \exp \hat{y}_k} . \tag{8.12}$$

You can implement the forward model as a function that takes the data $\mathbf{x}$ and weights $\mathbf{W}^{(1)}$, $\mathbf{W}^{(2)}$ as input. The function should return the predicted values $\mathbf{y}$, but also the values in the hidden nodes $\mathbf{h}$, because these will be used in the backpropagation. In the backpropagation, we will also need the derivative of the activation function for each neuron in the hidden layer. But those can be obtained from h_i since we know that we use ReLU activation, such that we have $a'(z_i)' = 1$ if $h_i > 0$ and otherwise $a'(z_i) = 0$. So you don't need to store the values of z_i.

Implementing backpropagation The backpropagation will run iteratively, where the weights $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ are updated in each iteration. You will start each iteration by computing the output of the forward pass including $\mathbf{y}$, $\mathbf{h}$, and $\mathbf{z}'$. From this, you can compute the loss based on the current weights.

We will now start from the output and compute the parameters needed to update the weights. First we compute the 1-by-2 vector

$$\delta^{(2)} = \mathbf{y} - \mathbf{t} \,. \tag{8.13}$$

where $\mathbf{t}$ is the target values. From $\delta^{(2)}$ we can compute the partial derivatives needed for updating $\mathbf{W}^{(2)}$ as a matrix (outer) product

$$\mathbf{Q}^{(2)} = [\mathbf{h}, 1]^T \, \delta^{(2)} \,. \tag{8.14}$$

Note here that $\mathbf{Q}^{(2)}$ is a 4-by-2 matrix, and its elements are are partial derivatives

$$q_{mj}^{(2)} = \frac{\partial L}{\partial w_{jm}^{(2)}} \,. \tag{8.15}$$

We now move on compute the values $\delta^{(1)}$ for the the hidden layer as

$$\delta^{(1)} = \mathbf{a}' \otimes \hat{\mathbf{W}}^{(2)} \left(\delta^{(2)}\right)^T \,, \tag{8.16}$$

where $\hat{\mathbf{W}}^{(2)}$ is the 3-by-2 weight matrix based on $\mathbf{W}^{(2)}$ where the last row (corresponding to bias) has been omitted. The vector $\mathbf{a}'$ is a 3-by-1 vector containing derivatives of the activation in the hidden layer, i.e. it has elements 0 or 1. The symbol $\otimes$ denotes element-wise multiplication.

We can now compute the partial derivatives needed for updating $\mathbf{W}^{(1)}$ as a matrix (outer) product

$$\mathbf{Q}^{(1)} = [\mathbf{x}, 1]^T \, \delta^{(1)} \tag{8.17}$$

Now we update the weights

$$\mathbf{W}^{(1),\text{new}} = \mathbf{W}^{(1)} - \eta \mathbf{Q}^{(1)} \,, \tag{8.18}$$

$$\mathbf{W}^{(2),\text{new}} = \mathbf{W}^{(2)} - \eta \mathbf{Q}^{(2)} \,. \tag{8.19}$$

You can implement the backpropagation as a function that takes the data point $\mathbf{x}$, weight matrices $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$, target value $\mathbf{t}$, and learning rate η as input and returns the updated weight matrices.

If you implement the forward model and backpropagation as described here using vector-matrix multiplications, you will also be able to input data points $\mathbf{x}$ and targets $\mathbf{t}$ as m-by-2 arrays. When you do this, you should remember to add values for bias nodes as m-by-1 arrays of ones. Here the forward model will return $\mathbf{y}$ as an m-by-2 array and $\mathbf{h}$ as an m-by-3 array.

Test your implementation The functions `make_data.m` or `make_data.py` returns points for training and test points on a regular grid. If you transform these point as you standardized the training points, you can pass the test points through the forward model, and use the ouput for illustrating the prediction by showing the result as an image.

Batch optimization A variant of stochastic gradient descend splits the training data into smaller subsets (minibatches) and updates weight according to the average of the gradients for the minibatch. One cycle through the entire training dataset is called a training epoch.

8.4.3 *Variable number of layers and hidden units*

In the exercise above, you have obtained a neural network with a fixed architecture, i.e. the number of neurons and hidden layer. The architecture is however central in modeling with neural networks, and therefore you should make the number of layers and the number of neurons in each of the hidden layers a part of your input choice. You will also be needing this flexibility in the later exercises for classifying the MNIST handwritten digits.

Suggested procedure

1. Implement a neural network keeping the single hidden layer and with variable number of hidden units.
2. Extend this by implementing a neural network with variable number of hidden layers and with a variable number of hidden units in each of these layers. The number of layers and hidden units are given as input to your function. Now you can play around with your model and from here it is easy to modify it to include other elements like other activation functions or minibatches.
3. Again test your implementation on the given data.

The expected output is shown in Figure 8.4. The coloring of the pixels in the background is obtained by passing points on the regular grid through the forward model and coloring the result in dark or bright gray depending on the classification result.

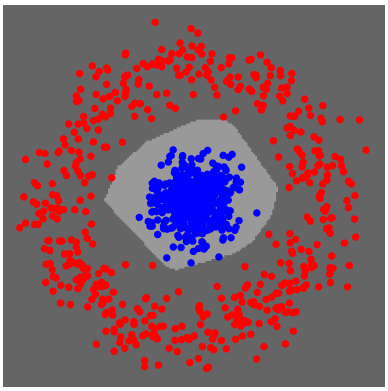

Figure 8.4: Result of training on the input data shown in colored points, and the test result is shown in the pixel colors in the background.

9 Image classification

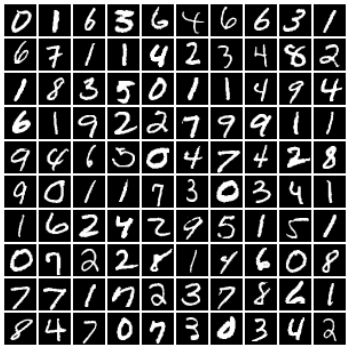

Figure 9.1: Example of the MNIST images.

[1] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009

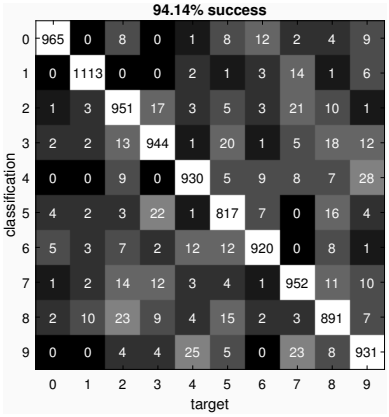

Figure 9.2: Table showing a classification performance example of a classification of the MNIST handwritten dataset.

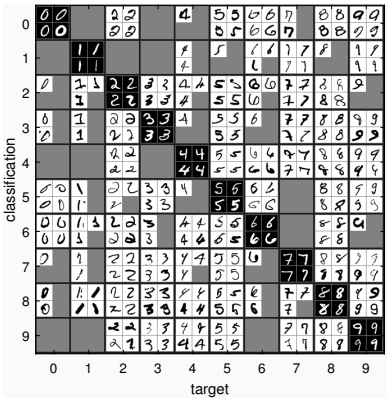

Figure 9.3: Examples of classified handwritten digits and misclassified digits.

In this exercise you should build neural networks for classifying images. We will work with the relatively easy MNIST image data set and the more challenging CIFAR-10[1] data set. Performance will be measured in number of misclassified images, and the goal is to obtain the result with the least number of misclassified images.

The MNIST dataset contains images of handwritten digits of 28×28 pixels as shown in Figure 9.1. Ground truth class labels are given together with the images. The ground truth is 10 dimensional vectors with 1 in the dimension representing the class containing the digit and zeros elsewhere. MNIST contains 60000 images for training your network. If you use all 60000 images for training your network, you might overfit your model, and to choose when to stop training you can split the data into a training and validation set. You can for example use 50000 images for training the network and reserve 10000 for validating it. By classifying the validation images, you can measure if you have overfitted your model, which is seen by a drop in classification performance of the validation data. In addition there are 10000 images for testing, but these should only be used for evaluating the performance of your networks after they have been trained. Figure 9.2 and 9.3 show a classification performance example and examples of correctly and misclassified digits.

The CIFAR-10 data set is similar to MNIST and contains 50000 small images for training and 10000 images for testing. The images are, however, 32×32 RGB color images with significantly higher variation in appearance.

The rules for the competition are:

1. Implement your own neural network for classifying the MNIST images.
2. Train the neural network on a part of the training data (e.g. 50000 images) and validate it on another part (e.g. the remaining 10000 images).
3. When you are satisfied with the obtained result – upload the trained

network together with Matlab or Python code for running it.

4. Hand in a description of your network and a guide on how to run your code

5. Be a fair player and do not use the MNIST and CIFAR-10 test data for training your networks (can be found on the internet, but do not use it!).

6. Do an effort in making the code efficient.

9.1 *Modifications of your network*

The classification will be using a fully connected feed forward neural network, similar to the one you implemented last exercise. But in contrast to last exercise, where data was points in two dimensions, you now have images. We treat these as vectors, so even though MNIST images are only 28×28 pixels the vector representation is 784 dimensions. Therefore, the network should take in 784 dimensional vectors and return a 10 dimensional vector for classifying the digits 0 to 9. For the CIFAR-10 data, the input vector becomes 3072 dimensional ($32 \times 32 \times 3$). You can use the book on deep learning by Goodfellow et al. [2] to get ideas for this exercise.

[2] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016

Obtaining a strong classifier requires many iterations of the backpropagation algorithm using 50000 training images. Therefore, it is a good idea to utilize vectorized code in your implementation. This can be done by computing the gradients for subsets of the training data using minibatches. You can have a minibatch as a matrix and compute the forward propagation and gradients using matrix operations. By averaging the gradients obtained from the minibatch, the backpropagation can be carried out in the same way as you would do when training with one sample at a time. Due to the averaging, the obtained gradients are less affected by noise and it is typically possible to have higher learning rates.

A part of optimizing the neural network is by changing its architecture. Therefore, it is recommended that you implement your network such that you can change the number of layers and the number of neurons in each layer.

1. Implement a fully connected neural network for image classification. The data should be normalized and centered, i.e. vectors of unit length using the 2-norm and with zero mean. Besides vectorizing the code it is also worth considering the data type. Single is faster than double, so you should consider if you want faster computations, at the cost of lower precision. You can experimentally evaluate if the high precision is necessary.

2. Train the network and plot the training and validation error for each iteration (epoch). The dataset could be split into 50000 images for training and 10000 for validation.

