Lecture Notes in Image Analysis

Olof Enqvist

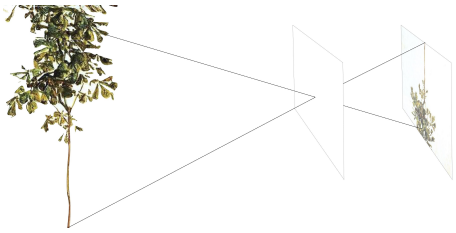
Contents

Introduction

What is an image?

Light passing through a small hole into a dark room projects an image of the outside world on the wall of the room; see Figure 1a. This room is the veiled chamber, the *camera obscura*, that has given our modern cameras their name. A pinhole camera uses the same principle to produce a photograph. A sensor (or photographic plate) is placed inside a small box and a tiny *pinhole* is made in the wall of the box. Figure 1b shows the principle. Light passes through the pinhole and projects an image of the tree on the rear wall of the camera.

(a) A camera obscura.



(b) Principle of a pinhole camera.

In an ideal pinhole camera, the pinhole is a single point, and the picture becomes a perfect depiction of the scene. In practice, we are faced with the following dilemma: A larger hole lets in more light and creates better contrast in low light conditions, but the images get blurry. To handle this, one uses a lens to collect the light. Unfortunately it is not possible to make a lens that works for objects at any distance, so a lens camera will only provide sharp images at a certain distance.

Somewhat unexpectedly, the pinhole camera model is very useful even for cameras with complex lens systems. More on this later.

A digital image. The image sensor in a digital camera consists of a rectangular grid of light-sensitive elements, so called pixels. Each pixel measures the amount of light hitting that particular part of the sensor. Hence a grayscale digital image is an array of non-negative intensity values, where high values

correspond to bright regions and low values to darker regions; see Figure 2. Apart from characteristics of the photographed object, the intensity value of a certain pixel also depends on the shutter time of the camera, the amount of lighting in the scene and the sensitivity of the light sensor. Hence the absolute intensity is rarely useful for recognition purposes and we will constantly rescale images to place focus on relative intensity differences instead.

$$\begin{bmatrix} 0 & 64 & 106 & 128 & 128 & 106 & 64 & 0 \\ 64 & 128 & 170 & 191 & 191 & 170 & 128 & 64 \\ 106 & 170 & 212 & 234 & 234 & 212 & 170 & 106 \\ 128 & 191 & 234 & 255 & 255 & 234 & 191 & 128 \\ 128 & 191 & 234 & 255 & 255 & 234 & 191 & 128 \\ 106 & 170 & 212 & 234 & 234 & 212 & 170 & 106 \\ 64 & 128 & 170 & 191 & 191 & 170 & 128 & 64 \\ 0 & 64 & 106 & 128 & 128 & 106 & 64 & 0 \end{bmatrix}$$

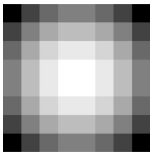


Figure 2: A digital image is simply an array of intensity measurements, often represented by 8-bit integers.

Colour. The first colour photographs were produced by the physicist James Clerk Maxwell in 1876. He took three photos using filters of coloured glass and then superposed the three images using a projector similar to the one in Figure 3. In many ways, the techniques used today to produce a colour photograph are surprisingly similar. A digital image still consists of three grayscale images containing the red, green and blue channels and this information is normally obtained by filtering the light using different types of filters.

Due to varying lighting conditions and camera characteristics, we should not expect, say a red apple, to always yield a similar relationship between the three colour channels.

Figure 3: Colour photographs taken by Sergey Prokudin-Gorsky in 1911. The upper row shows three grayscale images captured through filters of coloured glass. Using a projector similar to the one shown on the bottom left he could produce a colour image.

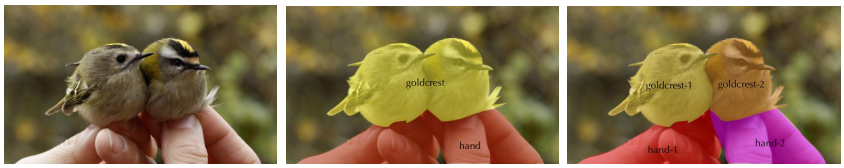
Other images. Apart from standard cameras, images are also produced by x-ray machines and scanners. Moreover three-dimensional images are produced with techniques such as computed tomography, positron emission tomography and magnetic resonance imaging; see Figure 4. Most of the techniques discussed in this course are applicable to these image types as well as the standard photos that you get from a cell phone or a digital camera.

Figure 4: A computed tomography (CT) image is a 3d array of pixels (sometimes called voxels). Each pixel corresponds to a rectangular block in 3D space and measures how well the tissue in this block blocks x-rays.

What is image analysis?

Recognition

The great question of image analysis is how to recognize objects in images. Different applications might require different levels of accuracy. Sometimes it is sufficient to tell whether or not there is a bird in the image. If you want a rough position for each bird in the image, we are dealing with bird *detection* and if you want an accurate delineation of the birds you are doing bird *segmentation*; see Figure 5.

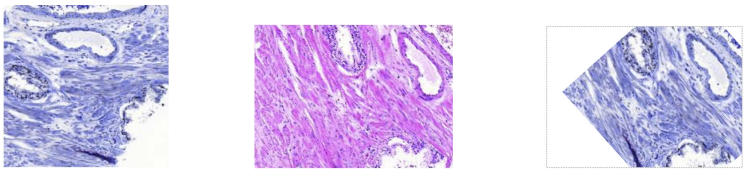


(a) (b) (c)

Figure 5: (a) Input image. (b) In segmentation we want to label each pixel with respect to a set of labels. (c) In instance segmentation we also want to separate different instances of the same object type.

Geometric models

A slightly different branch of image analysis is concerned with estimating accurate geometric models from images. Figure 6 shows an instance of *image registration*. The task here is to estimate an rigid transformation that aligns the source image with the target image. Similar methods can be applied to *3d reconstruction*, i.e., the process of extracting three-dimensional information from images. Figure 7 shows two images where a set of corresponding points have been matched across the images. From these point correspondences it is possible to estimate the motion of the camera. This problem is called *relative pose estimation* and will be discussed in Chapter 10.



(a) source image (b) target image (c) resulting warped image

Figure 6: The task in image registration is to estimate (and apply) a transformation that aligns the source image with the target image. The result is a warped image (c) of the same size and alignment as the target.

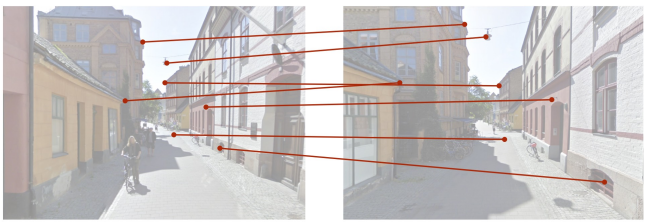


Figure 7: Using point correspondences across images, it is possible to extract geometric information such as the motion of the camera.

Part I

Recognition

Chapter 1

Filtering and linear classifiers

Arguably, the main objective of this course is to learn how to classify images. We want to design a classifier that takes an image and outputs a set of appropriate labels. The goal of this chapter is to introduce the linear classifier. While it is not a very powerful classifier for real-world problems, it will serve as a simple, and very useful example for many topics in this course.

1.1 Images as vectors

A digital image is essentially an array of intensity values. This makes it very similar to a vector, with the only difference that we write the numbers in a rectangular array rather than in a single column. Hence it is natural to borrow the standard vector operations and check whether they can be useful for images as well. Having images with the same number of elements, we can add or subtract them just as we add and subtract vectors. We can multiply with a constant or compute dot products. For example,

$$I - J = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 2 & 0 \\ 2 & 0 & 1 \end{bmatrix} - \begin{bmatrix} 0 & 0 & 2 \\ 1 & 0 & 0 \\ 2 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.1}$$

and the dot product is

$$I \cdot J = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 2 & 0 \\ 1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 2 \\ 1 & 0 & 0 \\ 3 & 0 & 0 \end{bmatrix} = 2 \cdot 2 + 1 \cdot 3 = 7 \tag{1.2}$$

Like with vectors, we allow multiplication with a number,

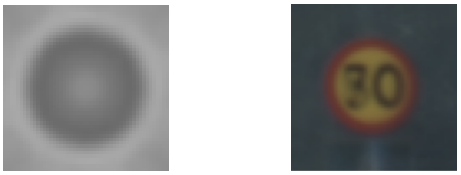
$$2\,I = 2 \begin{bmatrix} 1 & 0 & 2 \\ 0 & 2 & 0 \\ 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 4 \\ 0 & 4 & 0 \\ 2 & 0 & 2 \end{bmatrix} \tag{1.3}$$

A colour image can also be treated as a vector, but with three times as many elements. For colour images I and J, the dot product is

$$I \cdot J = I_R \cdot J_R + I_G \cdot J_G + I_B \cdot J_B, \tag{1.4}$$

where the subscripts indicate the R, G and B channels of the colour images.

Figure 1.1 shows two examples of averaging images, indicating that at least in special cases this is a meaningful operation. Apart from these operations, the vector interpretation gives us concepts such as subspaces, linear bases and principal component analysis, all of which can be useful for images as well.



(a) (b)

Figure 1.1: (a) Average over 100 cell images. (b) Average speed limit sign.

To refer to a pixel in the third column and the second row of an image, we will use the notation $I(3,2)$,[1]. For images, the upper left pixel will have coordinates $(1,1)$,

$$\begin{bmatrix} I(1,1) & I(2,1) & I(3,1) & \dots \\ I(1,2) & I(2,2) & I(3,2) & \dots \\ I(1,3) & I(2,3) & I(3,3) & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}. \tag{1.5}$$

This makes it possible to give a more precise definition of the dot product of two $M \times N$-images

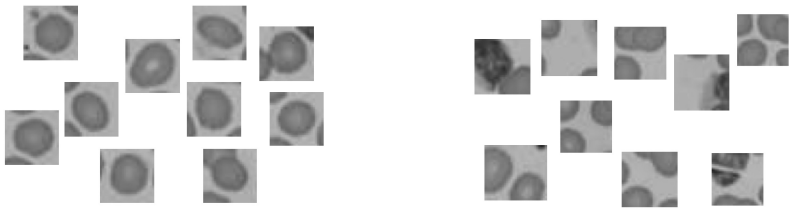
$$I \cdot J = \sum_{x=1}^{M} \sum_{y=1}^{N} I(x,y)\, J(x,y). \tag{1.6}$$

1.2 Linear classifiers

Consider the following set of image patches (extracted from larger microscopy images). We want to be able to differentiate between the cell patches on the left and the background patches on the right. We need a classifier, i.e., a function that takes an image and assigns to it one of the following classes

$$\left\{ \text{background, centred cell} \right\} \tag{1.7}$$

[1]Note that in Matlab, rows and columns are reversed when accessing the actual pixel value.



(a) Foreground patches. (b) Background patches.

Figure 1.2: Two types of image patches.

Recall that an image can be treated as a high-dimensional vector. One of the things that we can do with vectors is to compute dot products, so let us see if we can use this operation to design a classifier. We would like to find another vector/image w such that $I \cdot w$ is large for a cell image, I, and small for any other images. Using such a w together with a threshold τ, we can classify images as *centred cell* if $I \cdot w > \tau$ and *background* otherwise.

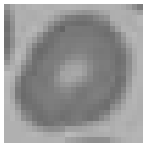


Figure 1.3: Template for cell detection.

For a cell, we want w to have low values in the middle and larger pixel values around. A simple way to obtain such a w is using an example from the class in question, a so called template. Let T denote a template patch such as the one shown in Figure 1.3. Setting $w = T$ does not work very well: As T consists solely of positive values the scalar product with a white patch will be very high. To avoid this we require

$$0 = \mathbb{1} \cdot w = \sum_x \sum_y w(x, y), \tag{1.8}$$

where $\mathbb{1}$ corresponds to an image of only ones. One way to achieve is by subtracting the average pixel intensity, setting

$$w = \frac{1}{N_{el}} \left(T - \mu_T \, \mathbb{1} \right), \quad \text{where} \quad \mu_T = \frac{1}{N_{el}} \sum_x \sum_y T(x, y) = \frac{1}{N_{el}} T \cdot \mathbb{1}, \tag{1.9}$$

and N_{el} is the number of elements in T, that is the number of pixels times the number of colour channels. (Dividing the whole expression with N_{el} does not

change the final classifier, but it is a good habit as it makes it makes the result independent of the number of pixels.)

