4 Feature-based segmentation

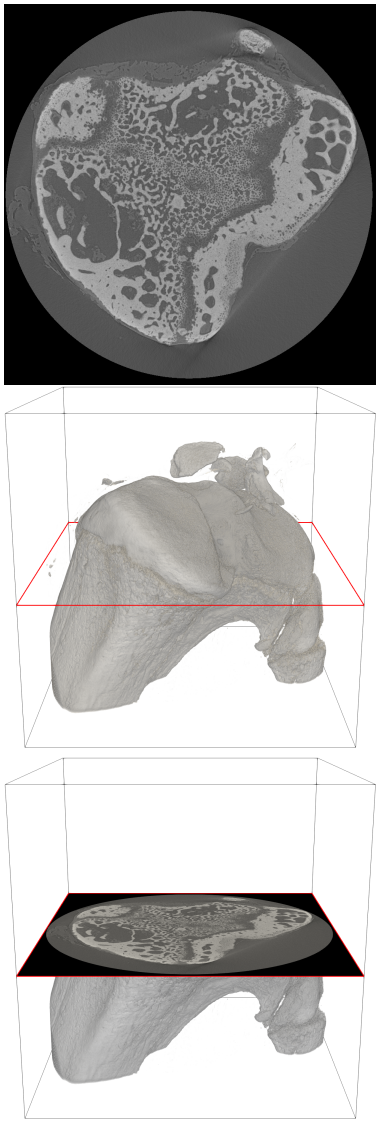

Figure 4.1: CT-scan showing the trabecular structure of a mouse bone. Top is the image slice, middle shows the bone in 3D and bottom shows where the slice is taken in the image volume.

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 4.1 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 4.2 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

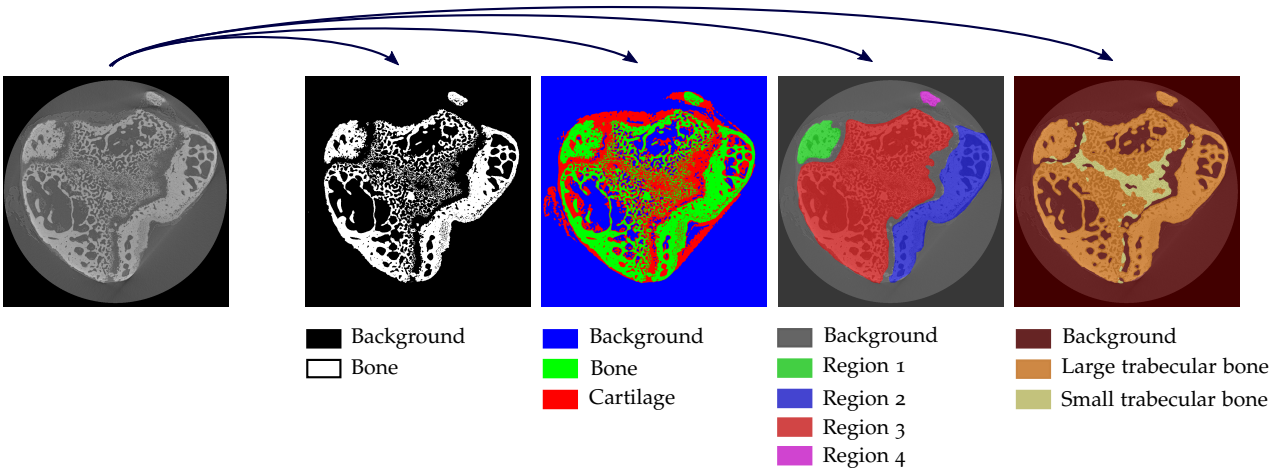

Figure 4.2: Slice of a CT-scan of a trabecular mouse bone that has been segmented based on three different criteria. The left image is the original. The next image in black and white is a segmentation into bone and background. The third image in red, green, and blue is a segmentation into bone, cartilage, 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 4.2, 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.

4.1 *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

4.1.1 *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.

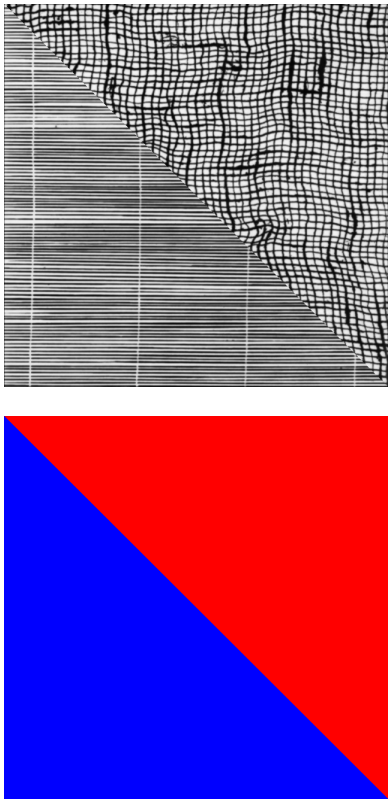

Figure 4.3: A training image with ground truth labeling. Red is one label and blue is another. Typically, this will be two scalar values, e.g. $\{0, 1\}$.

4.1.2 *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 4.3 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

$$p_C(\lambda) = \frac{\text{\# elements from } C \text{ with label } \lambda}{\text{\# elements in } C} \,, \tag{4.1}$$

which can be written as

$$p_C(\lambda) = \frac{1}{|C|} \sum_{f \in C} \delta\left(\ell(f) - \lambda\right) \,, \tag{4.2}$$

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

$$\delta(x) = \begin{cases} 1 & \text{if} \quad x = 0 \\ 0 & \text{otherwise} \end{cases} \,. \tag{4.3}$$

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.

4.2 *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

4.2.1 *(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 `local_features.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.

4.2.2 *(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.

4.2.3 *(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. 4.2 and 4.3.

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. 4.2 and 4.3, and store the probabilities in the array.

4.2.4 *(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 $r \times c$.

4.2.5 *(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.

4.2.6 *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.