9.2 *Optimizing the neural network*

A large number of techniques for optimizing the performance of the neural network has been proposed. Neural networks are typically initialized with random numbers, but the performance depends on the choice of the initialization strategy. You can therefore optimize the network by experimenting with different initialization strategies.

Optimization with stochastic gradient decent can be slow, but updating the gradients using momentum can accelerate the learning rate. Momentum is obtained by computing the gradients as a weighted combination of the previous gradient and the new gradient. Hereby, the gradient is computed as a moving average with exponential decay. Another way of ensuring convergence of the gradient decent is by adapting the learning rate. Here you can adapt the learning rate to the individual gradient estimates.

1. Implement one or more optimization strategies and document how it affects the obtained result.

9.3 *Regularization methods*

It is important that the neural network generalizes well such that it can classify new unseen data. Since neural networks often have many parameters it is easy to overfit the model, especially on small datasets where a very low training error can be obtained, but the validation error will be high. One way to overcome the problem with training a neural network on small datasets is through dataset augmentation, where fake data is fabricated by small modifications of the input data. This can be done by small permutations or by adding small amounts of noise. Hereby a much larger dataset can be obtained, which can help the training.

Instead of adding noise to augment the training data, small amounts of noise may be added to the hidden units in the network. You can add random noise in each minibatch iteration. Noise can also be added to the output targets for obtaining better performance.

Dropout is another method for regularizing the neural network. Here a random selection of neurons are set to zero during each minibatch iteration leaving out these neurons in that iteration. Setting the neurons to zero resembles having a number of different neural networks and is inspired by ensemble methods.

1. Try experimenting with regularization methods. You can also get inspired by architectures that other people have had success with.

10 Convolutional neural networks

Convolutional neural networks (CNNs) have many aspect in common with multilayer perceptrons (MLPs – fully connected feed forward neural networks) such as being a directed acyclic graph with weighted edges and non-linear activations. CNNs use convolutions, or more precisely correlation filters, and the weights of the convolution kernels are the parameters that are optimized when training the network.

Since the edge weights of the CNN are implemented using convolutions, many of the edge weights have the same value. This is called weight sharing, which means that only one edge weight is learned for many edges. This results in a significant reduction in number of model parameters, and therefore makes it possible to have much larger input data compared to MLP networks. Furthermore, the use of convolutions is very efficient, and is used in both the forward pass and the backpropagation. During backpropagation, the weights of the convolution kernels are updated.

Working with images in 2D makes it possible to apply additional operations to the hidden layers. This includes for example a pooling step typically combined with a down-scaling step. Max-pooling is an example of a widely used pooling method, where pixels are replaced by the maximum in a local neighborhood. Max-pooling ensures that only the important features are kept, and makes the analysis robust to small translations. There is a number of other operations that can be applied, and an overview is given in chapter 9 in Goodfellow et al.[1].

Despite that CNNs have many aspects in common with MLPs, they are typically more complicated and therefore not as simple to implement. A large number of software frameworks are available, and training neural networks for many engineering applications will involve GPU processing. This is, however, implemented in a user friendly way in many of the software packages and highly sophisticated neural networks can be developed using high-level programming using e.g. python that makes these frameworks easy to use. We will work with PyTorch and get access to GPUs (Graphics Processing Units) using

[1] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016

Google Colab.

10.1 *Exercise*

In this exercise we will work with a model for segmentation, which has a structure similar to the U-net[2]. Our model is implemented in PyTorch and requires that you code in python. Setting up the model and making sure that everything runs as expected is time consuming, so we have prepared a Google Colab notebook for you.

The U-net is designed for image segmentation and is a Fully Convolutional Network (FCN), meaning that all operations from the input to the output are convolutions. The name sounds like a fully connected network (MLP), but it is not the same. The FCN is a CNN containing convolutions, pooling, up-convolutions, and skip connections. The purpose of this exercise is to work with our segmentation network to understand the function of all the elements in the network, such that you will be able to use and modify the network for your application.

We will work with segmenting microscopy data of glands from patients with colon cancer and healthy controls. The microscopy images are shown in Figure 10.1. The data has been reshaped into 128 by 128 images in RGB and the labels are binary with the value 0 in the background and 255 in pixels labeled to contain glands. The task is to train our CNN to take an RGB image as input and give an image with labels as output.

To carry out the exercise, you need a Google account. As part of your Google account, you get access to Google Colab, where it is possible to run scripts in a notebook interactively. There is also access to GPU, which speeds up the training. Google Colab is especially suited for exercises, where the analysis problems are relatively small. If you have a larger project it is recommended that you use other computational resources.

The data for training the model is downloaded to your Google Colab account, but there are some test data that you can get from the data homepage.

[2]

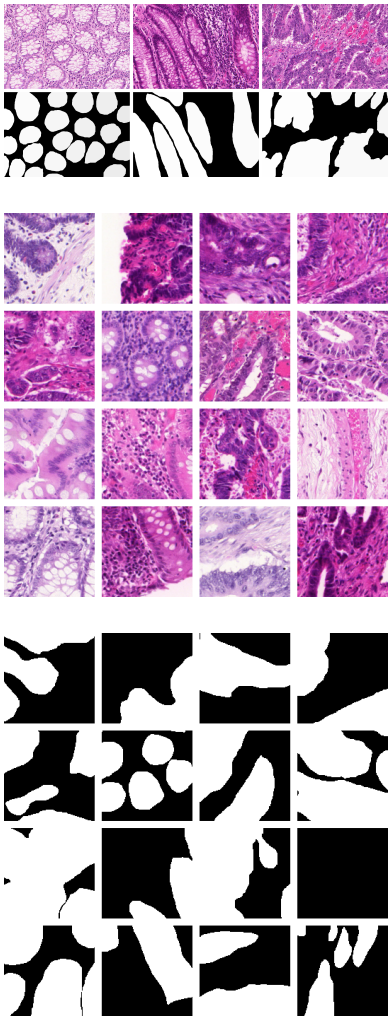

Figure 10.1: Data examples for the segmentation exercise. The top image shows an example of the original data. The task is to segment individual glands in the histological colon tissue. We have down-sampled to half size and cropped out 128 by 128 images for this exercise. The cropped images and their labels are shown in the middle and bottom. Data from: https://warwick.ac.uk/fac/cross_fac/tia/data/glascontest/

10.2 *Suggested procedure*

The exercise contains some steps to investigate the model. First you will investigate a model with pretrained weights and later you will train the model yourself. Since the model and the dataset is relatively small, the model can be trained within some minutes.

Taks 1: Load the model First you should follow the link to the Google Colab notebook and run the cell that downloads the data. After that

you should run the cells that load the data, set up the model, and load the pretrained model.

Task 2: Model overview To have an overview of the model, it is a good idea to make a sketch of the model on a piece of paper, which is similar to the drawing of the U-net. This means drawing boxes for the input image, internal neurons (images inside the network), the output image, and write up their dimension. Also draw arrows illustrating how layers are connected.

Through this drawing you will see what the output image size is if you have the 128 by 128 input image. The model can also take images of other sizes, but not all sizes are valid. What is required for sizes of images in this model?

Task 3: Run the model You can run the model on a test image that you can get from the data homepage. These images has the original shape, so you need to crop out a size that fits the model.

In cases where you have a large image, you will sometimes need to crop out smaller images and segment them individually. To see the effect of this, you should take one of the test images and crop it into 128 by 128 images and segment them individually and put them together into a full image. You can choose to have the segmentations overlapping to smooth out boundary effects.

Visualize how a segmentation with the largest crop compares to a segmentation assembled from smaller crops.

Task 4: Weights You can look at the weights in the trained convolutional kernels and you can try to visualize some of them. They are, however, very small, so they might not be very informative. You should also investigate the size of the convolutional kernels. What happens to their sizes as you go deeper in the network? What is the number of learnable parameters?

Task 5: Training You should now train the model. First you will create random model parameters by creating a new model object. What is the output, when you have not trained the model?

You should run the model for a number of epochs. When should you stop training? How many epochs are needed?

You can try changing the learning rate, and see how that affects the model.

Task 6: Modify the model You can try changing the model. For example you can try another optimizer, change the activation functions, or modify the architecture of the model.

You can also try working with data augmentation and dropout etc.

Part IV

Mini projects

11 Texture analysis

Image texture is important for a range of image analysis problems like object classification and quality control. Also a number of image processing problems like denoising and inpainting are based on principles of texture analysis. Here you will solve a texture classification based on the Basic Image Features described in Crosier and Griffin[1] and an inpainting problem described in the paper by Efros and Leung[2].

[1]

[2]

11.1 Data

The data for the exercise is found in the file called `texture_data.zip` that contains images with a wide variety of textures for BIF characterization used in the first part and corrupted images to be used during the texture synthesis part of the exercise. Examples images are shown in Figure 11.1. You are also welcome to find your own data set for the exercise.

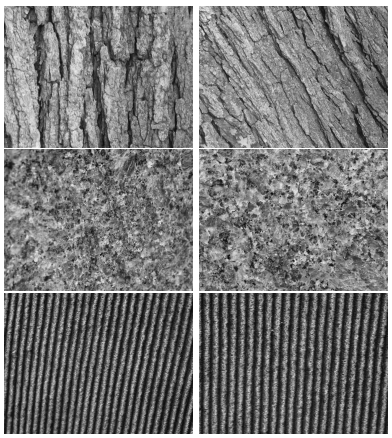

Figure 11.1: Examples of three classes of textured images.

11.1.1 Basic Image Features

This section will provide a small summary of how BIFs are estimated. The purpose of BIF is to go from an image of high dimensionality to a lower dimensional vector representation of the image texture. This representation uses simple geometric image features and will enable differentiation between different textures. The following recipe is used to estimate BIF:

1. Convolve with six Gaussian filters to get scale-normalized filter responses $(s, s_x, s_y, s_{xx}, s_{yy}, s_{xy})$.
2. Calculate the flat, slope, blob (2×), line (2×) and saddle feature responses using the formulas from[3] and $(s, s_x, s_y, s_{xx}, s_{yy}, s_{xy})$ from Step 1.
3. Classify each pixel as flat=0, slope=1, dark blob=2, white blob=3, dark line=4, white line=5, saddle=6, by finding the label index of the maximum feature responses of Step 2. Denote the resulting label image as $\mathcal{L}$.

[3]

For a fixed scale, a seven bin histogram can now be formed by counting how many times a pixel is classified as one of the classes. This histogram is the BIF of an image for scale σ. Please note that if we discard the flat pixels, a six bin histogram is used per scale, however we are generally interested in a histogram that also models scale.