As an example, assume we are given a template

$$T = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 5 & 4 \\ 0 & 4 & 5 \end{bmatrix} \tag{1.10}$$

To turn this template into a good linear classifier, we simply subtract the mean and divide by the number of elements (being 9),

$$\mu_T = (5 + 4 + 4 + 5)/9 = 2, \tag{1.11}$$

yielding

$$w = \frac{1}{N_{el}}(T - \mu_T \mathbb{1}) = \frac{1}{9}(T - 2\,\mathbb{1}) = \frac{1}{9}\begin{bmatrix} -2 & -2 & -2 \\ -2 & 3 & 2 \\ -2 & 2 & 3 \end{bmatrix}. \tag{1.12}$$

Finally, given a weight image, w, we determine an appropriate threshold by considering $w \cdot I$ for a set of positive and negative example patches. The goal, of course, is to find a threshold that yields as few misclassifications as possible.

Remark. When working with colour images, the weight matrix, w, should also have three different colour channels. This allows the classifier to use the colour information when determining the output.

1.3 Linear filtering

Let us say we have a blood cell classifier for 25×25-images and also a microscopic image of size 1000×1000 that is likely to contain thousands of blood cells. We want to use our classifier to detect all these cells.

First recall that our classifier consists of a weight image, w, and a threshold, τ. In an attempt to detect all cells in the larger image, we will apply the dot product with w in a sliding-window manner. We start by taking the dot product between w and the 25×25 patch in the upper left corner of the image. Then we move the patch one pixel to the right and repeat; see Figure 1.4. In the end we applied the classifier to all possible patches in the image. At least roughly, we get 1000×1000 output values. It is convenient to arrange store them in a 1000×1000 image similar to the input image; see Figure 1.4.

Let us look closer at the sliding window dot product. The sliding window dot product is more known as linear filtering and the weight image w is often called a filter. In signal processing filtering is often called cross-correlation, but we will avoid this name as it conflicts with the statistical definition of correlation. Also note that filtering is closely linked to convolution, but without flipping the filter.

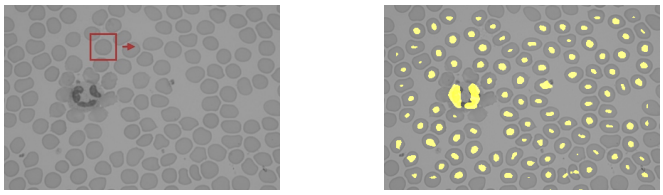


Figure 1.4: In a sliding window detector, a classifier is applied to every patch in a larger image. On the right the pixels that were classified as *cell* are overlaid in yellow.

Since filtering will be used extensively in this course (and elsewhere) we give it a convenient notation, $I \star f$ for an image I and a filter f. A filter is simply an image with a defined *anchor* pixel. For odd-sized filters, this will almost always be at the centre of the image:

$$\begin{bmatrix} f(-1,-1) & f(0,-1) & f(1,-1) \\ f(-1,0) & f(0,0) & f(1,0) \\ f(-1,1) & f(0,1) & f(1,1) \end{bmatrix} \tag{1.13}$$

Using this convention we can write the filtering of an image I with a filter f as

$$(I \star f)(x,y) = \sum_{i=-m}^{m} \sum_{j=-n}^{n} I(x+i, y+j) f(i,j), \tag{1.14}$$

assuming that the filter has $2m+1$ rows and $2n+1$ columns.

Image borders With any sliding-window operation, it is preferable if the resulting image has the same size as the input. But consider computing

$$(I \star f)(1,1) = \sum_{i=-m}^{m} \sum_{j=-n}^{n} I(1+i, 1+j) f(i,j), \tag{1.15}$$

we see that it requires that we sample the image outside the image borders, e.g., at $I(0,0)$. To deal with this we use padding. That is we define values for pixels outside the image. Figure 1.5 illustrates two common ways to do this.

1.4 Nonlinear filtering

Naturally, any operation that works on a small image can be applied to a larger image in a sliding window manner. Common examples are max- and min-filtering, where the maximum/minimum value of each sub-patch is saved.

Another filtering technique that can be useful is non-maximum suppression. An input patch, normally of size 3×3 yields the output *true* if the central pixel has a strictly larger value than the others, and *false* otherwise. Applying

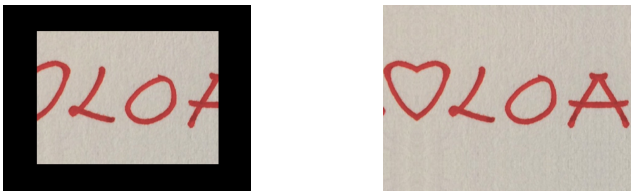


Figure 1.5: The left image illustrates zero padding. All pixels outside the original image border are given the value zero. A problem is that we are creating sharp edges at the original image border. The right image illustrates the *symmetric* solution. Values inside the image are mirrored outside the image to avoid creating edges. This yields a visually pleasing result but as the example shows it could give rise to unrealistic structures or, as in this case, realistic but deceiving structures.

this operation in a sliding window will create a response image that indicates all local maxima in the image

Counting blood cells in a microscopic image is a standard medical procedure, but to this end, the detection result in Figure 1.4 is useless as each cell gives rise to a number of *cell* pixels. Simply counting the yellow pixels will not give an accurate estimate of the number of cells. We can deal with this using non-maximum suppression. Before thresholding, we use non-maximum suppression to suppress any pixels which are not strict local maxima with respect to the classifier score $I \star w$. Figure 1.6 shows the resulting detections for our blood cell example.

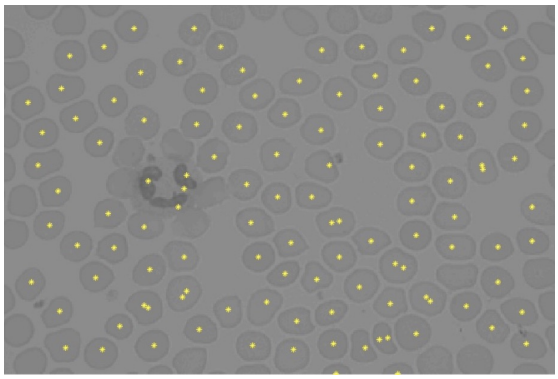


Figure 1.6: Detections from the sliding-window detector with non-maximum suppression.

1.5 Statistical measures of similarity

There is an alternative way of deriving the linear classifier of Section 1.2. Recall that our goal was to find images which are similar to a template image. Covariance is essentially the simplest statistical measure for this. In analogy to the statistical definition, we define the covariance of two images (of the same

size) as

$$Cov(I,T) = \frac{1}{N_{el}} (I - \mu_I \mathbb{1}) \cdot (T - \mu_T \mathbb{1}) = \frac{1}{N_{el}} I \cdot (T - \mu_T \mathbb{1}) \tag{1.16}$$

The second equality is easy to prove, as

$$\mathbb{1} \cdot (T - \mu_T \mathbb{1}) = \mathbb{1} \cdot T - \mu_T \mathbb{1} \cdot \mathbb{1} = N_{el}\, \mu_T - \mu_T\, N_{el}. \tag{1.17}$$

Note that (1.16) is exactly the dot product of I and

$$w = \frac{1}{N_{el}} (T - \mu_T \mathbb{1}), \tag{1.18}$$

which is the weight matrix that we arrived at in Section 1.2.

(a) $J = I + 0.3\mathbb{1}$ (b) Original image, I (c) $J = 1.3\, I$

Figure 1.7: The same image but with varying brightness and contrast.

Viewed as a image similarity measure, covariance has many problems. Figure 1.7 shows the effect of applying simple mathematical functions to the pixel intensities of an image. Similar changes can occur due to varying light conditions. One characteristic of the covariance measure is that it depends on the contrast of the image. With $J = kI$. We get

$$Cov(I,J) = Cov(I,kI) = \frac{1}{N_{el}} (I - \mu_I \mathbb{1})(kI - k\mu_I \mathbb{1}) = k\, Cov(I,I). \tag{1.19}$$

This means that in terms of covariance, I is *more* similar to J than it is to itself. This is hardly what we expect from a similarity measure. If we want to avoid this peculiarity and the contrast sensitivity, we can use correlation instead. Just as in statistics, we define the variance of an image $Var(I) = Cov(I,I)$, and the standard deviation as $\sigma_I = \sqrt{Var(I)}$. We get the correlation of two images by normalizing the covariance,

$$Corr(I,J) = \frac{Cov(I,J)}{\sqrt{Var(I)}\sqrt{Var(J)}}. \tag{1.20}$$

It is easy to verify that with respect to the correlation measure, all three images in Figure 1.7 are interchangeable.

Chapter 2

Filtering, gradients and scale

In the last chapter we used filtering mainly to apply a linear classifier in sliding window on a larger image. Now we will see that linear filtering is much more versatile than that and how it can be used to detect edges and robustly recognize objects across different scales and viewing angles.

2.1 Gradients and edge detection

The gradient at position (x, y) in the image is

$$\nabla I(x,y) = \begin{pmatrix} I'_x(x,y) \\ I'_y(x,y) \end{pmatrix}. \tag{2.1}$$

To approximate the derivatives, we use finite differences, e.g.,

$$I'_x(x,y) \approx \frac{I(x+1,y) - I(x-1,y)}{2}. \tag{2.2}$$

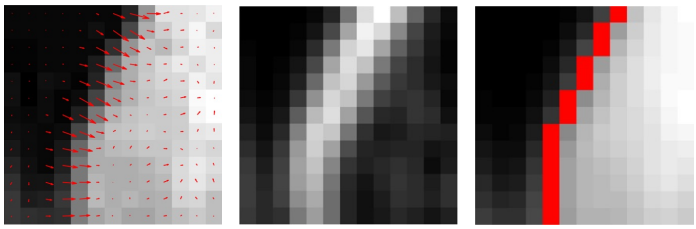
Note that we approximate all derivatives in a larger image using filtering,

$$I'_x \approx I \star (-0.5 \quad 0 \quad 0.5). \tag{2.3}$$

Similarly

$$I'y \approx I \star \begin{pmatrix} -0.5 \\ 0 \\ 0.5 \end{pmatrix}, \tag{2.4}$$

will approximate the vertical derivative. Figure 2.1a shows a small example image. At each pixel the approximate image gradient has been drawn as a red arrow. As expected the gradients point from darker regions (corresponding to low pixel values) into the brighter regions (corresponding to higher pixel values).



(a) (b) (c)

Figure 2.1: (a) Example image with gradients drawn as red arrows. (b) Image indicating the length of the gradient vectors. (c) Original image with resulting edge pixels after thresholding and non-maximum suppression overlaid in red.

2.1.1 Edge detection

Perhaps the most classical use for image gradients is edge detection. An edge point is more or less a point where the image changes sharply, that is, where the magnitude of the gradient

$$|\nabla I| = \sqrt{(I'_x)^2 + (I'_y)^2} \tag{2.5}$$

is large. Hence we can perform edge detection by computing the magnitude of the image gradients and then perform thresholding and non-maximum suppression as we did for the cell detection problem in the Chapter 1. However, as we are looking for a one-dimensional structure, we only require edge point to be a local maximum in a direction perpendicular to the edge, that is, along the gradient direction. Exactly what this means varies between different edge detection techniques. One simple implementation is to consider only those two neighbours which lie (approximately) along the gradient direction. Figure 2.2 shows an example. The gradients points northwest and the gradient magnitude at the central pixel is larger than that for the neighbours indicated in red. Hence the central pixel is accepted as an edge point. Figure 2.1c shows an example of edge points after non-maximum suppression.

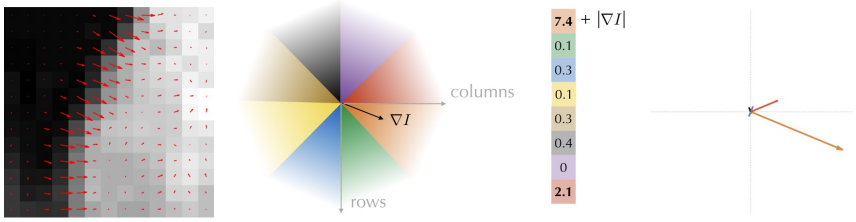
Figure 2.2: For edge detection we only want to do non-maximum suppression along the image gradient, that is perpendicularly to the edge. A simple technique is to check whether the gradient magnitude of the central pixel is larger than the two neighbours along the gradient direction (indicated in red).