Four scales: How to get a histogram? When extending this BIF representation to multiple scales, the process of forming a histogram becomes increasingly complicated. If we choose four scales $\sigma = (1, 2, 4, 8)$ we need to run steps 1 - 3 four times. This will lead to a four channel label image $\mathcal{L}(\mathbf{x}; i)$, where $\mathbf{x}$ is the position in the image and $i = 0, ..., 3$ for the four scales. To get the texture characterization, we have to convert these four channels of the label image into a histogram. Each bin of this histogram counts how often a specific label configuration occurs across the four scales. If we ignore flat BIF regions, the pixels can be classified as one of the labels $\mathcal{L}(\mathbf{x}; i) \in \{1, \ldots, 6\}$. First we want to translate this to a number between 0 and 1295 (i.e. $6^4 = 1296$ unique combinations). This is done in all pixels resulting in an image $\mathcal{B}(\mathbf{x})$ by converting the four BIF classes to one number

$$\mathcal{B}(\mathbf{x}) = \sum_{i=0}^{3} (\mathcal{L}(\mathbf{x}; i) - 1)6^i \, . \tag{11.1}$$

This means that the label combination $[1, 1, 1, 1]$ is a pixel classified as slope on all the scales, and similarly $[1, 3, 4, 6]$ is a pixel that is classified as a slope at the first scale, a blob on the second scale, a line on the third scale, and a saddle on the fourth scale.

11.2 *Texture classification*

In this exercise you will experiment with *Basic Image Features* (BIF) for texture description.

The BIF features are computed from the following equations:

Classify according to the largest of the features:

Flat: ϵs
Slope: $2\sqrt{s_x^2 + s_y^2}$
Blob: $\pm\lambda$
Line: $2^{-\frac{1}{2}}(\gamma \pm \lambda)$
Saddle: γ

where

$\lambda = s_{xx} + s_{yy}$
$\gamma = \sqrt{(s_{xx} - s_{yy})^2 + 4s_{xy}^2}$

Suggestions for experiments:

- Illustrate the BIF response in some images using color codes similar to how this is done in Crosier and Griffin[4]. [4]
- Show the BIF histogram (set $\epsilon = 0, \sigma \in \{1, 2, 4, 8\}$) for an example image.
- Compare BIF histograms for a total of 30 images (6 texture classes, 5 images per class). Construct a 30×30 *confusion matrix* containing the histogram distances based on the L_1-norm (sum of absolute difference). Show the histogram as an image and explain the pattern.

11.3 *Texture synthesis: Task 2*

In this exercise you will synthesize image texture using a method similar to the one presented in[5]. This method is based on fitting partly [5] overlapping image patches to an image with holes by sampling (randomly) from a distribution of similar image patches. The distribution is approximated from the image itself by measuring distances to patches from the image itself.

Suggestions for experiments:

- Choose or construct a simple test example with repeated texture and a small hole and fill in the missing part.
- Choose a natural image and fill in a hole. Try varying number of patches, patch size, hole size, type of image, etc.
- Can the method be used for noise reduction? Experiment with e.g. salt and pepper noise.

12 Optical flow

Small movements between two consecutive frames of an image series can be modeled as optical flow. The problem of optical flow is to determine local translations between two frames as a vector field such that the brightness constancy constraint

$$I(x, y, t) = I(x + \Delta x, y + \Delta y, t + \Delta t), \tag{12.1}$$

is fulfilled. Here, x and y are spatial coordinates and t is time. In this exercise you will implement methods for computing optical flow of the movement between two images.

A number of algorithms have been suggested for solving the flow problem, and a simple solution is to match patterns locally using block matching, where a window around a point in the first image is translated and compared to a window in the other image using e.g. sum of squared differences. This is however a time consuming task, so other methods based on computing differentials have been suggested. These include the Lucas-Kanade[1] and the Horn-Shunck methods[2] that you will work with in this exercise. This exercise is based on the book Computer Vision: Algorithms and Applications[3], Chapter 8 (can be downloaded from http://findit.dtu.dk/). But you may also find relevant information from the internet and the two original papers.

[1]
[2]
[3]

In this exercise, different approaches for solving the optical flow problem are given, and it is suggested that you start by working with the basic elements of *Optical flow* and then try working with the Lucas-Kanade method or the Horn Shunck method and preferably both. These methods have a number of common elements, so when one is implemented it is relatively easy to implement the other.

12.1 *Optical flow*

The assumption behind optical flow is that the movement between frames is small. Therefore, the optical flow can be computed by

$$\frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v = -\frac{\partial I}{\partial t}, \tag{12.2}$$

where u and v are displacements in the horizontal and vertical directions respectively. $[u, v]^T$ is the velocity of the flow field.

Task: Derive (12.2) form (12.1) by using a first order Taylor approximation.

In (12.2) two unknown parameters (u, v) must be computed, but in a single pixel there is just one equation, so the problem is underdetermined. If a small neighbourhood around a pixel is assumed to have the same displacement, it will be possible to solve (12.2) as a linear least squares problem, where we have $\mathbf{Au} = \mathbf{b}$, where $\mathbf{u} = [u, v]^T$.

Task: Write up the elements of $\mathbf{A}$ and $\mathbf{b}$ for a 3×3 neighbourhood.

Two small images of 10×10 pixels called `composedIm_1.png` and `composedIm_2.png` are available in the `optical_flow_data.zip` file on Campusnet. They are made from an image by extracting two patches shifted by one pixel. Furthermore a 3×3 region of the same pattern is placed in the image, but shifted in the opposite direction as shown in Figure 12.1. You can use these two images to try simple experiments with optical flow.

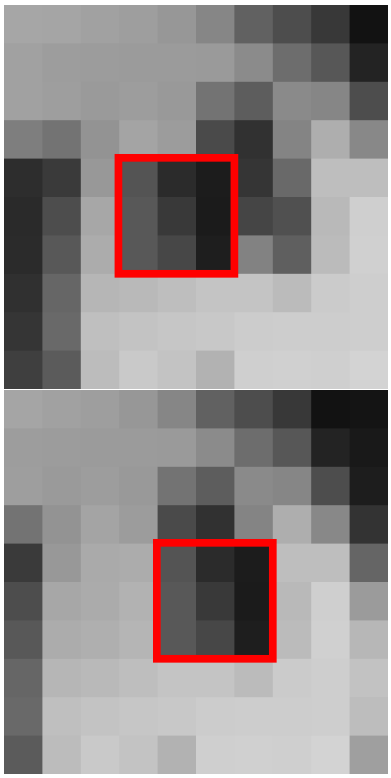

Figure 12.1: Two images of the same patten shifted one pixel to the left, whereas the center part marked in the red box is shifted on pixel to the right.

Task: Compute the optical flow vector for a window of 3×3 pixels centered at $(r, c) = (2, 6)$ and $(r, c) = (5, 4)$, where r is the row and c is the column. You should use a simple pixel differences to compute the differential by using the central difference filters $[-1, 0, 1]$ and $[-1, 0, 1]^T$. You can also use $[-1, 1]$ and $[-1, 1]^T$, but then the derivative is computed between pixels. You can ignore this and compare the result to the central difference filter.

The least squares solution to $\mathbf{Au} = \mathbf{b}$ is found by solving the minimization problem

$$\arg\min_{\mathbf{u}} \|\mathbf{Au} - \mathbf{b}\|^2. \tag{12.3}$$

Taking the derivative with regards to $\mathbf{u}$ and setting to zero yields

$$\mathbf{u} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}. \tag{12.4}$$

This allows us to precompute $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}^T\mathbf{b}$ as a number of sums over the image that can be obtained efficiently by filtering.

Task: Write up the elements of $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}^T\mathbf{b}$.

Task: Precompute the input needed for $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}^T\mathbf{b}$ and implement a filtering scheme that sums the elements for the small test images `composedIm_1.png` and `composedIm_2.png`. Compute the flow vectors for the full images.

Task: Display the flow vectors on top of the images using the MATLAB function `quiver`.

12.2 *Lucas-Kanade method*

In this and the next part you should experiment with larger images. There are some benchmark images available from the Middlebury benchmark homepage `http://vision.middlebury.edu/flow/`. Some images from here have been uploaded to CampusNet that you can use for this exercise. But you are also welcome to experiment with your own images.

What you implemented in until now is a simple version of the Lukas-Kanade method. When you are working with larger images, there are however some practical aspects to consider. The linear system $\mathbf{Au} = \mathbf{b}$ may be ill posed such that there are no vector $\mathbf{u}$ that fulfills the equation. If this is the case, the 2×2 matrix $\mathbf{A}^T\mathbf{A}$ does not have an inverse.

Task: Find a way to check if there is a solution to the equation $\mathbf{Au} = \mathbf{b}$, i.e. that $\mathbf{A}^T\mathbf{A}$ has an inverse. Implement that in your solution for computing the optical flow vector field.

Task: Compute and display the flow field in two larger images from e.g. the Middlebury dataset.

In the first part of this exercise you implemented the image differential using pixel differences. There are also other methods for computing image differentials like central differences using the filter $[-1, 0, 1]$ and its transpose. You may also the first order derivative of a Gaussian.

Task: Test one or more differential filters.

Instead of treating all pixels in the window equally, better results can be obtained by weighting the pixels using a weight matrix $\mathbf{WAu} = \mathbf{Wb}$ where $\mathbf{W}$ is a diagonal weight matrix, resulting in the least squares solution $\mathbf{u} = (\mathbf{A}^T\mathbf{WA})^{-1}\mathbf{A}^T\mathbf{Wb}$. This can efficiently be implemented using a wight filter e.g. a Gaussian.

Task: Implement a weighted sampling window as a filter operation. Display the result on test images using different filter size.

Task: Comment on performance and processing time for the Lucas-Kanade method.

12.2.1 *Horn-Shunck method*

A problem with the Lucas-Kanade method is that it operates locally, so in regions with no texture it is not possible to compute the flow vectors. This is solved in the Horn-Shunck method where the Laplacian over the vector field is minimized in addition to ensuring that the brightness is constant by minimizing

$$E = \int\int [(I_x u + I_y v + I_t) + \alpha^2(\|\nabla u\|^2 \|\nabla v\|^2)] \mathrm{d}x\mathrm{d}y. \tag{12.5}$$

The energy E is minimized by iteratively updating the flow vectors using the update rules

$$u^{k+1} = \bar{u}^k - \frac{I_x(I_x\bar{u}^k + I_y\bar{v}^k + I_t)}{\alpha^2 + I_x^2 + I_y^2} \tag{12.6}$$

$$v^{k+1} = \bar{v}^k - \frac{I_y(I_x\bar{u}^k + I_y\bar{v}^k + I_t)}{\alpha^2 + I_x^2 + I_y^2} \tag{12.7}$$

where $\bar{u}^k$ is the average flow over a window in the x-direction and

$$I_x = \frac{\partial I}{\partial x}\,, \quad I_y = \frac{\partial I}{\partial y}\,.$$

Task: Implement the Horn-Shunck method and test it on two larger images from e.g. the Middlebury dataset. Display the flow vectors.

The choice of the α parameter and the number of iterations influence the smoothness of the obtained result. Also, the choice of how the image is differentiated will affect the performance.

Task: Display results with varying parameter choices. Display results of a good and a bad choice of α. Do the same for number of iterations and size of averaging.

Task: Experiment with different choice of differentiation method.

Task: Comment on performance and processing time for the Horn-Shunck method.

13 Nerve segmentation

This project is on the practical aspects of volumetric segmentation using geometric priors. This is demonstrated on a problem of segmenting myelinated axons from the volumetric data containing scans of the human posterior interosseous nerve.

There are various strategies for solving this problem. Here we demonstrate the use of two approaches we already presented: Markov random fields introduced in Chapter 5 and defromable models introduced in Chapter 6. Furthermore, an approach from Chapter 7 would be very suitable for segmenting nerves (but not explained in this project).

You will be working on the small part of the large data set. For better understanding of the goals of image analysis, we provide background information on the large study involving a complete data set.

13.1 X-ray tomography of human peripheral nerves

Nerve disorders caused by trauma or disease can have serious consequences for the affected people. One aspect of understanding nerve disorders involves a knowledge of the structure of peripheral nerves and their subcomponents. This is typically obtained trough microscopy.

Imaging using conventional light and electron microscopical techniques only allows a two-dimensional visualization of tissues such as peripheral nerves. With recent advances in synchrotron imaging techniques, we can now also obtain detailed three-dimensional images of tissue. This in turn allows extraction of 3D morphological information.

For a larger study[1], biopsies of the posterior interosseous nerve at wrist levels were taken from otherwise healthy subjects and from subjects with type 1 and 2 diabetes. The aim of the study was to determine whether diabetes influences the radius, trajectory and organization of myelinated axons in human peripheral nerves.

[1]

Biopsies were stained in osmium (a heavy metal used for staining lipids) which provides contrast to the image, and embedded in Epon (epoxy resin) for stability. The samples were then imaged using X-

ray phase contrast zoom tomography at the European Synchrotron Radiation Facility (ESRF, Grenoble, France) with an isotropic voxel size of 130nm. In the obtained volumetric data, the nerve fibers are aligned with the z direction, and the stained myelin sheaths around axons (see Figure 13.1 for a schematic drawing of a nerve) appear circular in the x-y slices through the volume, as shown in Figure 13.2.

Complete data-set contains more than 10 samples, each resulting in a volume of a size $2048 \times 2048 \times 2048$ voxels. For the exercise, we extracted a small region from one volume, as indicated in Figure 13.2. Furthermore, extracted volume has been downsized by a factor 2, which yields a volume of size $350 \times 350 \times 1024$. The extracted volume is saved as a stacked tiff image `nerves_part.tiff`.

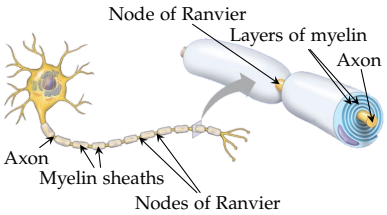

Figure 13.1: A schematic drawing of a nerve showing an axon, myelin sheaths and nodes of Ranvier.

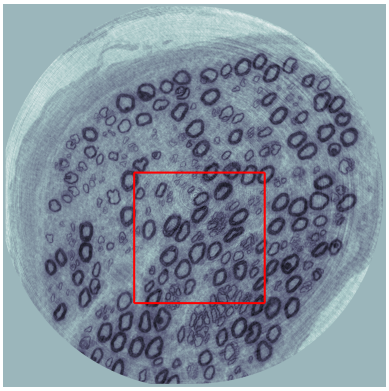

Figure 13.2: One slice from the volume showing peripheral nerves, and a region which was extracted for the exercise.

13.2 *Segmentation of myelinated nerves*

A good contrast between stained myelin sheaths and the background makes it possible to clearly distinguish individual nerve cells in the volume. For this reason, a reasonable segmentation strategy would utilize dark appearance of the myelin. Segmentation may be improved by incorporating a prior knowledge about the directionality of the nerves.

Furthermore, circular appearance of myelin sheaths allows a segmentation of a single nerve cell by aligning a closed curve with the periphery of the myelin. For this, the circle can be manually initialized around the nerve cell, and automatically moved to the boundary of the myelin. For segmenting a whole nerve, the curves are automatically propagated trough the volume, such that the surface moves only slightly between the slices. In every slice the surface is to be attracted to the boundary of the myelin layer.

Try segmenting nerves using Markov random fields and deformable models. In Figure 13.3, 13.4, 13.5 and 13.6 we show results of image analysis performed for the original study. However, those results are obtained on a full-resolution volumes, and using a combination of deformable models(Chapter 6) and layered surfaces(Chapter 7). Your results might therefore be of poorer quality.

For this open assignment, we provide some tips, but you are free to investigate other approaches.

- For MRF segmentation you might consider further downsizing the data or using only a subset of slices.
- When using MRF segmentation you can process the volume slice-by-slice. This corresponds to a situation where MRF-modelled smoothness prior has a parameter β for two neighbouring pixels in x and y direction, while the change of labels for neighbours in z direction

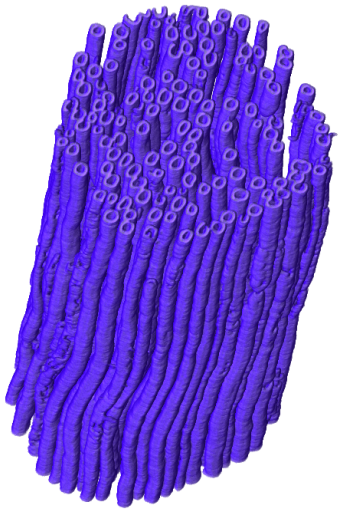

Figure 13.3: 3D visualization of segmented of myelinated nerves.

is not penalized. However, note that nerves are elongated in the direction orthogonal to image slices. For this reason, it would be more appropriate to set MRF-modelled smoothness especially high along the z direction, and this calls for a full 3D implementation of the MRF segmentation.

- When using deformable models note that assumption of Chan-Vese about a object of different intensity than the background does not apply for nerves. This is because a nerve consists of a dark myelin and a bright axon. Instead of allowing for the automatic estimation of the parameters m_{in} and m_{out} by averaging, it might be better to fix those parameters using the values estimated from the images. Alternatively, values m_{in} and m_{out} may be estimated from a thin band inside and outside the curve.

- The robustness of the deformable models might be improved by moving the curve towards the point where the change of intensity in the normal direction is high. This can be implemented by unwrapping the image (similar to exercise 1.1.5) following the curve normals. The gradient in normal direction can than be computed for the unwrapped image, and curve moved to the point where gradient is high.

- A node of Ranvier (see Figure 13.1 for schematic drawing of a nerve) can be seen on a few of the nerves in the volume to be analysed, as shown in Figure 13.6. Nodes are of a high interest for understanding nerve disorders. However, those might be challenging to capture due to the lack of myelin.

- Visualizing results in 3D usually provides a useful information on the segmentation results. Visualization options provided by MATLAB and python are good. Still, for large datasets, and advanced visualization you may want to use a specialized software, and we suggest trying ParaView. A few notes on 3D visualization using ParaView be found in ParaView notes.

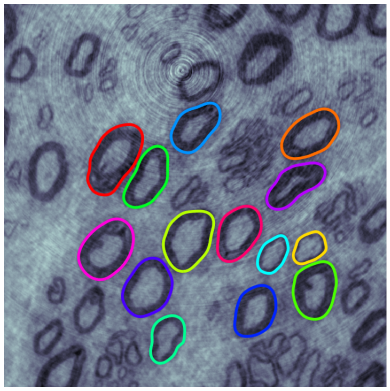

Figure 13.4: Axons segmented using deformable curves visualized on a single slice.

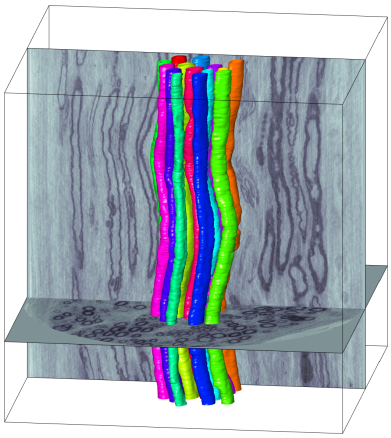

Figure 13.5: 3D visualization of axons segmented using deformable curves.

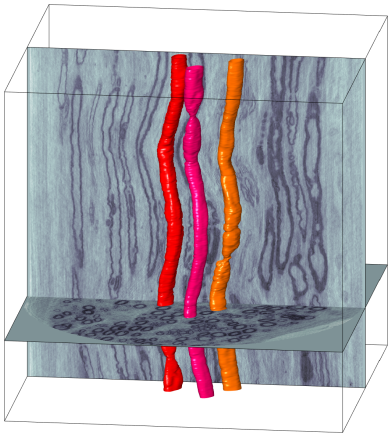

Figure 13.6: Three of the axons with visible node of Ranvier.

Tasks

1. Consider following microstructural measurements which can be extracted from volumetric data:

   Nerve density count: Number of axons per area of nerve-fibre cross-section. Measured in number per area.

   Myelin density: A fraction of nerve cross-section corresponding to myelin. Expressed as dimensionless fraction (a number between 0 and 1), or a percentage.

Average nerve area: Average area of nerves, broken down into average axon area, and average myelin area. This measure is very related to average nerve radius.

Average nerve radius: Average radius of nerves, broken down into average axon radius, and average myelin thickness. This measure is very related to average nerve area.

2. Perform a binary segmentation of the data using MRF. Which microstructural measurement can you extract from your MRF segmentation?

3. Perform a volumetric segmentation of few nerves using using deformable models. Which microstructural measurements can you extract from your segmentation?

14 Spectral segmentation and normalized cuts

Spectral clustering is an approach to data clustering problem, and it includes a number of related techniques. Spectral clustering is used in machine learning, computer vision and signal processing, with applications in processing speech spectrograms, DNA gene expression analysis, document retrieval and computation of Google page rank. The name *spectral* originates from the mathematical term *spectrum* (a set of the egenvalues of a given matrix), and this is because spectral clustering utilizes eigenvalues and eigenvectors of the data similarity matrix.