2.2 Gradient-based similarity measures

Gradients are also used to describe a small patch of an image. So far we have used covariance and correlation to measure image similarity. They treat the images as vectors and completely ignore the two-dimensional structure. Figure 2.3, illustrates the the limitations of this approach. For many objects a slight change of perspective is all it takes to change almost all pixel values dramatically.

In the human visual system there are neurons, which are sensitive to certain gradient directions but ignores the exact position of these gradients. It is believed that this is one factor that allows us to recognize objects under perspective distortion. This idea of collecting a certain type of information from a small region in the image is called *pooling*. We will encounter different versions of pooling when working with convolutional neural networks in Chapter 5. Here, we will consider one specific variant called gradient histograms.

Figure 2.3: Perspective changes are poorly handled by covariance or correlation. The absolute difference image on the right shows that the correlation will be very low. Despite this a human would deem these image rather similar.



(a) (b) (c)

Figure 2.4: (a) Example image with gradients. (b) Each gradient is classified according to orientation and its length is added to the corresponding histogram bin. (c) Illustration of the resulting histogram.

Figure 2.4a shows an example image with an estimated gradient at each pixel. Each estimated gradient is assigned to one of eight bins depending on its orientation; see Figure 2.4b. For each bin we sum the length of the gradients in that bin. This gives us a vector of eight positive numbers, being our histogram. A nice illustration of such a histogram is to draw a bouquet of vectors as in Figure 2.4c. Each vector represents one orientation bin, and its length is the sum of the lengths of all gradients assigned to that bin. Figure 2.5 returns to

the checkerboard example. Note that the histograms of the checkerboards are very similar despite the perspective distortion.

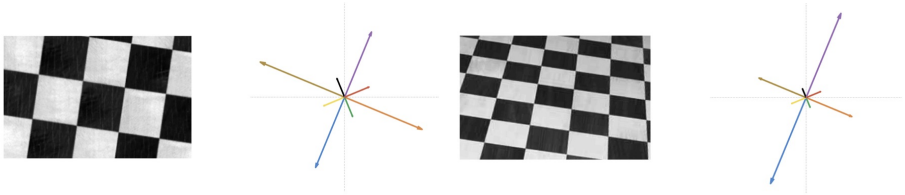


Figure 2.5: A checkerboard under different viewing angles and the corresponding gradient histograms. Unlike correlation and covariance, this measure views the image patches as fairly similar.

2.3 Gaussian filtering and scale

Figure 2.6: Zebras and horses are similar on a coarse scale, but very different on a fine scale.

Figure 2.6 shows an image of a zebra and one of a horse. How can we force a computer to see the similarities between these images, despite the zebra being striped. We need a way to disregard finer structures and look at the big picture. The normal way to do this is by Gaussian filtering, also called Gaussian smoothing. A Gaussian filter is simply a discretized version of the Gaussian function

$$g_\sigma(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right). \tag{2.6}$$

It is not entirely necessary to use a Gaussian, but it does have some desirable characteristics. It does remove fine structures from the image as can be seen for the zebra example in Figure 2.7. Filtering twice with a Gaussian has the same effect as filtering with one Gaussian with a larger variance,

$$I \star g_{\sigma_1} \star g_{\sigma_2} = I \star g_{\sigma_3} \tag{2.7}$$

where $\sigma_3^2 = \sigma_1^2 + \sigma_2^2$. If we compare to a museum visitor moving away from an artwork, this is pretty much saying that taking 3 steps back and then 7 more has the same effect as taking 10 steps back at once.

Figure 2.7: Gaussian filtering suppresses the fine structures.

Gaussian filtering gives us a way to view an image at different scales. We move from a fine scale to a coarser one by filtering with a Gaussian. Working with an image on different scales, it is convenient to represent it as a function of three variables $L(x, y, \sigma^2)$. The value at a certain position depends not only on the position but also on the scale at which we choose to view the image. This is a continuous model and the discrete images we work with are sampled from this continuous function.

So what is $L(x, y, 0)$? Is that the digital image without any Gaussian smoothing? Due to the size of individual sensor elements in a digital camera, there is some smoothing even in the original image. One could debate if this smoothing is really Gaussian, but it is usually a good enough model. People normally assume that the original image has been sampled from $L(x, y, 0.5^2)$ (in pixel units).

2.3.1 Scale and pixels

Note that scale here, is not directly linked to the number of pixels in the image. An image can have a coarse scale but still have many pixels. However, having many pixels for an image without fine details is inefficient, so in any application where efficiency is an issue, you should downsample the coarse scale representations.

2.4 Detecting object size

In the first chapter, we saw how to detect the position of objects using a linear classifier in a sliding window fashion. What if similar objects of different sizes are visible in the image? Just computing a sliding covariance will not do the trick since the covariance of objects of different sizes will be low. The solution is to use templates of different sizes. To illustrate, we will use a very basic example. Consider an image of a circle with radius 67 pixels. To estimate the radius, we form a large set of templates with radii between 50 and 100 and compute both covariance and correlation between the templates and the query image. The resulting curves are shown in Figure 2.9. As expected correlation attains its maximum when the template is identical to the query, but not the covariance curve that reaches its maximum at $r = 62$. This is related to the fact

noted in Chapter 3,

$$Cov(I, 2I) = 2\,Var(I) > Cov(I, I) \tag{2.8}$$

showing that covariance is not maximal for identical images. Still covariance can be used to estimate relative scale, as doubling the circle radius would lead to doubling the covariance-based radius estimate.

Figure 2.8: Circle templates used for size estimation. Note that the whole template should be resized, so the relationship between black and white regions is kept.

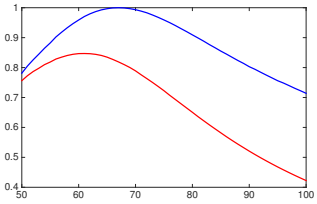


Figure 2.9: Size estimation for a circle with radius 67: The blue correlation curve has a maximum at exactly 67, but not the red covariance curve. Covariance is less appropriate for size estimation as it is not maximal for identical images.

A more efficient solution

We can view the correlation curve in Figure 2.9 as a function, $f(s)$, of the radius, s. Sampling this function as densely as we have done in Figure 2.9 is quite expensive, so let us try fewer samples at

$$s \;=\; 10,\; 20,\; 30, \ldots \tag{2.9}$$

To find the correct radius we seek a local maximum of $f(s)$. Among these, $s = 70$ is a local maximum in the sense that it yields a larger response than its neighbours $s = 60$ or $s = 80$. Now let us see if we can improve this estimate. It is convenient to first make a change of variables. Let

$$\rho = \frac{s - 70}{10}. \tag{2.10}$$

This moves the local maximum at $s = 70$ to $\rho = 0$ and the neighbouring $s = 60$ and $s = 80$ to $\rho = -1$ and $\rho = 1$. Recall from calculus that

$$f(x) = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{h}. \tag{2.11}$$

Using this formula with small values of h gives us a means to estimate derivatives. For example,

$$f'(0) \approx \frac{f(1) - f(-1)}{2} = a. \tag{2.12}$$

Applying this formula twice, gives us an approximation of the second derivative,

$$f''(0) \approx f(1) - 2f(0) + f(-1) = b. \tag{2.13}$$

Using the Taylor expansion,

$$f(\rho) \approx f(0) + \rho\, f'(0) + \frac{\rho^2}{2} f''(0) \approx f(0) + \rho a + \frac{\rho^2}{2} b \tag{2.14}$$

To find a local maxima we look for stationary points,

$$0 = f'(\rho) \approx a + \rho b \Rightarrow \rho = -\frac{a}{b} \tag{2.15}$$

Applying this to the circle example yields an estimate of

$$s_{max} = 10\rho + 70 = 67.9 \tag{2.16}$$

reducing the error significantly. If we sample at 60, 65 and 70, we get an estimate of 67.1. Naturally the quality of the estimate also depends on the appearance of the template.

2.4.1 Detecting position and size

To detect objects of different sizes, we use multiple classifiers of multiple sizes. This produces one score map for each classifier. Just like before we will use non-maximum suppression to avoid getting multiple detections at the same object. But this time we have classifier scores for differently sized templates, so we will require a detection to be a local maxima spatially, but also larger than the corresponding pixels on the neighbouring scales. Figure 2.10 shows an example. This allows us to detect not only the position of an object but also a rough scale. In a moment, we will see how this extra information can be used.

Sub-pixel precision

The responses at different positions (x, y) and scales s which are shown in Figure 2.10 can be seen as sampled from a function $f(x, y, s)$. Just as in the one-variable case in Section 2.4, we can often obtain a more accurate position of the local maxima by considering the Taylor expansion. We start with one of the

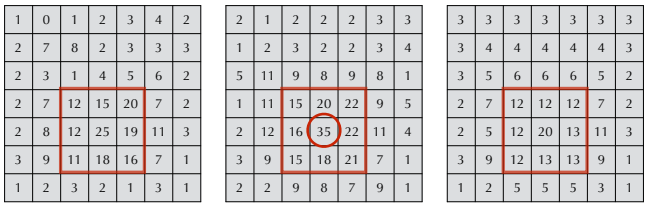


Figure 2.10: Sliding window with classifiers of increasing size produces three response maps. The pixel in the red ring is not only a local maximum in that response but also larger than the corresponding 18 pixels on the neighbouring scales.

discrete sample point that was a local maxima (cf. Figure 2.10). After a change of variables we can assume that this point is at $(0,0,0)$ and that the neighbours are at $(0,0,1)$, $(0,1,0)$, etc. The Taylor expansion is

$$f(x,y,s) \approx f(0,0,0) + \begin{pmatrix} x & y & s \end{pmatrix} \nabla f(0,0,0) + \frac{1}{2} \begin{pmatrix} x & y & s \end{pmatrix} H(0,0,0) \begin{pmatrix} x \\ y \\ s \end{pmatrix} \tag{2.17}$$

where $H(x,y,s)$ is the so called Hessian matrix be a 3×3-matrix with all the second-order derivatives. Again, we have to estimate these derivatives with difference quotients, for example,

$$f''_{xs} = \frac{1}{4} \left(f(1,0,1) + f(-1,0,-1) - f(1,0,-1) - f(-1,0,1) \right). \tag{2.18}$$

To find a stationary point we set differentiate the approximation and set to zero,

$$\nabla f(x,y,s) \approx \nabla f(0,0,0) + H(0,0,0) \begin{pmatrix} x \\ y \\ s \end{pmatrix} \Rightarrow$$

$$\begin{pmatrix} x \\ y \\ s \end{pmatrix} = -H^{-1}(0,0,0) \nabla f(0,0,0). \tag{2.19}$$

Chapter 3

Invariant local features

We have already seen a few ways to describe the similarity of two images and how this can be used to perform image classification. For the typical classification task, we will soon learn much more powerful techniques than measuring the similarity to a given template, but for other tasks image similarity measures are very useful, especially for patch matching between pairs of images. Figure 3.1 shows two images with a set of defined patches. Using a image similarity measure, we could establish correspondences between the patches. In later chapters, we will learn how a set of such correspondences allows us to compute the motion of the camera and the geometry of the scene. Hence patch matching is the first step in computing large 3D models that could be used for mapping and localization. That explains why the topic of this chapter, being SIFT features, have been, and still are, extremely popular with thousands of citations. The acronym SIFT stands for Scale-Invariant Feature Transform and includes both the descriptor and a method to detect interest points.

Figure 3.1: Patch matching is often the first step in computing the geometry of a scene.

3.1 SIFT: The descriptor

We have already looked at the idea behind SIFT when we considered gradient histograms in the previous chapter. To recapitulate, consider the two images in Figure 3.2. A human can easily accept that these are two images of the same

shape but viewed from slightly different angles. However, considering the image pixel per pixel as we do if we compute correlation or covariance, they are not similar at all. As we saw in the last chapter, gradient histograms are much more robust to perspective distortion, but on the downside, a single 8-bin histogram has very limited descriptive power.

Figure 3.2: Two geometric shapes under perspective distortion.

The compromise used in SIFT is to divide the input image (or patch) into a number of smaller regions and use one gradient histogram for each region. For illustrative purposes, we will use 2×2 regions although the most common choice is 4×4. Figure 3.3 illustrates the principle. The example image has been divided into 2×2 regions. For each region we extract the gradients and compute a gradient histogram (shown on the right). These gradient histograms are simply stacked into a single vector. This vector is the actual SIFT descriptor, owing its name to the fact that it describes the appearance of the original image patch. Recall that each gradient histogram consists of 8 non-negative values. This means that the SIFT descriptor has 8 times the number of regions bins. For our example we get a descriptor of length 32, whereas the original SIFT descriptor has length 128 (as it uses 4×4 regions). Normally, the full descriptor vector is normalized to reduce the effect of contrast differences between different images.