Spectral clustering is one of the fundamental data clustering approaches, it is easy to implement and solve by standard linear algebra software, there is no assumptions on the nature of the clusters, and the techniques have been mathematically rigorously proved. Disadvantages include high computational and memory requirements of the direct implementation. For this reason the practical use of spectral clustering often involves computational simplifications and significant pre- or postprecessing.

Spectral approach has gained a great popularity for image segmentation following the seminal paper on normalized cuts by Shi and Malik [1]. Recent uses of spectral approach is in the superpixel segmentation. Another intriguing example is the Copenhagen-based company Spektral (formerly known as CloudCutout) where spectral segmentation is a part of the successful green-screen removal product Figure 14.1.

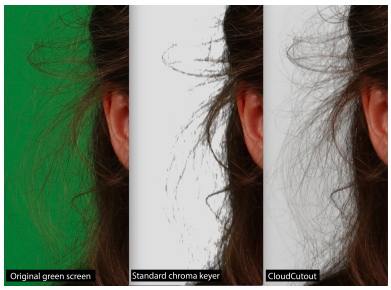

Figure 14.1: Spektral (formerly known as CloudCutout) green-screen product. [1]

Spectral methods can be applied to the data which is represented using a similarity matrix, and is often described in terms of graph partitioning. For a comprehensible coverage of (general) spectral clustering we recommend an excellent tutorial by von Luxburg [2]. We will here briefly cover spectral clustering, and will then turn to its use in image segmentation.

[2]

Boiled down to four words, the essence of spectral clustering is: *Eigensolution gives graph partitioning*. To be able to understand spectral

clustering, you need to be acquainted with the concept of graph cuts in order to describe the problem we want to solve. Furthermore, we need to represent a graph using an adjacency matrix and a closely related Laplacian matrix. Then we can see how eigensolution provides a solution to a graph cut problem.

14.1 *Graph cuts, graph representations and eigensolutions*

Recall that a graph consists of nodes and edges, and in general may be node-weighted, edge-weighed, directed or undirected. In context of spectral image segmentation, each pixel will correspond to a graph node, and pairs of pixels define graph edges – we will get back to image segmentation after covering the general case. For spectral clustering we work with edge-weighted undirected graphs which we represent using an adjacency matrix W with elements w_{ij} being the weight of the edge connecting the node i and a node j. Consider for example a graph in Figure 14.2 consisting of 12 nodes. This graph can be represented in terms of a 12×12 adjacency matrix, as illustrated in Figure 14.3.

A graph cut is a partitioning of a graph. The graph partitioning problems are concerned with finding a graph cut with the least cost. The simplest way of defining the cost of a cut is to consider all edges between the two partitions

$$\text{cut}(A, B) = \sum_{i \in A, j \in B} w_{ij} \,,$$

but that might lead to unbalanced cuts. For this reason we might prefer using some other measure of the cut cost, which also consider the size of the partitions. Commonly used are normalized cuts

$$\text{Rcut}(A, B) = \frac{\text{cut}(A, B)}{|A|} + \frac{\text{cut}(A, B)}{|B|} \,,$$

where we use the notation $|A|$ for a number of vertices in subset A, but one could also consider ratio cut and min-max cuts

$$\text{Ncut}(A, B) = \frac{\text{cut}(A, B)}{\text{vol}(A)} + \frac{\text{cut}(A, B)}{\text{vol}(B)} \,,$$

$$\text{MMcut}(A, B) = \frac{\text{cut}(A, B)}{\text{cut}(A, A)} + \frac{\text{cut}(A, B)}{\text{cut}(B, B)} \,,$$

where $\text{vol}(A)$ is weight of all edges associated with the subset, i.e. $\text{vol}(A) = \sum_{i \in A} d_i$ and $d_i = \sum_j w_{ij}$ is a degree of ith node. Figure 14.4 shows three graph cuts, while Figure 14.5 lists the costs associated with the three cuts given by different measures. You may confirm that the values are correct, and notice the different balancing properties of the cost measures.

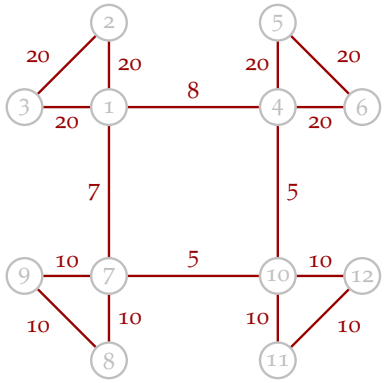

Figure 14.2: An example of an edge-weighted undirected graph.

	1	2	3	4	5	6	7	...
1	0	20	20	8	0	0	7	...
2	20	0	20	0	0	0	0	...
3	20	20	0	0	0	0	0	...
4	8	0	0	0	20	20	0	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

Figure 14.3: An adjacency matrix of a graph shown in Figure 14.2.

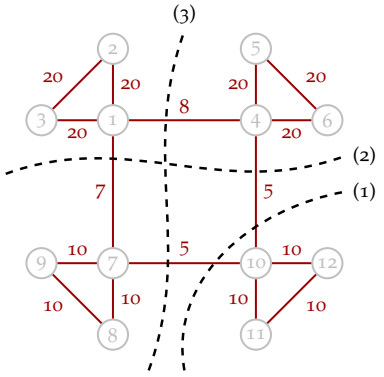

Figure 14.4: Three different partitioning of a graph.

	(1)	(2)	(3)
cut	10	12	13
Rcut	4.4̇	4.0	4.3̇
Ncut	0.1723	0.1293	0.1268
MMcut	0.3939	0.2784	0.2709

Figure 14.5: Values of the different cuts shown in Figure 14.4.

It turns out that the solution to the graph cut problem may be found by considering the eigensolution of the matrix closely related to the adjacency matrix – the graph Laplacian. Depending on the cost measure, the derivation will be slightly different, leading to the different (unnormalized and normalized) versions of the graph Laplacians. To normalize the Laplacian, we first define a diagonal degree matrix $\mathbf{D}$ with diagonal elements being node degrees $d_i = \sum_j w_{ij}$.

We define unnormalized graph Laplacian

$$\mathbf{L} = \mathbf{D} - \mathbf{W},$$

and two normalized graph Laplacians

$$\mathbf{L}_{\text{sym}} = \mathbf{D}^{-1/2}\mathbf{L}\mathbf{D}^{-1/2} = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{W}\mathbf{D}^{-1/2},$$

$$\mathbf{L}_{\text{rw}} = \mathbf{D}^{-1}\mathbf{L} = \mathbf{I} - \mathbf{D}^{-1}\mathbf{W}.$$

These matrices are closely related to each other, and so are their spectra. All three matrices have real-valued eigenvalues with 0 being the smallest eigenvalue, see tutorial by von Luxburg for a more complete list of properties.

For our purposes, the most important are eigenvectors of $\mathbf{L}_{\text{rw}}$, i.e. vectors satisfying $\mathbf{L}_{\text{rw}}\mathbf{u} = \lambda\mathbf{u}$. The smallest eigenvectors (corresponding to smallest eigenvalues) yield solution relaxed versions of finding the normalized cut. The relaxed solution is subsequently transformed into an approximate discrete solution to the original problem, for example using k-means clustering. In particular, the second smallest eigenvector gives the partitioning of the data in two subsets – the very smallest eigenvector (corresponding to 0) is constant.

It can be shown that eigenvectors of $\mathbf{L}_{\text{rw}}$ are generalized eigenvectors of $\mathbf{L}$ and $\mathbf{D}$, see againg von Luxburgs tutorial, Proposition (3) part 3. Therefore, to fint the solution to normalized cut, one may also seek solution to generalized eigenproblem $\mathbf{L}\mathbf{u} = \lambda\mathbf{D}\mathbf{u}$. This is the formulation of normalized spectral clustering according to the original paper by Shi and Malik. In practice, solving a standard eigenproblem requires less computation.

We can utilize other matrix algebra identities to make computation of the spectra more accurate and/or efficient. Many eigensolvers are more accurate when working of symmetric matrices. A strategy exploiting matrix symmetry would involve computing eigenvectors of the (symmetric) $\mathbf{L}_{\text{sym}}$, and transforming those to eigenvectors of $\mathbf{L}_{\text{rw}}$ by multiplying with $\mathbf{D}^{-1/2}$, see tutorial by von Luxburg, Proposition (3) part 2. Furthermore, if only a subset of eigenvectors is to be found, some eigensolvers are more efficient when finding eigenvectors corresponding to largest eigenvalues. This may be utilized by noticing that that smallest eigenvectors of $\mathbf{I} - \mathbf{A}$ are largest eigenvectors of $\mathbf{A}$.

14.2 *Clustering 2D points*

You should implement two functions. One function should take and affinity matrix and return eigenvectors corresponding to solution for normalized cuts. The second function should perform discretization of the eigenvectors using k-means.

You should first test your implementation on a small 2D point set. You can use points previously used for neural networks, but we also provide five point sets in a mat file `points_data.mat`. For each of the point sets you should:

- Visualize the point set, and identify the clusters (there are 2 clusters in set 3 and 5, and 3 clusters in set 1, 2 and 4).
- Construct the affinity matrix W. Use the fully connected graph and Gaussian similarity function (Luxburg, Section 2.2). You should initially estimate parameter σ so that it reflects the distance between the neighbouring points of the point cloud.
- Compute eigenvectors and clustering given by the normalized cut. Visualize the clustering.
- Determine ordering (permutation) of the points according to the clustering (so that points from the first cluster come first, followed by points in the second cluster, etc.)
- Visualize the values of the second eigenvector, first for unsorted points, then for sorted points.
- Visualize affinity matrix, and the affinity matrix for sorted points.
- Estimate the parameter σ which results in a meaningful clustering. You will need to change the parameter σ between point sets.

14.3 *Image segmentation*

Now we use spectral clustering on a pixels of a small image. Consider some of the provided test images. You might want to (drasticaly!) reduce the size of your images, to avoid memory problems.

You should:

- Construct the affinity matrix W using Equation (11) from article by Shi and Malik. Initially estimate parameters σ_I, σ_X and r. Instead of setting a radius r, for our small example you may use a fully connected graph (i.e. ignore if–otherwise condition of Equation (11) which sets affinity of distant points to 0).
- Visualize the spatial part of W, the brightness part of W and the final W.

- Compute eigenvectors and clustering given by the normalized cut.
- Visualize the values of the second eigenvector on the image grid.
- Visualize the segmentation results.
- Estimate the model parameters to obtain meaningful segmentation.

Start by the grayscale image. Use 2 clusters for plane and 5 clusters for vegetables. For the similarity (brightness, color) part of W treat an RGB value of each pixel as a vector to compute the Euclidian distance between the pair of pixels. Try also clustering the pixels using k-means clustering by treating RGB pixels values as vectors.