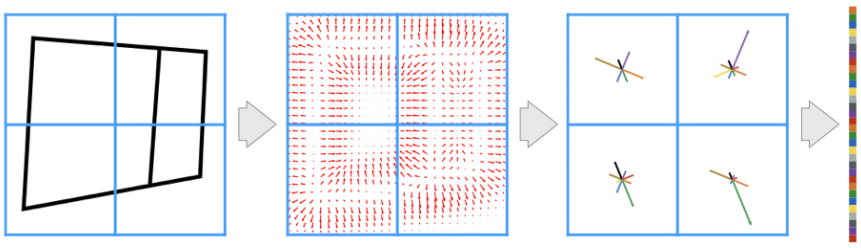


Figure 3.3: To compute a SIFT descriptor we divide the image into a number of smaller regions and compute a gradient histogram for each region. Finally, all the histograms are stacked into a single descriptor vector, that is normalized to unit length.

3.2 Feature extraction: Overview

As we have already said, the main application for SIFT and other local features is in patch matching, that is, the process of establishing point to point correspondences between a pair of images; see Figure 3.4. The steps are roughly the following

1. Detect a set of separated points in the image which are stable in the way that roughly the same points will be detected in other images of the same scene. We will refer to these as interest points

2. Extract a patch around each detected interest point.

3. Compute a descriptor for each patch.

Having done this for a pair of images, we can start looking for similar descriptors between the two images; see Figure 3.4. With a good descriptor we should be able to establish a large number of correct point-to-point correspondences. In the rest of this chapter, we will consider the various steps in greater detail.

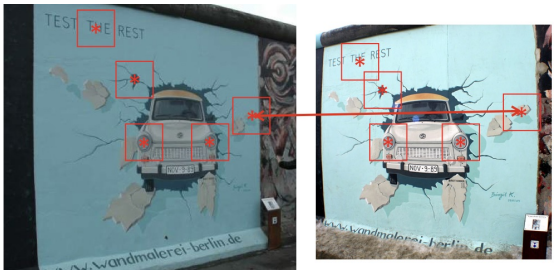


Figure 3.4: Extracting SIFT features. First a set of separated points are detected. Around each point a patch is extracted and the SIFT descriptor is computed for this patch. Given multiple images the descriptors can be used to match corresponding points.

3.3 Blob detection

The first step in feature extraction, is to compute a set of separated interest points are detected. These points need to have a well-defined centre position and a detectable size. Our simplest option is probably to look for blobs in the image. In Chapter 2 we saw how to detect both the scale and position of a simple structure in an image. In the case of blobs, we can use difference-of-Gaussians filters. Just as the name implies, a difference-of-Gaussians filter is the difference of two Gaussian filters

$$f_{dog} = g_\sigma - g_{k\sigma}. \tag{3.1}$$

With $k > 1$, this yields a filter with a sharp peak in the middle surrounded by a ring of negative values; see Figure 3.5. These filters yield a high response on bright blobs and low response on dark blobs. Hence, both local maxima and local minima correspond to relevant structures in the image, so we will keep both kinds.

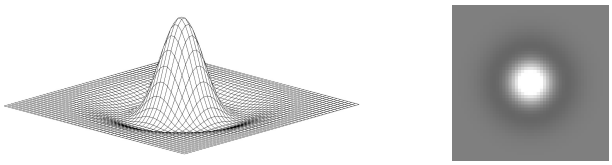


Figure 3.5: A difference-of-Gaussians filter viewed as a function plot (left) and as an image (right)

By using multiple difference-of-Gaussians filters with varying σ, we can detect both position and size for blobs in the image. As we saw in Section 2.4.1, these measurements can be improved to sub-pixel precision by working with the Taylor approximation of the response map.

3.3.1 Avoiding edges ⋆

Recall that we need the detected points to have a well-defined position. If the exact position moves between different images, we might get different descriptors and the matching can fail. One problem with difference-of-Gaussians is that they get fairly high values at edges as well. This leads to random local extreme points at edges but these are tend to move between different views so they are not well suited for our needs.

To see how we can separate local extrema arising at edges from those at blobs. Let $f(x, y)$ denote the difference-of-Gaussians response and assume that we have found a local maximum at (a, b). At least if this was detected with sub-pixel precision, we can assume that $\nabla f(a, b) = 0$. Hence, the Taylor approximation tells us that

$$f(x,y) \approx f(a,b) + \frac{1}{2} \begin{pmatrix} (x-a) & (y-b) \end{pmatrix} H(a,b) \begin{pmatrix} x-a \\ y-b \end{pmatrix}. \qquad (3.2)$$

This means that the behaviour of f about (a, b) is determined entirely by the Hessian, $H(a.b)$. Recall that this is a symmetric matrix and hence there is has ON-basis of eigenvectors. If we switch to this basis (and also move the origin to (a, b)) we get the following expression for f

$$f(\tilde{x},\tilde{y}) \approx f(0,0) + \frac{1}{2} \begin{pmatrix} \tilde{x} & \tilde{y} \end{pmatrix} \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix}, \qquad (3.3)$$

where λ_1 and λ_2 are the eigenvalues of $H(a, b)$. Note that λ_1 determines how fast the f decreases if we move away from the local maxima along the $\tilde{x}$-direction whereas λ_2 plays the same role for the $\tilde{y}$-direction. If our local max is

at an edge, moving along the edge normally does not change the response very much, whereas moving perpendicularly to the edge does. Hence, the ratio

$$r = \frac{\lambda_1}{\lambda_2} \tag{3.4}$$

can be used to exclude local extreme points at edges. If $r > 10$ or $r < 0.1$, the local extreme point is likely to lie on an edge, so we discard it.

3.4 Computing the descriptor

Having detected both the position and size for a set of blobs in the image, we will use the estimated sizes to obtain scale invariance. Consider the images in Figure 3.6. The same dark blob has been detected in both images. We have a *scale invariant* method if it assigns similar descriptors to these two blobs. The first step to achieve this is to make the sizes and offsets of the 4×4 SIFT regions proportional to the detected blob size; see Figure 3.6.

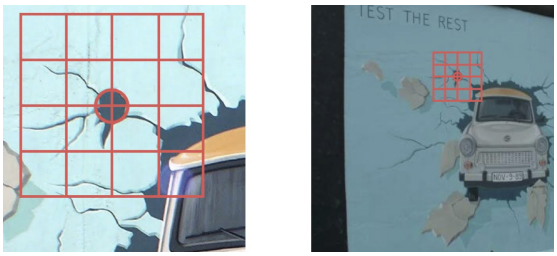


Figure 3.6: For scale invariance, we choose descriptor patch and the descriptor regions proportional to the size of the blob.

The next step is to compute a gradient histogram for each of these regions, but we need to do this at the correct scale. Figure 3.7 shows two images of the same umbrella but seen at different distances. Let us pretend that these are SIFT regions for which we want to estimate a gradient histogram. Figure 3.7 shows the result without any preprocessing. As you can see they get very different gradient histograms. According to the discussion in Chapter 2, we should be able to go from the finer scale of the left image to the coarser scale of the right one by means of Gaussian filtering.

To determine the amount of filtering required, we use the scales determined by the blob detector. Assume that in the left image the left image the detected blob size was 5 pixels. We want to compute the gradients in a smoothed version of the image, that is, in $L(x, y, \sigma^2)$, where σ is proportional to the detected scale. In this case let us choose $\sigma = 5$. Recalling that the image itself can be seen as a sampling from $L(x, y, 0.5^2)$. Hence we filter the image with a standard deviation σ_1 satisfying

$$0.5^2 + \sigma_1^2 = 5^2. \tag{3.5}$$

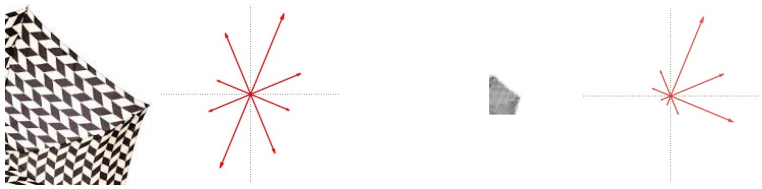


Figure 3.7: Umbrella at different distances along with the corresponding gradient histograms.

In the other image the blob has scale 0.5, so no extra filtering is required. After filtering, we get the result in Figure 3.8. Note that as we are computing gradient histograms for images of different resolution, the absolute values in the histogram will not be the same, but as the last step in computing SIFT-like descriptor is normalization, this does not affect the final descriptor.

Figure 3.8: Histograms for the umbrella images after applying Gaussian filtering to the high-resolution image. Remark: The scale of the histogram plots is not the same.

3.5 Rotation invariance

While the histogram binning makes SIFT-like descriptors robust to small orientation changes it cannot handle large rotations as shown in Figure 3.9. If rotation invariance is desirable we can obtain this in the following way. We extract circular patch around the interest point and make a gradient histogram for this patch. The bin with the largest value corresponds to the dominant gradient direction.

Knowing a dominant gradient direction we can align the SIFT regions with respect to this direction; see Figure 3.10. We also rotate all gradients with respect to this direction before computing the descriptors. Assuming that the dominant gradient direction is stable, this will yield rotation invariance.

Remark. Note that invariance always comes at the cost of reduced descriptive power. If the image orientation is known it is preferable to avoid rotation invariance. Figure 3.11 indicates that the human visual system is not rotation-invariant.

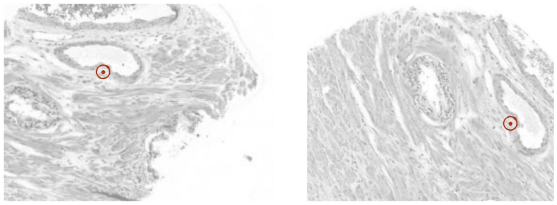


Figure 3.9: These detected interest points in images with different orientation

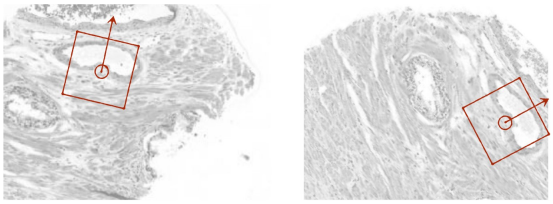


Figure 3.10: By aligning the SIFT regions to the dominant gradient direction and rotating the image gradients in the corresponding way, we can obtain rotation invariance.

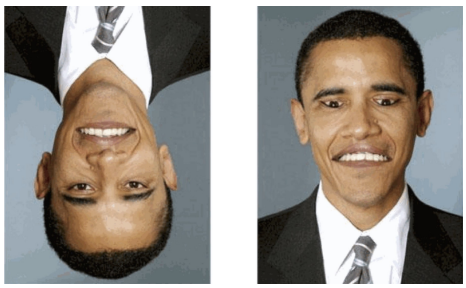


Figure 3.11: At least for faces, the human visual system is not rotation invariant.

3.6 Descriptor matching

As we have already indicated, the typical application for SIFT descriptor is to determine corresponding points/patches between two images; cf. Figure 3.1. This can be used for image alignment or to estimate the three-dimensional geometry of a scene.

Figure 3.12 shows two images from Lab 4. Establishing correspondences between the interest points will allow us to reconstruct their position in 3D. Let q_i, $i = 1, 2, 3 \ldots$ be the 128-dimensional descriptor vectors for the points from the left image and d_j, $j = 1, 2, 3 \ldots$ the same for the other image. Let us search for a good match for q_1. The best candidate is the descriptor in image 2 with shortest euclidean distance to $|q_1 - d_j|$. But how can we tell if the *best* candidate

Figure 3.12: Two images of the same scene, with detected interest points in red.

is good enough? Figure 3.13 shows that having very similar descriptors does not necessarily mean that we can be certain to have a correct match.

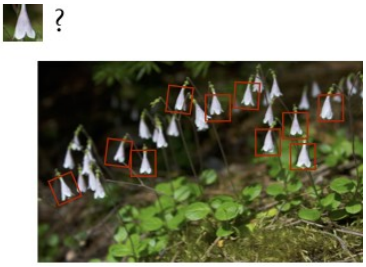


Figure 3.13: A candidate match can be uncertain although the descriptors are very similar in terms of $|q_i - d_j|$. The reason is often that there are many similar patches in an image.