Consider adapting spectral methods to be able to handle larger images.

15 Probabilistic Chan-Vese

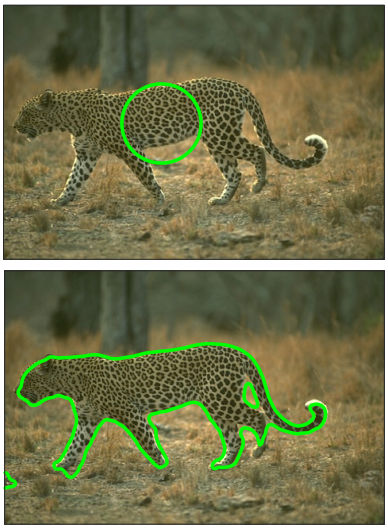

Figure 15.1: A deformable model used for segmenting forward-background image where two regions are characterized by different textures.
[1]

Chan-Vese segmentation algorithm alternates between two updates: update for mean intensities of the two regions given region boundaries, and update of the boundary between the two regions given mean intensities. The use of mean intensities implies that the two regions are distinguished by having different mean intensities. There are situations, where this is not the case, see Figure 15.1. Regions can be characterized by distributions of intensities or other features.

In this mini-project you will investigate generalizations of Chan-Vese algorithm, which allow for segmenting more challenging situations than with the original Chan-Vese. Some inspiration can be found in paper by Dahl and Dahl [1], which proposes a intensity-distribution approach Figure 15.2 and patch-distribution approach Figure 15.3. The approach is illustrated in Figure 15.2. Once the curve is initialized, instead of computing mean intensities for the inside and outside region, the distributions of intensities are collected for the inside and the outside region. For every pixel value we now have an information on how often it is in the inside and outside region, which can be translated into a probability that this pixel value is inside or outside. Computing such probabilities for all image pixels leads to probability image which can be used to deform the curve. Alternating, in a Chan-Vese manner, between computing probability image given the curve, and deforming the curve given probability image leads to segmentation.

For even more general case, instead of working with distributions of pixel intensities, distributions of image features may be used to compute the probability image. Figure 15.3 shows an example of using dictionary of image patches. For every patch from the dictionary we can compute the probability of it occurring in the inside or outside region.

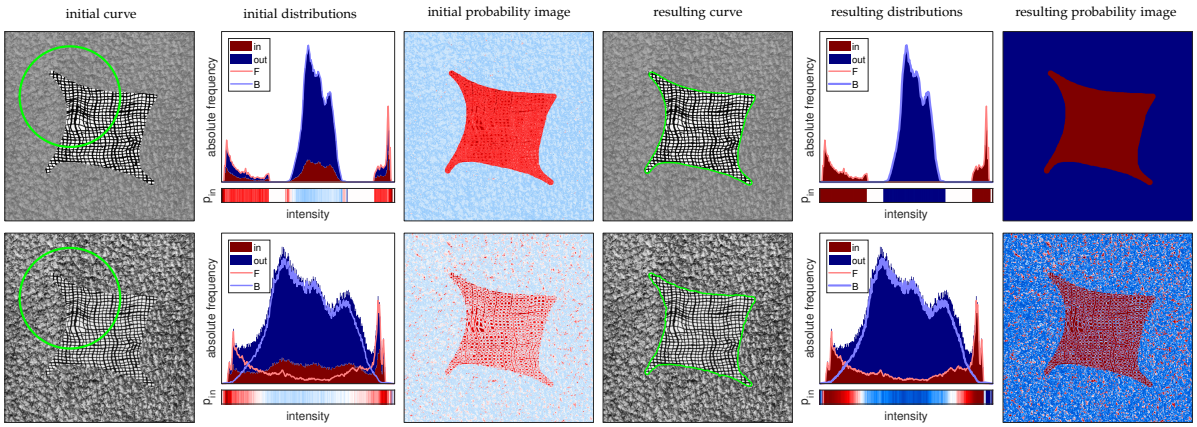

Figure 15.2: A probabilistic Chan-Vese approach when regions inside and outside are characterized by different distributions of pixel intensities. Top row shows easier problem of non-overlapping distributions, bottom row shows two overlapping distributions. Colums 1–3 show initialization, columsn 4–6 show result after iterating.

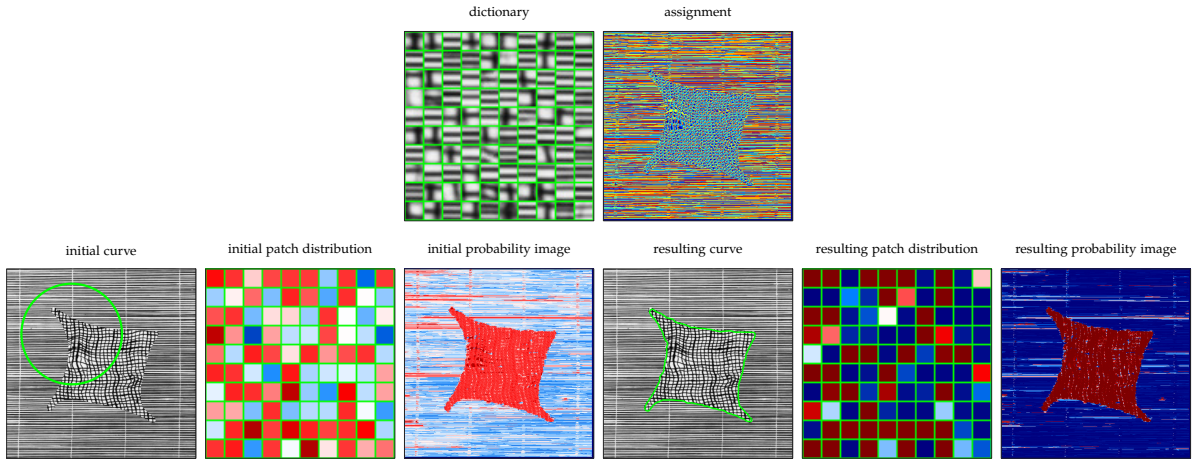

Figure 15.3: A probabilistic Chan-Vese approach when regions inside and outside are characterized by different texture. Top row shows used dictionary of image patches, and an assigmnent of image pixels to dictionary. In the second row, Images 1–3 show initialization and images 4–6 show result after iterating.

16 Learning snake deformation

Snakes, covered in 6, provide a very strong model for segmenting one simple object (foreground) from the background. The deformation forces used to align the snake with the boundary of the object may be defined to solve a special problem. The forces may also be learned, as attempted by a few recent research papers (see also Figure 16.1): Learning active contours, CVPR 2018; Fast Curve, CVPR 2019; Deep Snake, CVPR 2020; Learned Snakes, Signal Processing 2021.

In this project, we can attempt learning snake deformation using either the edge-based or the region-based approach. We could learn deformation from images with available ground-truth labels, for example cells or pets, or we could attempt an approach learning deformation without ground truth.

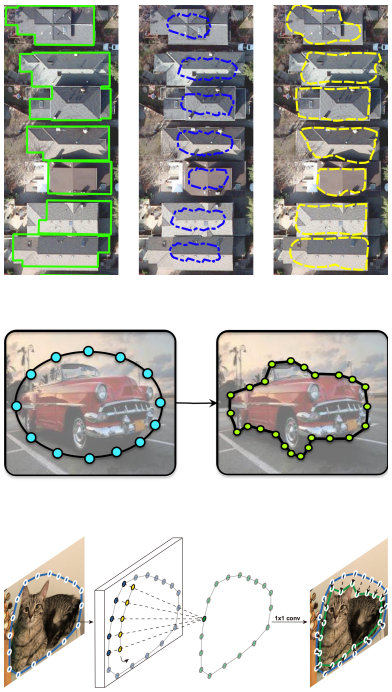

Figure 16.1: Approaches for learning snakes deformation, images taken from recent CVPR papers.

17 Orientation analysis

Analyzing orientations of images structures is often needed if we want to visualize, quantify, or elsewhere utilize orientation information obtainable from images. A common tool for orientation analysis is a structure tensor.

In this mini-project you will be working with structure tensor and orientation analysis. You may decide to focus on computation of structure tensor, visualization of the orientation information, quantification of orientation, or some similar aspect.

17.1 Computating structure tensor

In the context of volumetric (3D) image analysis, a structure tensor is a 3-by-3 matrix which summarizes orientation in a certain neighborhood around a certain point.

For example, consider volumetric data showing a bundle of roughly parallel fibers. If we extract two cubes from this volume, mutually displaced along the predominant fiber orientation, the two cubes will have very similar intensities. For two cubes displaced along other orientations the cube intensities would be more different. For this reason, measuring the change of intensities between slightly displaced cubes may be used for determining predominant orientation of imaged structures.

It turns out that, given an initial point and a size of the cube, squared change of intensities may be expressed as $\mathbf{u}^T \mathbf{S}\, \mathbf{u}$, where $\mathbf{u}$ is the direction of the displacement and $\mathbf{S}$ is 3-by-3 symmetric positive semi-definite matrix – a structure tensor. Finding predominant orientation now amounts to finding $\mathbf{u}$ which minimizes $\mathbf{u}^T \mathbf{S}\, \mathbf{u}$.

For a more formal derivation, consider a volumetric data V defined on the domain $\Omega \subset \mathbb{R}^3$, where $V(\mathbf{p})$ denotes voxel intensity at the point $\mathbf{p} = [p_x \quad p_y \quad p_z]^T$. Consider an arbitrary but fixed neighborhood N around a point $\mathbf{p}$, such that $N(\mathbf{p}) \subset \Omega$. We want to measure

$$D = \sum_{\mathbf{p}' \in N(\mathbf{p})} \left(V(\mathbf{p}' + \mathbf{u}) - V(\mathbf{p}') \right)^2 .$$

Assuming a small displacement we use first order Taylor expansion and arrive to

$$D = \sum_{\mathbf{p}' \in N(\mathbf{p})} \left(\begin{bmatrix} V_x(\mathbf{p}') & V_y(\mathbf{p}') & V_z(\mathbf{p}') \end{bmatrix} \mathbf{u} \right)^2$$

where we use notation $V_x = \frac{\partial V}{\partial x}$, and correspondingly for partial derivatives in y and z direction. Finally, exploiting commutativity of the inner product leads to

$$D = \mathbf{u}^T \sum_{\mathbf{p}' \in N(\mathbf{p})} \begin{bmatrix} V_x(\mathbf{p}') \\ V_y(\mathbf{p}') \\ V_z(\mathbf{p}') \end{bmatrix} \begin{bmatrix} V_x(\mathbf{p}') & V_y(\mathbf{p}') & V_z(\mathbf{p}') \end{bmatrix} \mathbf{u} \,.$$

So, as earlier claimed, we arrived to expression $D = \mathbf{u}^T \mathbf{S}\, \mathbf{u}$, where $\mathbf{S}$ is a 3-by-3 matrix – a structure tensor computed in a point $\mathbf{p}$ and using a neighborhood N. That $\mathbf{S}$ is symmetric and positive semi-definite follows directly from the construction of $\mathbf{S}$ (note that $D \geq 0$).