Lowe ratio criterion. The inventor of SIFT, David Lowe, suggested the following criterion, which we will call the Lowe ratio criterion

$$\frac{\text{distance to the best match}}{\text{distance to the next best match}} < \tau, \tag{3.6}$$

where τ between 0.6 and 0.9 are reasonable values. This ratio is a better measure of how certain we are about a match. In the flower example the distance to the best match is short since the flowers are very similar to the query patch. But the distance to the next best match is also small as there are many flowers. Hence the ratio is fairly close to 1 indicating an uncertain match.

Symmetric matching. An alternative to the Lowe ratio criterion is to only accept symmetric matches. Again consider matching a descriptor q_1 in one image. Assume that, of all the descriptors in the other image, d_7 is the best match, that is,

$$\arg\min_i |q_1 - d_i| = 7. \tag{3.7}$$

This match is symmetric if

$$\arg\min_j |q_j - d_7| = 1. \tag{3.8}$$

Descriptive power

To illustrate the descriptive power of the SIFT descriptor, consider the image in Figure 3.14 for an example.

Figure 3.14: A 3D model of central San Fransisco. Each dot is a SIFT feature that has been matched and triangulated in 3D. In total there are about 100 million points. Despite this it is possible to match SIFT descriptors extracted from a query image and find its location in the 3D model!

Chapter 4

Learning a classifier

In Chapter 1 we used templates to construct simple linear classifiers. In this chapter we will consider more elaborate schemes to construct a linear classifier.

Recall that a linear classifier is simply a weight image w together with a threshold τ. Given a patch, I, from an image, we will *classify* this as a cell patch if the $I \cdot w > \tau$. Equivalently we can say that I is classified as a cell if

$$I \cdot w + w_0 > 0, \tag{4.1}$$

where we have eliminated the threshold by adding an unknown constant. This formulation will be more convenient. The purpose of this chapter is to find a structured way of determining good values for w and w_0. To do so, we need examples, that is, cell patches and non-cell patches. If the task is to detect cells, we will refer to the cell examples as positive examples and anything else as negative examples; see Figure 4.1.

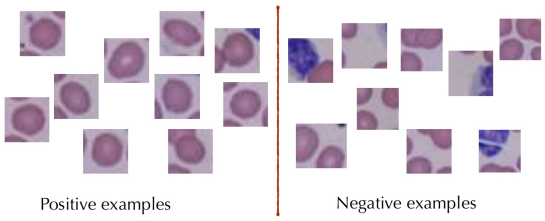


Figure 4.1: Example patches for cell detection.

Detection and classification

Recall that any classfier can be turned into a detector by applying it in a *sliding window* manner. To detect cells we move a window across the image and use a *classifier* to tell if the patch currently inside the window is a cell or not.

After obtaining the classifier score for each possible window, we can use non-maximum suppression to avoid getting multiple detections at the same cell. In the rest of this chapter, and in the next chapter, we will focus on the problem of learning a classifier. But keep in mind that if we have a good classifier, we can use a sliding window to perform detection.

4.1 How to evaluate a classifier

Naturally we could measure the performance of a given classifier by the rate of misclassifications it makes. But consider a classifier that is supposed to tell the different between birthmarks and skin cancer. If we test this classifier on a set of birthmarks photgraphed at a dermatologist, it is likely that 99% of these are ordinary birthmarks rather than skin cancer. Hence a classifier that *always* predicts the label *birthmark* gets it right in 99% of the cases. The concepts of precision and recall are designed to highlight this type of behaviour. Given a class $\textsc{a}$, the

$$\text{precision for class } \textsc{a} = \frac{\text{number of examples } \textit{correctly} \text{ classified as } \textsc{a}}{\text{number of examples classified as class } \textsc{a}} \tag{4.2}$$

and the

$$\text{recall for class } \textsc{a} = \frac{\text{number of examples } \textit{correctly} \text{ classified as } \textsc{a}}{\text{number of examples from class } \textsc{a}}. \tag{4.3}$$

In the skin cancer example, the recall for the skin cancer class is 0, indicating that this is indeed *not* a good classifier.

4.1.1 Negative log-likelihood

In the process of choosing parameters, measures such as precision and recall has certain problems. Figure 4.2 illustrates how well two different filters perform at separating positive training examples (green squares) from negative (red dots). In terms of the number of misclassifications, they are just as good (or bad). But the upper filter looks much more certain about the correct classifications and, under reasonable assumptions, is more likely to perform well on new data.

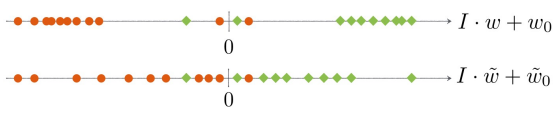


Figure 4.2: The output of the filter is marked with a red dot for negative examples and a green square for positive examples. In terms of the number of misclassifications the two filters perform equally well.

Logistic regression is a method designed to deal with this issue. Class probabilities are estimated for each example using the logistic sigmoid function,

$$p = \frac{e^y}{1+e^y} \quad \text{with} \quad y = I \cdot w + w_0 < 0, \tag{4.4}$$

and the likelihood is used to to measure the total performance of a classifier. Figure 4.3 shows the effect for one of the filters from Figure 4.2.

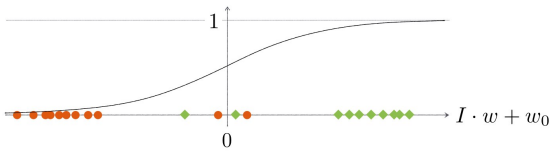


Figure 4.3: The logistic sigmoid assigns a probability to each training example. For this filter most positive examples have a probability close to one and most negative examples have a probability close to zero. The result would be more mixed for $\tilde{f}$ of Figure 4.2.

Consider a training set with n examples, $I_1, \ldots, I_n$. Let S_0 be the set of indices to negative samples and S_1 the set of indices to positive samples. The likelihood over the training set is

$$\prod_{i \in S_0} (1 - p_i) \cdot \prod_{i \in S_1} p_i \,, \tag{4.5}$$

where p_i refers to the network output for the ith training example. This is a good measure of the performance of a classifier, but working with the product is inconvenient. Hence, we take the logarithm of this expression. Finally, most people prefer minimizing loss functions rather than maximizing performance functions, so we add a minus sign, arriving at the negative log-likelihood:

$$L(\theta) = \sum_{i \in S_0} -\ln(1 - p_i) + \sum_{i \in S_1} -\ln p_i = \sum_i L_i(\theta), \tag{4.6}$$

where θ is shorthand for a vector of all the parameters in our model. We will refer to $L_i(\theta)$ as the *partial* loss.

4.2 Multiple classes and softmax

In many cases we want to separate between more than two classes. Maybe there are multiple cell types in our images and we want the classifier to tell the difference. If the patch is not large enough to contain multiple cells, we want the class probabilities to sum to 1. This can be achieved with a generalization of the logistic sigmoid, called softmax. Given a set of numbers y_c, we can produce probabilities p_c such that

$$\sum_j p_j = 1, \quad \text{and} \quad y_c > y_j \Leftrightarrow p_c > p_j \tag{4.7}$$

by setting

$$p_c = \frac{e^{y_c}}{\sum_j e^{y_j}}. \tag{4.8}$$

As for the loss function, we can still use negative log-likelihood. Let S_c be the indices of the samples from class c and let $p_{c,i}$ be the output probabilities for sample i. Then,

$$L(\theta) = -\sum_{i \in S_0} \ln p_{0,i} - \sum_{i \in S_1} \ln p_{1,i} - \ldots - \sum_{i \in S_n} \ln p_{n,i} = \sum_i L_i(\theta). \tag{4.9}$$

4.3 Stochastic gradient descent

We now have a loss function that measures how good a certain classifier is. The next thing that we need is a way to minimize this loss. Except in very special cases, there is no method guaranteed to find the global minimizer. Instead we have to rely on local optimization. Starting from a random starting point, we try to move in a direction that will decrease the loss.

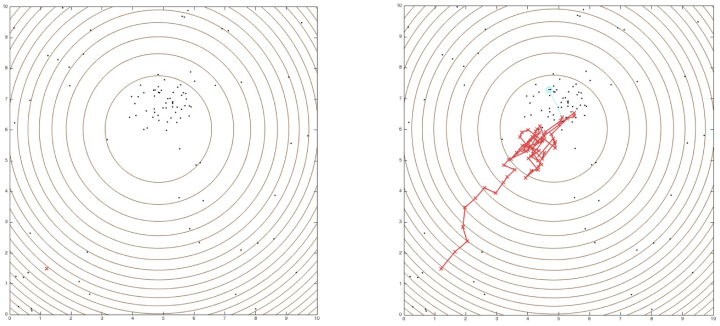


Figure 4.4: The black dots are 2d measurements of some unknown parameter θ. Level curves for the sum of squared residuals are drawn in brown. The red line in the right plot follows the parameter updates when performing stochastic gradient descent.

Let us look at a two-dimensional example. We have asked a number of people to mark Linköping on a map. The result is a set of two-dimensional measurements x_i, shown in Figure 4.4. Our model is that these are all noisy measurements of a (2D) parameter θ. For each measurement we get a residual

$$r_i(\theta) = x_i - \theta. \tag{4.10}$$

If we assume Gaussian noise, then the maximum likelihood estimate of θ is the solution that minimizes the sum of squared residuals, that is,

$$L(\theta) = \sum_{i=1}^{n} r_i^2(\theta) = \sum_{i=1}^{n} (x_i - \theta)^2 = \sum_{i=1}^{n} L_i(\theta). \tag{4.11}$$

For this specific loss there is actually a formula for the minimizer, but we are using it as an example, so try to forget this. Instead, we will start at a random point (marked with a red x in Figure 4.4), and try to move towards the optimum.

Recall from your calculus course that the gradient direction is the direction in which the function value increases most rapidly. Hence, if we aim to minimize $L(\theta)$, we should move in the direction of $-\nabla L$. The gradient of L is

$$\nabla L(\theta) = \sum_{i=1}^{n} -2(x_i - \theta). \tag{4.12}$$

In the simplest form of gradient descent, we update our parameters according to

$$\theta^{(k+1)} = \theta^{(k)} - \mu \nabla L. \tag{4.13}$$

In the machine learning community, the factor μ is called the learning rate. A large μ means that we take large steps.

What if $\boldsymbol{n}$ is huge? Gradient descent is already a rather efficient method, but for many image classification problems it is still intractable due to the huge number of training examples, n. Stochastic gradient descent is a method designed for this case. Instead of computing the full sum in (4.12), we randomly select one measurement, x_i, and use the update rule

$$\theta^{(k+1)} = \theta^{(k)} - \mu \nabla L_i = \theta^{(k)} + \mu 2(x_i - \theta^{(k)}). \tag{4.14}$$

One interpretation is that we are approximating the gradient ∇L with the gradient of the partial loss ∇L_i. This means that at each iteration, we only need to look at one measurement, x_i. On the right of Figure 4.4 you can follow the rather bumpy path that the parameter values will follow. As you can see the basic variant of stochastic gradient descent does not converge in the normal sense. Instead, it will keep moving around the optimum invariably.

Momentum. A variant of stochastic gradient descent designed to make the path less bumpy, is stochastic gradient descent with momentum. The gradient of the partial loss, ∇L_i, can be viewed as an estimate of the true gradient ∇L. A more conservative estimate is to set

$$g^{(k)} = \gamma g^{(k-1)} + (1 - \gamma) \nabla L_i(\theta^{(k)}), \tag{4.15}$$

and use the update rule,

$$\theta^{(k+1)} = \theta^{(k)} - \mu g^{(k)}. \tag{4.16}$$

The effect is a tendency to keep moving in the same direction and the factor γ is called momentum. You can study the effect in our test case in Figure 4.5.

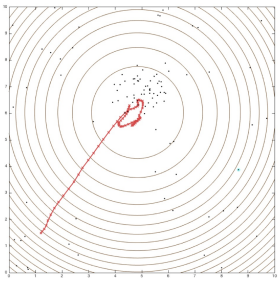


Figure 4.5: Stochastic gradient descent with momentum.

4.4 Learning a classifier

So what happens if we apply this to the linear classifier discussed earlier. After working out a formula for the gradient ∇L_i and deciding on a starting point, we can get to work. Figure 4.6 shows the initial weight image w and the results after a few rounds of stochastic gradient descent. Figure 4.7 shows the resulting cell detections in a new test image.

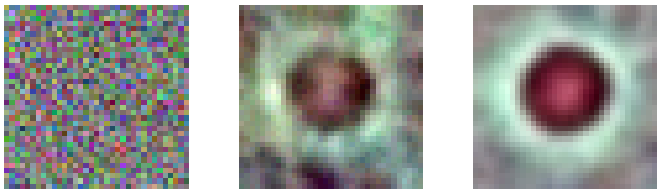


Figure 4.6: A linear cell classifier. The initial (random) weight image (left) and the weight image after 100 (middle) and 1000 (right) rounds of stochastic gradient descent.

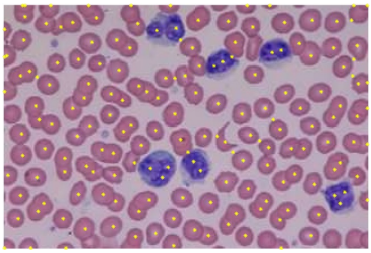


Figure 4.7: Resulting detections from the learned linear classifier.

4.5 Overfitting

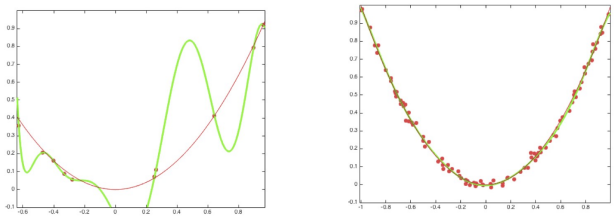
The cell classifier that we just saw, has 1226 parameters. With so many parameters to tune, there is a clear risk of overfitting. Figure 4.8 shows a classic example of overfitting. We have a set of measurements (x, y). The true model for these is that

$$y_i = x_i^2 + e_i \tag{4.17}$$

where e_i is moderately-sized noise. In image analysis, we very rarely know a simple model for the data. Instead we use a complex model with many parameters and hope that it will be rich enough to approximate the true model. In this example it could correspond to using a very high-order model

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \ldots + a_{10} x^{10}. \tag{4.18}$$

Figure 4.8a shows the effect. The fitted model is very accurate for the given training data, but between those points it deviates dramatically from the true model. This means that the model is likely to work very poorly on new data.



(a) (b)

Figure 4.8: (a) Fitting a high-order polynomial model to a few data points, will lead to overfitting. Here, the measurements are shown as red $\star$, the correct model in red and the fitted polynomial in green. Note that new data points will be close to the red curve, so the fitted model generalizes very poorly to new data. (b) With more data the overfitting decreases.

4.5.1 Data augmentation

The best way to reduce overfitting is probably to have more training data; cf. Figure 4.8b. However, sometimes data is hard to come by—or expensive. One example is with medical data, where obtaining manually annotated data often requires that a medical doctor spends huge amounts of time in front of a computer. In such case, data augmentation is a very attractive option.

Data augmentation is basically the process of simulating new annotated data using the data that we have. It is only possible if we have a sense of the natural variation in the type of images that we are considering. Let us once again use the cell images as an examples. All the following operations can be applied to a patch with a centred cell without altering the fact that it is a patch with a centred cell:

- Any rotation or reflection
- Moderate scaling
- Moderate change of brightness
- Adding moderate pixel noise

This means that we can synthesize a large set of cell examples from a single real example, see Figure 4.9.

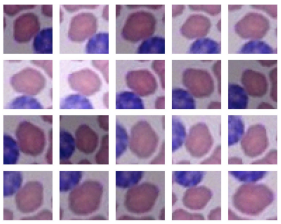


Figure 4.9: Data augmentation is the process of synthesizing new examples from those available.

4.6 Training, validation and test

Even with a very simple classifier there are some difficult choices. For example, to use a linear classifier we need to choose the size of w. A problem here is that a larger size will always be better at minimizing the loss function over the training set. For a filter the explanation is simple, the small filters can be viewed as a subset of the larger filters (with the extra elements set to zero). Optimizing a larger filter we have a larger set of models to choose from and will always find at least as good a solution (in terms of minimizing the loss function over the training set).

This means we cannot choose classifier size by looking at the loss function. To deal with this we use a second set of positive and negative examples, called a validation set. Testing different classifiers on this new set will measure their ability to generalize to new data. A larger w will normally give a smaller loss on the training set, but might have no improvement on the validation set. In this case we probably prefer the smaller filter as it is more efficient. A common strategy to choose this type of hyper-parameters is to test larger and larger classifiers while evaluating their performance on a validation set. As long as the performance is improving we keep increasing the size.

This way a validation set helps to choose hyper-parameters or even choose between different model types. Should we use a learned linear classifier or use SIFT descriptors? However, having chosen one of many model candidates we need to be very careful about assessing the quality of that candidate based on the validation set. Let us look at an example.

Example 1. *Consider* 100 *different classifiers. The misclassification rate of a classifier on a randomly selected dataset can be viewed as a random variable. Let's assume that the* 100 *classifiers are equally good on average, meaning that on a new dataset the misclassification rate of classifier* k *is a random variable,* X_k *that follows a normal distribution with expected value* 10% *and standard deviation* 2%*.*

If we evaluate these 100 *classifiers on our validation set to try to find the best one, they will all get slightly different performance. The average misclassification rate will be close to* 10%*, but if we believe that these classifiers are truly different, we will choose the best performer. Simulating this case* 5 *times in Matlab, we get the following misclassification rates for the best performer,*

$$2.8\%, \quad 3.2\%, \quad 5.2\%, \quad 5.5\% \quad \textit{and} \quad 5.4\%. \tag{4.19}$$

As the example shows, the validation set is often useless to estimate how good a classifier will do in practice. To address this issue we need a third set of manually annotated examples, the *test set*. For comparison, we simulate taking the classifier that performed best on the validation set, and testing it on a test set. Five simulations yielded the following misclassification rates,

$$10.6\%, \quad 12.5\%, \quad 10.4\%, \quad 9.5\% \quad \text{and} \quad 10.4\%, \tag{4.20}$$

being much closer to the true performance. Ideally the test set should be used *once*, when you have a classifier and want to evaluate how good it will perform in practice. As soon as you start re-using it, there is a risk of overestimating the performance.

Chapter 5

Convolutional neural networks

In this chapter, we will look at a powerful kind of classifiers called neural networks. As a template we will use the problem of classifying image patches as cells or non-cells.

5.1 The neuron

Neurons are the basic units of the human brain and the human visual system, making them a natural inspiration when constructing artificial image analysis systems. Figure 5.1 shows the structure of a neuron. The dendrites of the neuron are connected to axons of neurons closer to the eye. If these axons are activated they release neurotransmitters that excite the dendrites. If enough dendrites are excited, an electrical impulse is sent down the axon, in turn exciting neurons closer to the brain.

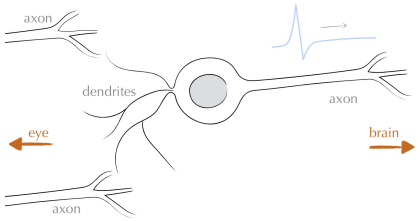


Figure 5.1: A neuron.

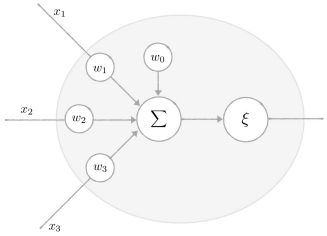


Figure 5.2: Naive model of a neuron.

Figure 5.2 shows our naive model of a neuron. The inputs, x_k, are multiplied by weights, w_k, to reflect that some dendrites are more important than

others. Different functions have been suggested to model the activation of biological neurons. These include the logistic sigmoid, $\tanh$ and the so called rectified linear unit (Relu). The latter is simply $\max\{0, y\}$. For purpose of generality, we will simply use a ξ to denote this function.

5.2 A neural network

The human visual system consists of a large number of neurons. To try to mimic its ability to solve complex visual tasks, we will use a network where each node consists of the naive model just discussed; see Figure 5.3. We will refer to such a node as a neuron.

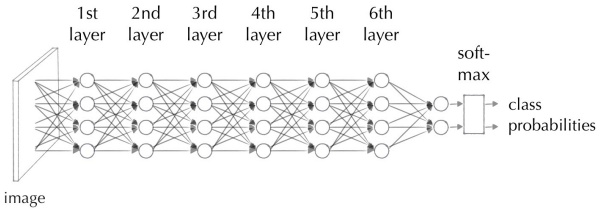


Figure 5.3: A neural network consists of a large number of neurons with a layered structure. Note that only the neurons in the first layer take input from the image. The neurons in the second layer take their inputs from the outputs of the first layer and so on.

5.3 Learning the weights

Recall that our goal is a neural network that takes an image patch and classifies this as either *cell* or *non-cell*. Using a set of manually labelled training examples, as in Figure 4.1, we will try to learn good parameters for this task. With large datasets and many neurons, this is really a typical case for stochastic gradient descent—or some variant thereof. As we saw in the previous chapter, the negative log-likelihood is an appropriate loss function to measure the performance learning classifier. All we need now a starting point and an algorithm for computing gradients. First note that the only parameters in a neural network are the weights w_k and the bias terms w_0. By adding an auxiliary input $x_0 \equiv 1$ to each neuron, we can treat the bias terms in exactly the same way as the other weights.

Initialization

Since we know nothing about the best values of the network weights, we might as well start anywhere. Natural choices are to start at a random point or initializing all weights to zero. Figure 5.4 illustrates the issue with the latter approach. Since all neurons at a certain layer are identical, they will also get

identical derivatives and never diverge. (Different weights within the same neuron can still be different and also neurons in different layers.) In contrast, if the initial weights are small random numbers different neurons will diverge to code for different structures.

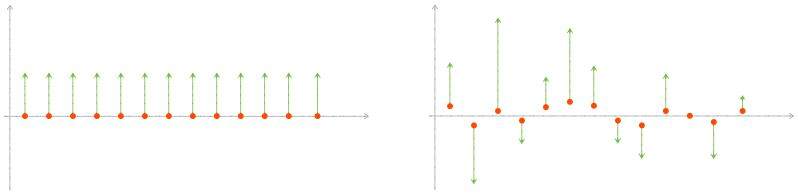


Figure 5.4: Consider a network with 13 neurons in the first layer. Each neuron have the same input and their outputs are also treated in the same way. Consider the weight $w_{k,1}$ that is multiplied with the first input x_1 for each of the neurons, $k = 1, \ldots, 13$. If all weights are initialized to zero then the derivatives with respect to the $w_{k,1}$'s will all be identical (left) and our gradient step will change all $w_{k,1}$'s in the same way. Choosing the weights as small random numbers instead will promote variation. (right)

Computing the gradients

The second thing we need to perform stochastic gradient descent is a way to evaluate the gradients. Recall that the negative log-likelihood is a sum

$$L(\theta) = \sum_i L_i(\theta), \tag{5.1}$$

where L_i is the partial loss due to one single training example and θ are the parameters that we are trying to learn, in this case the weights w_k. To perform stochastic gradient descent, we need an algorithm to compute ∇L_i. In our case the only parameters are weights w_k, so if we can compute

$$\frac{\partial L_i}{\partial w_k} \tag{5.2}$$

for an arbitrary w_k, we are done. Let's start with a really simple example.

5.3.1 A minimal example

The network in Figure 5.5 consists of a single input number x, that is multiplied with a weight w and fed into a logistic sigmoid function

$$p = \xi(y) = \frac{e^y}{1 + e^y}, \tag{5.3}$$

to produce a probability, p. Finally, this probability is used to compute the negative log-likelihood loss. For a positive training example the partial loss is

$$L_i = -\ln p \tag{5.4}$$

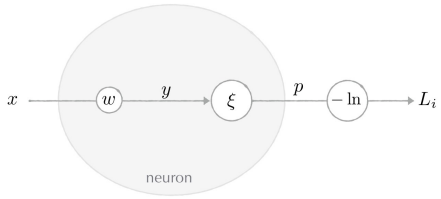


Figure 5.5: A very simple network.