Using a compact notation for gradient $\nabla V = \begin{bmatrix} V_x & V_y & V_z \end{bmatrix}^T$, structure tensor is

$$\mathbf{S} = \sum \nabla V \, (\nabla V)^T \,.$$

Her we imply that structure tensor is computed for every voxel of the volume, i.e. structure tensor is a matrix-valued function over Ω. The summation is conducted over a neighborhood of each voxel, and result will be the same (up to the multiplicative factor) if summation is replaced by an averaging filter.

Two Gaussian filters are usually involved in computing structure tensor, see [1] for detailed description of 2D case. The one Gaussian filter has to do with averaging orientation information in the neighborhood. This can be achieved using a convolution with a Gaussian K_ρ, where parameter ρ, called *integration scale*, reflects the size of the neighborhood. Now we have

$$\mathbf{S} = K_\rho * \left(\nabla V \, (\nabla V)^T \right) .$$

The second Gaussian has to do with computing partial derivatives in gradient ∇V. To make differentiation less sensitive to noise we may convolve the volume with a Gaussian prior to computing derivatives. More efficiently, utilizing derivative theorem of convolution, partial derivatives can be computed by convolving with derivatives of Gaussian. We denote such gradient $\nabla_\sigma V$. The parameter σ is called *noise scale*. Expression with both Gaussians is

$$\mathbf{S} = K_\rho * \left(\nabla V_\sigma \, (\nabla V_\sigma)^T \right) .$$

In summary, computing structure tensor for each voxel of a volume V involves three steps:

1. Convolve V with derivatives of Gaussian with standard deviation σ to obtain V_x, V_y and V_z. For efficiency, use separability of Gaussian kernel.

2. Using element-wise multiplication compute six volumes V_x^2, V_y^2, V_z^2, V_xV_y, V_xV_z and V_yV_z.

3. Convolve each of the six volumes with the Gaussian kernel with standard deviation ρ. For efficiency, use separability of Gaussian kernel. The resulting volumes contain per-voxel elements s_{xx}, s_{yy}, s_{zz}, s_{xy}, s_{xz} and s_{yz} of the structure tensor

$$\mathbf{S} = \begin{bmatrix} s_{xx} & s_{xy} & s_{xz} \\ s_{xy} & s_{yy} & s_{yz} \\ s_{xz} & s_{yz} & s_{zz} \end{bmatrix} .$$

17.2 *Computing orientations*

Given structure tensor $\mathbf{S}$, predominant orientation is found by minimizing Rayleigh coefficient $\mathbf{u}^T\mathbf{S}\,\mathbf{u}$ trough eigendecomposition of $\mathbf{S}$. Being symmetric and positive semi-definite $\mathbf{S}$ yields three positive eigenvalues $\lambda_1 \leq \lambda_2 \leq \lambda_3$ and mutually orthogonal eigenvectors $\mathbf{v}_1$, $\mathbf{v}_2$ and $\mathbf{v}_3$. The eigenvector $\mathbf{v}_1$ corresponding to the smallest eigenvalue is an orientation leading to the smallest variation in intensities, which indicates a predominant orientation in the volume. Note that $\mathbf{v}_1$ is an orientation, and we usually represent it using an unit vector, but this is still not an unique representation since there are two opposite unit vectors sharing the orientation with $\mathbf{v}_1$.

Eigendecomposition of a 3-by-3 real symmetric matrix can be computed efficiently using an analytic approach by Smith [2] which uses an affine transformation and a trigonometric solution of a third order polynomial.

If there is no strong orientation in the volume, all eigenvalues will be similar, and dominant direction will be influenced by small local variations or the noise in the data. For this reason it is customary to analyze the ratio between eigenvalues to determine the degree of anisotropy in the neighborhood, and how (locally) line-like or plane-like the imaged structure is, see illustration 17.1. Inspired by diffusion tensor processing [3], we define values, so-called shape measures,

$$c_l = \frac{\lambda_2 - \lambda_1}{\lambda_3} , \quad c_p = \frac{\lambda_3 - \lambda_2}{\lambda_3} , \quad c_s = \frac{\lambda_1}{\lambda_3}$$

where c_l gives a measure of linearity, while c_p and c_s measure planarity and sphericity. Shape values are positive and sum to 1.

In summary, structure tensor is a 3-by-3 matrix that can be computed in each volume voxel. The computation of structure tensor requires

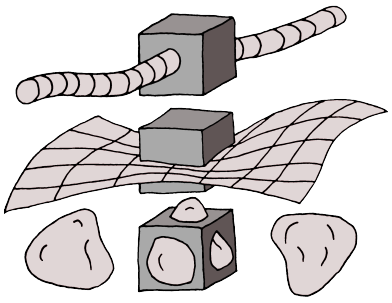

Figure 17.1: Neighborhoods and structures corresponding to linear, planar and spherical shape. On a linear structure *(top)* cubical neighborhood can move along the predominant direction leading to small change in intensities, while other two orthogonal directions lead to significantly larger and approximately equal change. On a planar structure *(middle)* two directions lead to small and approximately equal change in intensities, while third direction leads to significantly larger change. For a case with no predominant direction *(bottom)* the three orthogonal directions lead to roughly equal changes in intensities.

[2]

[3]

two parameters: noise scale σ and integration scale ρ. Being symmetric, structure tensor can be represented with 6 scalar values. The most important information extracted from structure tensor is a dominant orientation. Dominant orientation is a unit vector with equivalence relation $-\mathbf{v} \equiv \mathbf{v}$. Shape measures, also extracted from structure tensor, may also be of interest. Shape measures are three scalar ,c_l, c_p and c_s, summing to 1.

17.3 *Visualization*

Having extracted structure tensor and performed its eigendecomposition, following values are available for every volume voxel.

- Voxel intensity V, a scalar value in a certain range.
- Shape measures c_l, c_p and c_s, three scalar values summing to 1.
- Dominant orientation $\mathbf{v_1}$, an orientation vector (unit vector with equivalence relation $-\mathbf{v} \equiv \mathbf{v}$).
- Other information, such as $\mathbf{v_2}$, $\mathbf{v_3}$, and the values λ_1, λ_2 and λ_3 is also available, but usually of no special interest.

Visualizing this information often requires some care.

Shape measures, being three scalar values can be conveniently represented using three RGB color channels. Voxels with large c_l will appear red, large c_p will be green, and large c_s blue. This is the approach used in Figure 17.2.

Predominant orientation, is a 3D unit vector with equivalence relation $-\mathbf{v} \equiv \mathbf{v}$. A common way of visualizing orientation is to use absolute values of vector coordinates, i.e. $|v_x|$, $|v_y|$, and $|v_z|$, as three RGB color channels. Orientations roughly aligned with x direction will be red, those aligned with y green, and z blue. This has a desirable property that$-\mathbf{v}$ and $\mathbf{v}$ map to the same color. Furthermore, when the imaged object has a certain geometry aligned with the coordinate system (for example, elongated object aligned with z axis), this coloring scheme may be favorable for the interpretation of orientations. Undesirable property of the RBG color scheme is that different orientations map to the same color, for example four orientations corresponding to main diagonals in unit cube all map to gray.

Shape measure and predominant orientation are computed for every voxel in the volume – regardless of whether the voxel is within the object or material which we investigate. When using volume rendering to visualize the extracted measures, if the value is shown in every voxel, the valuable information might be occluded. For this reason it might be beneficiary to combine visualization of extracted measures

with the intensity values, which carry information on voxels containing, or not containing, material. A visually pleasing result is obtained if voxels containing no material are shown transparent. Furthermore, predominant orientation is only relevant for voxels exhibiting high linearity, so shape measure may be combined with visualization of predominant orientation. This is the approach used in Figure 17.2.

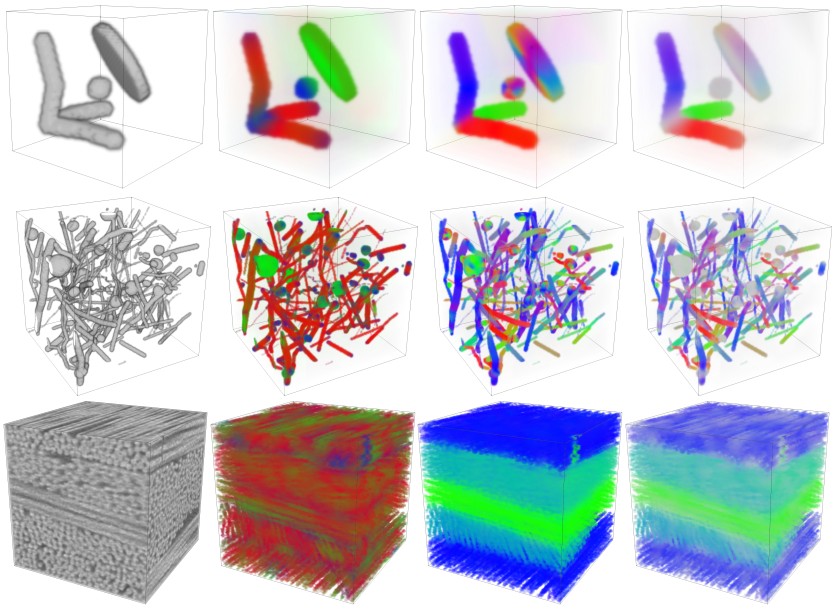

Figure 17.2: Orientation information. In each row, from left to right: 3D rendering of the volumetric data, shape measures visualized in colors, predominant orientation in the material phase, visualized in colors, predominant orientation weighed by the measure of linearity. Top two rows show the result on synthetic data, while in the bottom row we see the result of orientation analysis for composite material.

17.4 *Applications*

Some uses of orientation information are:

- Volumetric visualization of the shape measures and/or the predominant orientation.
- Producing histograms of orientations by binning orientation vectors. These distributions live on a half sphere, which can complicate the visualization, comparison, and fitting of distributions.
- Comparing orientation information extracted form volumetric data with, for example, the orientations obtained via modeling and/or simulation.
- Using local orientation to segment the volume into regions of constant orientation.
- Using local orientation to tune smoothness constraint in MRF segmentation.
- Using local orientation to guide fibre tracking.