To use stochastic gradient descent, we need the gradient of the partial loss with respect to the network parameters. In this case there is only one parameter, w, so there is only one derivative to compute. Using the chain rule, we get

$$\frac{\partial L_i}{\partial w} = \frac{\partial L_i}{\partial p}\frac{\partial p}{\partial y}\frac{\partial y}{\partial w} \tag{5.5}$$

We can assume that we have precomputed p, y and L_i for the current input. Let's start from the left in (5.5),

$$\frac{\partial L_i}{\partial p} = \frac{\partial}{\partial p}\left(-\ln\left(p\right)\right) = -\frac{1}{p}, \tag{5.6}$$

$$\frac{\partial p}{\partial y} = \xi' = \frac{e^y(1+e^y) - (e^y)^2}{(1+e^y)^2} = \frac{e^y}{(1+e^y)^2} = p(1-p), \tag{5.7}$$

and

$$\frac{\partial y}{\partial w} = x. \tag{5.8}$$

Hence, we can write the sought derivative

$$\frac{\partial L_i}{\partial w} = (p-1)x. \tag{5.9}$$

In a similar manner we can compute the derivative of the partial loss for a negative training example,

$$\frac{\partial L_i}{\partial w} = px. \tag{5.10}$$

As you can see, all the derivatives have really simple formulas, (if we assume that we know all the intermediate variables in the net). Naturally, this is no coincidence. The functions used in neural nets have been chosen carefully to allow for efficient computations.

There are a few more things to note from these derivatives. For example, if $p \approx 1$ for a positive training example, we get a very small derivative. In other words, if we already have correct classification, we don't need to change anything. The same thing happens if $p \approx 0$ for a negative training example.

5.3.2 Backpropagation

To understand how to compute gradients in a general neural network it will be sufficient to consider a single weight w_k of a neuron in the middle of the network. Figure 5.6 shows the neuron's place in the full neural net and Figure 5.7 takes a closer look at the neuron, defining the involved parameters.

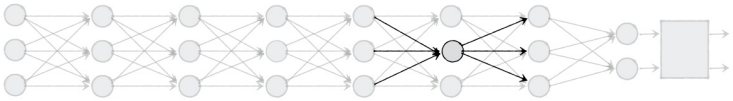


Figure 5.6: To understand backpropagation it is sufficient to analyze one neuron as highlighted in this neural net.

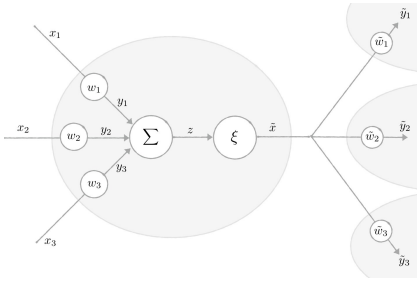


Figure 5.7: One neuron in a general neural network. The argument is easily generalized to deal with an arbitrary number of inputs and outputs.

First note that

$$\frac{\partial L_i}{\partial w_k} = \frac{\partial L_i}{\partial y_k} x_k. \tag{5.11}$$

So if we can compute $\frac{\partial L_i}{\partial y_k}$ for all y's in the net, we can get the desired gradient (w.r.t. the w's). As Figure 5.7 shows, the output from the neuron is passed on to multiple neurons in the next layer, so we need to follow all these paths. More precisely,

$$\frac{\partial L_i}{\partial y_k} = \sum_j \frac{\partial L_i}{\partial \tilde{y}_j} \frac{\partial \tilde{y}_j}{\partial y_k} \qquad \text{and} \qquad \frac{\partial \tilde{y}_j}{\partial y_k} = \frac{\partial \tilde{y}_j}{\partial \tilde{x}} \frac{\partial \tilde{x}}{\partial z} \frac{\partial z}{\partial y_k} = \tilde{w}_j \xi'. \tag{5.12}$$

Hence,

$$\frac{\partial L_i}{\partial y_k} = \xi' \sum_j \tilde{w}_j \frac{\partial L_i}{\partial \tilde{y}_j}. \tag{5.13}$$

This doesn't look spectacular, but it means that the derivatives in one layer depends in a very simple way on the derivatives in the next layer. Hence we

can start from the last layer, compute

$$\frac{\partial L_i}{\partial y_k} \tag{5.14}$$

for all the y_k's in that layer and then proceed backwards through the network. Having the derivatives with respect to y_k we use to (5.11) to get the desired derivatives with respect to the weights w_k, that is, with respect to the parameters that we wish to learn. Note that the formulas frequently use intermediate values of the network, so before doing backpropagation, we need to do a forward pass to compute these intermediates.

5.4 Translation invariance

General neural networks contain a huge number of weights and training them turned out to be very difficult. Hence a number of researchers started advocating a more restricted family of networks. Their argument starts with the fact that in computer vision, we want classifiers to be translation invariant. Let us see where this argument takes us.

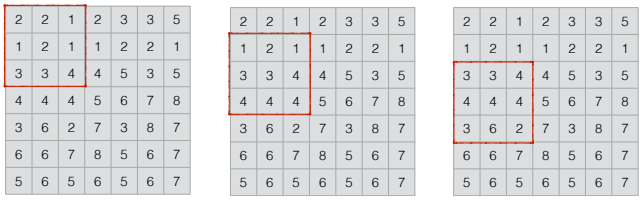


Figure 5.8: The input of the neurons in (5.17), (5.18) and (5.19), respectively.

$$\begin{aligned} &a\,I(3,1) + b\,I(4,1) + \\ &c\,I(3,2) + d\,I(4,2) \end{aligned} \tag{5.15}$$

(5.16)

In the first layer of our network, the neurons take input directly from the image. Let's say that there is a neuron that computes

$$\begin{aligned} &w_1\,I(1,1) + w_2\,I(2,1) + w_3\,I(3,1) \\ &+w_4\,I(1,2) + w_5\,I(2,2) + w_6\,I(3,2) \\ &+w_7\,I(1,3) + w_8\,I(2,3) + w_9\,I(3,3) \end{aligned} \tag{5.17}$$

and that all other weights are zero for this neuron. If we expect our network to be fully translation invariant, we want identical neurons at every possible

position in the image; cf. Figure 5.8. One neuron that computes

$$\begin{aligned} & w_1\, I(1,2) + w_2\, I(2,2) + w_3\, I(3,2) \\ & + w_4\, I(1,3) + w_5\, I(2,3) + w_6\, I(3,3) \\ & + w_7\, I(1,4) + w_8\, I(2,4) + w_9\, I(3,4) \end{aligned} \tag{5.18}$$

and one that computes

$$\begin{aligned} & w_1\, I(1,3) + w_2\, I(2,3) + w_3\, I(3,3) \\ & + w_4\, I(1,4) + w_5\, I(2,4) + w_6\, I(3,4) \\ & + w_7\, I(1,5) + w_8\, I(2,5) + w_9\, I(3,5) \end{aligned} \tag{5.19}$$

etcetera. We get an array of neurons, with one neuron per pixel, as in Figure 5.9. All these neurons have identical weights, but take their inputs from different patches in the image. This is simply an implementation of a filter. Each neuron

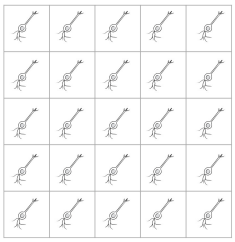


Figure 5.9: To achieve translation invariance we need an array of neurons which all have the same weights. These neurons implement a filter.

computes the dot product of the filter

$$w = \begin{pmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{pmatrix} \tag{5.20}$$

and a patch in the image. Rather than learning the weights of individual neurons, we learn a set of filters. The same argument holds for the later layers of the network. The second layer should perform filtering on the output of the first layer and the third layer should perform filtering on the output of second layer and so forth.

The pursuit of this idea leads to a type of networks called convolutional neural networks. They have a drastically reduced set of parameters compared to the classical, fully-connected nets, while enforcing a desirable characteristic of the obtained classifiers.

5.5 A convolutional neural network

Figure 5.10 shows an example of a convolutional neural network. It is getting too messy to draw individual neurons so we have to make due with a

schematic view. On the far left we have the input image. The first layer consists of 30 5×5-filters and a RELU-nonlinearity. There is also a bias term, but that is often left out in a schematic view of the network or counted as part of the nonlinearity. Each of the 30 filters are applied on the image and produces a response map with the same size. Hence the output from the first layer consists of 30 channels of size 64×64. You can think of them as similar to the RGB channels of a colour image . Each channel has the same set of pixels (spatially) but they tell us different things about these pixels.

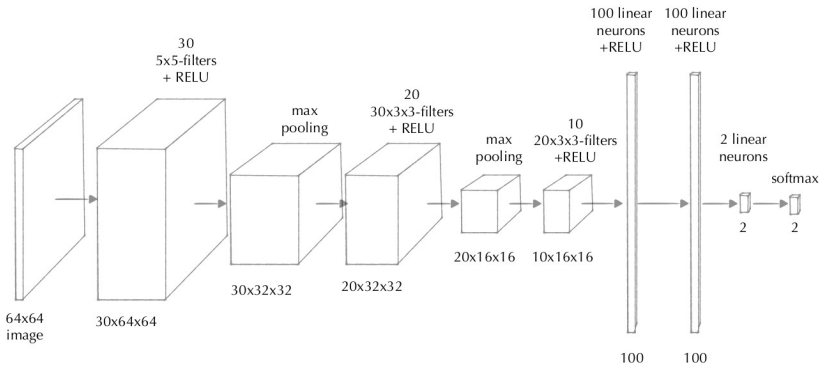


Figure 5.10: Example of convolutional neural network.

The next layer performs an operation called max pooling. Each of the 30 response maps is divided into 2×2 regions and from each region, we only keep the maximum value; see Figure 5.11. This means that the output from this layer still has 30 channels corresponding to the 30 filters, but the spatial dimensions has been reduced to 32×32.

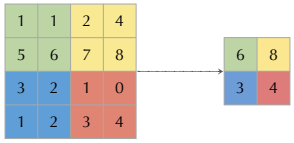


Figure 5.11: Max pooling over 2×2-regions.

The third layer is another convolutional layer but with 20 filters. The spatial dimensions of these filters is 3×3, but note that each filter works with all the 30 available channels, so each filter, f, consists of of 30 3×3-filters, f_j working on the 30 channels. If $X_j(x, y)$ is the jth channel in the output from the previous layer, then the filter response is

$$\sum_{j=1}^{30} X_j \star f_j. \tag{5.21}$$

This also means that each filter has $30 \times 3 \times 3$ parameters.

Layers four and five look familiar by now, but the sixth layer is different. Here we drop the convolutional structure in favour of a classical *fully-connected* layer. This means that each of the 128 neurons in the sixth layer computes a linear combination of all $20 \times 16 \times 16$ outputs from the previous layer. The fully-connected layers are used to reach a final decision. Is it a cell? Finally, a softmax function is used to transform the network outputs to probabilities.

5.6 Overfitting

In the previous chapter a few general techniques to avoid overfitting were discussed. Naturally, the very best thing we can do is to somehow get more data.

Data augmentation

This is also the logic behind *data augmentation*, where we use the data we have together with our knowledge of the problem to create new semi-synthetic data.

Transfer learning

Transfer learning is based on a similar idea. If we cannot get a large dataset for the problem at hand, we simply train our network to perform a different task—where there is abundance of data. If, for example, we want segment cars, we can start with training it on general general recognition using the extensive ImageNet dataset. The idea is that filters and structures which are good to solve one image-related task are likely to work well for other tasks as well.

Regularization

One view of overfitting is that we design specialized *rules* for individual examples in the training set. One way to punish this is by adding an auxilliary loss that puts a penalty on the sum of squared weights. We simple alter the loss by adding a term

$$L(\theta) + c \sum_{i=1}^{n} w_i^2, \tag{5.22}$$

where c is a small positive constant. Note that you normally do not put a penalty on the constant or bias terms, w_0. The intuition is that the extra loss will prevent changing a few weights to improve our performance on a single example in the training set.

Dropout

A recent and elegant method to deal with overfitting, that is tailored for neural networks, is something called dropout. The idea is illustrated in Figure 5.12.

At each iteration of stochastic gradient descent, a portion of the neurons are *removed*. Applying the network in practice, we use all neurons, but weight their outputs to compensate for the dropout.

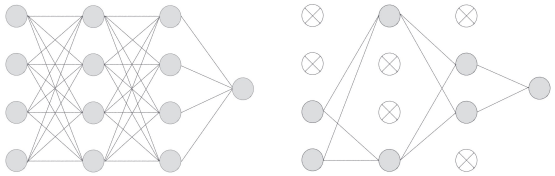


Figure 5.12: Dropout.