18 CNN for segmentation

Image segmentation is often needed as an intermediate step when quantifying structures in images, and we have previously been working with patch-based segmentation. In this exercise you should train a convolutional neural network to segment electron microscopy images of neuronal structures. This data was part of the ISBI Challenge: *Segmentation of neuronal structures in EM stacks*[1], and can be found in the file `EM_ISBI_Challenge.zip`.

The data is from a serial section Transmission Electron Microscope and depicts the ventral nerve of a Drosophila larva. An example image is shown in Figure 18.1. The task is to segment the membranes between cells. These appear mostly darker than the rest of the image, but they are often thin and smeared out, and there are other dark regions that are not membranes, but structures such as mitochondria, that should be part of the cell class. Therefore, this segmentation problem is difficult and requires either a biologist that knows the anatomy or an advanced automated image segmentation method. You should aim at building the automated segmentation method.

The data consists of 30 images with associated labels and additional 30 test images without labels. All images are 512×512 pixels. The task is to build and train a neural network that can segment this type of images. Normally, you would split your data into a training, validation and test set. You would use the training set for learning the model parameters and the validation set to ensure generalization of the model, e.g. that it does not overfit to the training set. In an ideal performance assessment, you will only use the test set once to measure the actual performance of your method. When the same test set is used multiple times, you would optimize for precisely that test set, which will bias your performance assessment.

The problem here is that you only have 30 images with labels to train, validate, and test your algorithm, if it should be done using quantitative measures. The test set does not come with the ground truth labels, since it was kept for evaluating performance of algorithms at the ISBI Challenge, and therefore the images in the test set only allows you to

[1] Ignacio Arganda-Carreras, Srinivas C Turaga, Daniel R Berger, Dan Cireşan, Alessandro Giusti, Luca M Gambardella, Jürgen Schmidhuber, Dmitry Laptev, Sarvesh Dwivedi, Joachim M Buhmann, et al. Crowdsourcing the creation of image segmentation algorithms for connectomics. *Frontiers in neuroanatomy*, 9:142, 2015

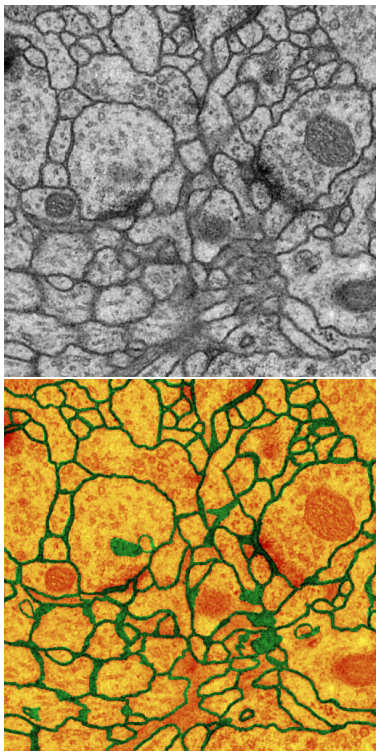

Figure 18.1: Example of EM data for segmentation. Top image is the original EM slice and the bottom image is the labels overlaied.

do a qualitative evaluation. It is your task to choose the images for training, validation, and test, and you should argue for your choice.

You can find inspiration for building your segmentation method in other peoples work. The ISBI challenge data set has e.g. been used in the U-net paper[2], which is a very successful model for segmentation based on deep learning. This network uses four down-sampling steps followed by four up-sampling steps, and at each resolution there more convolutions and activation steps. In addition there are so-called skip connections that connects layers at different scales. This architecture can be drawn in an u-shape, which has given the name to the method. Furthermore, the paper describes data augmentation suited for this type of data, so you might use the paper to get inspiration for your solution.

[2]

18.1 *Suggestions*

You are suggested to implement your model using either the Deep Learning Toolbox from MATLAB or Pytorch or similar for Python. Furthermore, to avoid spending too much time in the initial training, it can be a good idea to start with smaller images than the 512×512 pixels. You can e.g. split the images into smaller patches and work from these. Then you can at a later stage, when you are sure that your model is working as expected, increase the size of your images.

It is also a good idea to start with a simple model, where it is easy to ensure that all steps are working as they should. Then you can gradually increase the complexity.

Besides training the deep learning model and setting it up such that it works as expected, there are many choices to be made for building your model and for optimizing it. You are welcome to try out different approaches and investigate the effect of e.g. data augmentation. It is a very good idea to be systematic and show the effect of different parameter choices – either using quantitative and qualitative evaluation measures.

19 Superresolution from line scans

Scanning along a set of parallel lines is a common setting in medical imaging. For example, consider optical coherence tomography (OCT), well established in ophthalmology for obtaining images of the retina. Using OCT, the retina is scanned in along a line with a high transversal resolution. Collecting a number of scans along parallel lines, a larger area of the retina may be covered. Since scanning speed of the OCT systems employed in clinic is limited, and prolonged scanning is unpleasant for the patient, the distance between the parallel lines is often large compared to the transverse resolution of the scans. Therefore, the resolution of the scan is much coarser in one direction. In other words, each pixel covers a non-square area. Elongated pixels appear as stripes and influence the visual appearance of the image. The stripes can disturb the interpretation of the image and make it difficult to distinguish the anatomical structures, especially evident with blood vessels running parallel to scan lines.

To reveal additional anatomical structures, another OCT scan may be performed, along the lines orthogonal to the first scan, as a pari of images shown in Figure 19.1. Several problems emerge in connection to this. The eye might move during scanning, and the intensity might vary significantly between the scans. And most importantly, how to combine two scans covering the same area, one with high resolution in x direction, and the other in y, such that the resulting image has a satisfactory quality? To simplify the problem, we consider a set up as in Figure 19.2, where pixels of unknown values are to be estimated from the pixels of known values.

This problem is very similar to single-image superresolution, which can be obtained with great quality using neural networks. In this mini-project, you can try using neural network for upsampling line scans. The project involves setting up a framework based on the publicly available frameworks. We would expect the performance to improve significantly if prior knowledge about the appearance of the images is incorporated in the method. The training should therefore be performed on images having similar appearance as OCT image, but

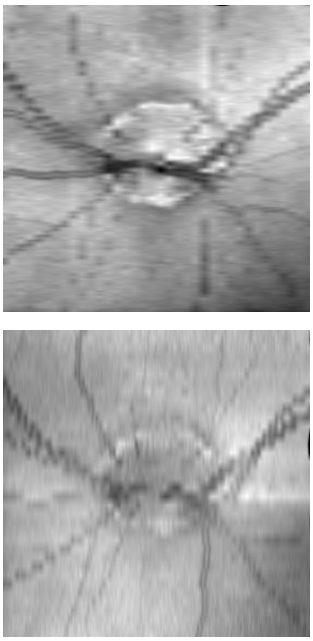

Figure 19.1: Example of images obtained using line scans in orthogonal directions. Notice that vertical lines are less clear in the first image, while horizontal lines are unclear in the second.

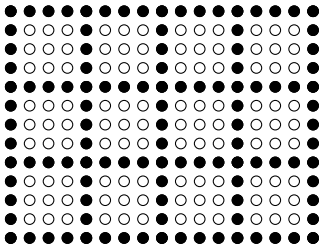

Figure 19.2: An image with pixels divided in pixels with known intensity (black), and pixels with unknown intensity (white), here shown with an upsampling rate of 4.

that can be images of vasculature obtained using other modalities. We will provide you such images.

To evaluate the performance of the upsampling, the conventional approach is to use peak signal-to-noise ratio (PSNR) metric. Furthermore, a comparison with the simple base-line upsampling scheme would be nice. For example, consider a schme where each direction is linearly upsampled, and the two contributions are combined as an average. Such an approach is show in Figure 19.3.

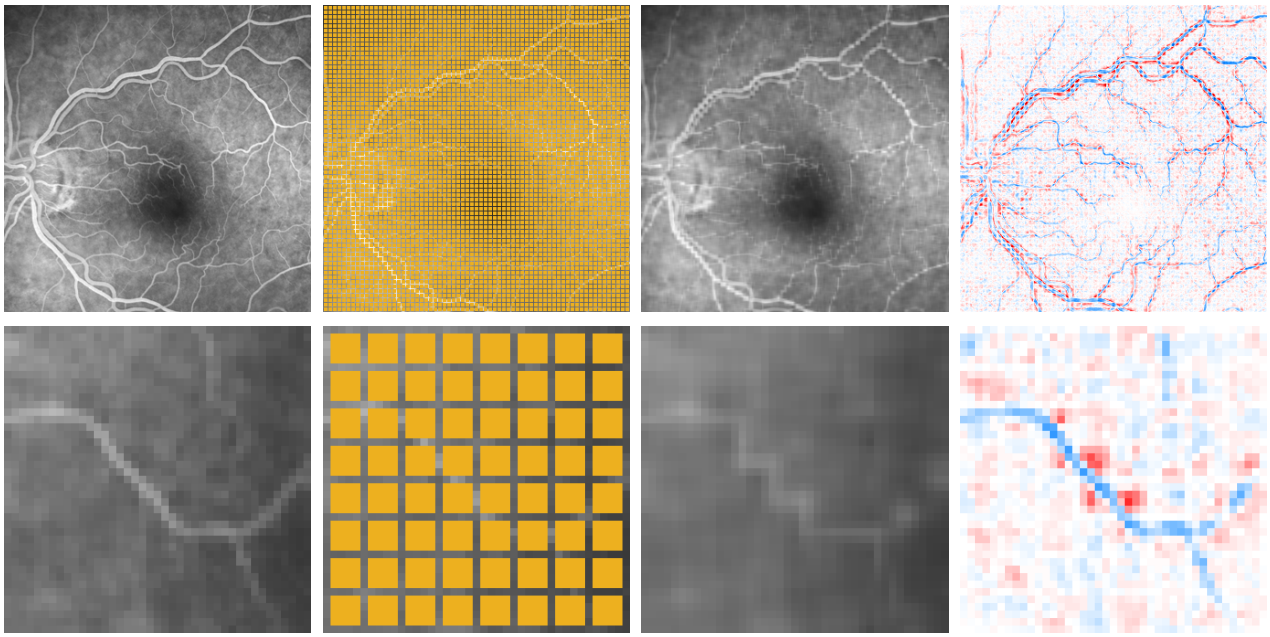

Figure 19.3: A simple linear upsampling scheme. First column: ground truth; second column: unknown pixels masked; third column: upsampling result; fourth column: the error, with color indicating the sign. Top row: image of vasculature; bottom row: zoom in on a small part of the image.