The idea is to learn multiple ways to recognize an object at the same time. Let us assume that in our training sets all cats are facing the camera. At early stages in the training a cat face detector has evolved. Recall that the gradient of the partial loss, ∇L_i, will contain a factor $(1 - p)$, where p is the output probability for the correct class. If the cat face detector makes us certain about the correct class ($p \approx 1$), we will essentially ignore the cat examples from now on. This means that we won't learn alternative ways to detect cats that could have been useful to detect cats which are not facing the camera. But if we use dropout, at some iterations, the cat face neurons will be removed. Then p_{cat} is small and the network is forced to re-learn how to detect cats. With a little luck it will learn a different way this time—and possibly one that generalizes better.

5.7 Networks for regression

Convolutional neural networks can be used for other tasks than classification. Figure 5.13 shows an example. Here a network has been used to determine centre and radius for a cell. The only thing that has to be changed in order to learn this type of measurements is the loss function. The negative log likelihood is not appropriate for outputs other than probabilities. In this case, the squared residual was used instead. Similar approaches are used in state-of-the-art object detection to predict rectangular object bounding boxes.

Figure 5.13: Cells with detected centres and radii.

Chapter 6

Fully-convolutional networks

In this chapter we will look at some principles for designing networks for segmentation, detection and regression problems. Training and applying a convolutional neural networks requires a massive amount of computations, so we cannot overlook the question of computational complexity when we design a network.

6.1 Computational complexity

Naturally the exact execution times for different layers depend on both hardware and software. Still there are a few general principles which are good to keep in mind. Let's first consider a convolutional layer with 5×5-convolutions. If the input to this layer has 30 channels, then the actual filtering operation is

$$w_0 + (I \star w)(x, y) = w_0 + \sum_{c=1}^{30} \sum_{i=-2}^{2} \sum_{j=-2}^{2} I_c(x + i, y + j) w_c(i, j), \tag{6.1}$$

so in the normal case all channels are used as input and 5×5 filter actually has

$$5 \times 5 \times 30 + 1 = 751 \tag{6.2}$$

trainable parameters. To apply this filter at a single position, we need to perform roughly 750 multiplications and 751 additions. For our needs it is fully sufficient to state that the time consumption is roughly $c\,750$. To apply this filter at every position in an image with N_{pix} pixels, requires $c\,750 N_{pix}$ operations.

6.2 Fully-convolutional networks

Strictly speaking a fully-convolutional network is simply a convolutional neural network that does not contain any fully-connected layers. One advantage of

this is that fully-convolutional networks can be applied to input images of different sizes. It also means that (depending on the input size) the output from a fully-convolutional network can be an image sometimes even of the same size as the input. Figure 6.1 shows the structure of a very simple fully-convolutional network with a few convolutional layers with ReLU nonlinearities. If we use padded convolutions the output from such network has the same shape as the input image. Thus it is very natural to train this type of networks on image segmentation problems. A classical network that has a fully-connected layer at the end, would have to be applied individually for every pixel to create a segmentation, whereas a fully-convolutional network is applied once to the full image. This is normally much more efficient.

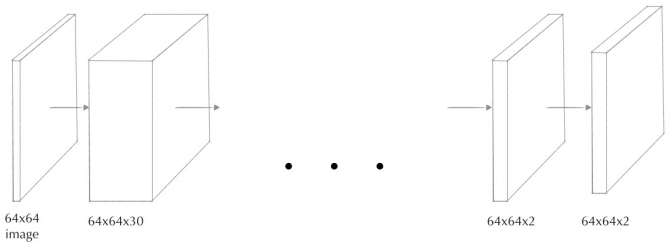


Figure 6.1: In segmentation problems we want an output probability for each pixel in the input image. Fully-convolutional networks are much more efficient at that as they can be applied simultaneously to the whole input image.

6.2.1 Fully-convolutional networks for detection

In Chapter 4, we discussed how to train a linear classifier for cell detection. Naturally, we could train a CNN instead and apply it in a sliding window manner to detect cells in an image. It is much more efficient however, to use a fully-convolutional network that can be applied to the full input image at once. The goal is to train a network that produces an output with a positive blob at every true detection and zeros anywhere else. Figure 6.2 shows an example.

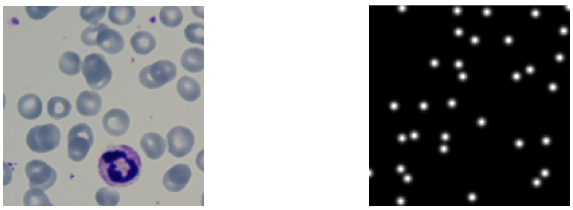


Figure 6.2: Input and desired output for a network used for detection. We want a positive blob at each cell centre, ideally with a unique local maximum at the centre.

Recall from Section 6.1 that the complexity of applying a convolutional layer is heavily dependent on the spatial resolution of the input. This makes the network in Figure 6.1 very slow as the spatial resolution is the same throughout the network. (But still much faster than running a classical CNN for each pixel individually.) For detection problems we can often accept that the output has a slightly lower resolution than the input. Figure 6.3 shows a possible network structure. By using one or two max-pooling layers early in the network to reduce the spatial resolution, we get a much faster network.

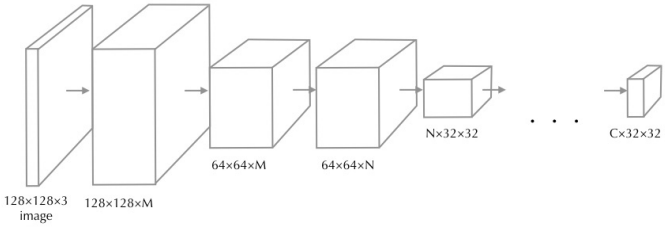


Figure 6.3: A fully-convolutional networks for detection. The output is a 2D image just as the input albeit with a lower resolution than the input. Reducing the resolution gives less precise coordinates for the detected objects, but at a much lower computational cost.

6.3 Variants of filtering

Standard linear filtering is by far the most common variant, but in neural networks, there are a number of useful variants. The simplest one is called strided filtering. Note that in conjunction with neural networks, filtering is often called convolution (even though the filter is not flipped as in classical convolutions). I will ignore this habit and refer to filtering as filtering.

6.3.1 Strided filtering

One way to view filtering is as a sliding window dot product. At each step we move the filter one pixel and compute a new dot product. This produces an output of the same resolution as the input. If we move the filter k pixels instead we get strided filtering with a stride of k. The effect is identical to filtering followed by downsampling, but more efficient. Figure 6.4 illustrates strided filtering with stride 2.

6.3.2 Filtering as matrix multiplication

Consider filtering an $m \times m$-image I with a filter f, producing a

$$J = I \star w. \tag{6.3}$$

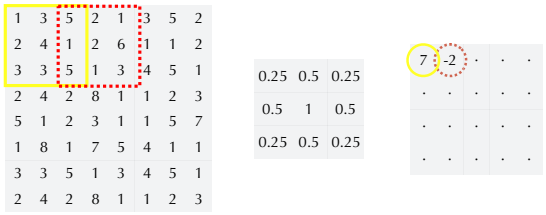


Figure 6.4: Principle for strided filtering.

Each value in a J is a linear combination of pixel values in the input. If we stack all the pixel values in vectors $\mathrm{vec}(J)$ and $\mathrm{vec}(I)$ we can describe the operation with a matrix product,

$$\mathrm{vec}(J) = F\,\mathrm{vec}(I), \tag{6.4}$$

where F is an $m^2 \times m^2$ matrix. This representation might seem rather inefficient, but it will be useful as an analytical tool and in fact, a similar (but slightly more economical) matrix representation is used to perform super-efficient filtering on modern graphics cards.

The matrix representation starts to pay off if we consider backpropagation. Recall the setup for backpropagation: We are moving backwards one layer at the time. At some point we have all derivatives

$$\frac{\partial L}{\partial y_1}, \frac{\partial L}{\partial y_2}, \ldots, \frac{\partial L}{\partial y_n}, \tag{6.5}$$

where $n = m^2$, and want to estimate the derivatives with respect to x_k's further back in the layer. Assuming that x and y are separated by a convolutional layer, we have $y = x \star f$. Using the matrix representation the same relationship is described by

$$\mathrm{vec}(y) = F\mathrm{vec}(x). \tag{6.6}$$

To see how the derivatives are related, we consider

$$\frac{\partial L}{\partial x_1} = F_{11}\frac{\partial L}{\partial y_1} + F_{21}\frac{\partial L}{\partial y_2} + \ldots + F_{n1}\frac{\partial L}{\partial y_n}. \tag{6.7}$$

Similar formulas for the other x_k's yield

$$\nabla_x L = F^T \nabla_y L, \tag{6.8}$$

so with this notation we can do backpropagation by simply transposing the filtering matrix!

6.3.3 Transposed filtering

In many cases, it makes sense with a network layer that reduces the resolution of the input. We have already seen one example with the pooling layers.

But there are also cases when we want to go from a low-resolution input to a higher-resolution output.

So what is a systematic way to do this? Naturally we could interpolate the input image to a higher resolution and then continue using normal filtering, but this would be a waste of both memory and computational power. Why not learn specific interpolation parameters which work for the specific problem method. One way to do this is using transposed filtering, sometimes called fractionally-strided filtering.

Essentially transposed filtering is strided filtering run backwards. Consider strided filtering of high-resolution image x, with N pixels, to a low-resolution image, y, with M pixels, where naturally, $M < N$. This can be described as

$$\text{vec}(y) = F\text{vec}(x), \tag{6.9}$$

where F is an $M \times N$-matrix. If we want to go the other way, from a low resolution input to a high resolution output, we keep the same nonzero entries but apply $F^{\boldsymbol{T}}$ instead. So assuming that the input x has M pixels and the output y has N pixels we can define transposed filtering as

$$\text{vec}(y) = F^{\boldsymbol{T}}\text{vec}(x). \tag{6.10}$$

In the popular software packages, transposed filtering is often defined in this way. You set up a strided filtering and then tell the software to use it backwards instead.

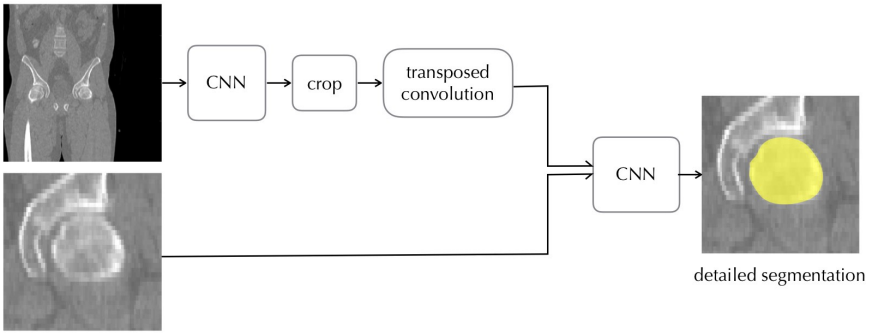


Figure 6.5: Transposed filtering can be used to combine low-resolution data with fine-grained high-resolution data. In this example we want to segment the femur. A low-resolution image with large field of view is used to determine the rough position in the body. To get a detailed high-resolution segmentation we need to combine this with a high-resolution zoomed-in image. This is done via cropping and transposed filtering.

Part II

Geometric Models

Chapter 7

Robust model fitting

For the next few chapters, we will focus on what we can do with the kind of measurements that we get from SIFT-like feature matching. The output of feature matching is normally a set of correspondences between two images. These correspondences were established by finding points with similar descriptors, but once we have found such pairs, we can forget about the descriptors and about the image. All we need are the coordinates of the corresponding points. Hence a correspondence pair is a four-dimensional measurement $(x, y, \tilde{x}, \tilde{y})$. We will soon start working with this type of data but let us start with something even simpler: Line fitting.

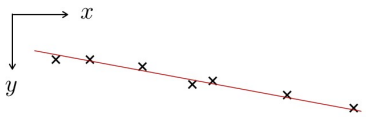


Figure 7.1: Measured points (x's) that fit well with a linear model.

Figure 7.1 shows a number of points in 2D. The points will be points in an image, so we refer to them as (x_i, y_i), and draw the y-axis pointing down. These are our noisy measurements. The model is that these points lie on a line (drawn in red). A line is given by an equation

$$ax + by + c = 0. \tag{7.1}$$

This is our model and the process of *model fitting* is the process of estimating a, b and c such that the model fits well with our measurements.

As soon as we have more than two measured points, there is normally not a line that fits perfectly to all the measurements. In other words (7.1) will not be satisfied if we plug in our measurements. To measure the deviation we introduce the concept of residuals. The residual is the deviation from the current model. Exactly what we mean by this will vary depending on the application.

Figure 7.2 shows the definition for line fitting. The ith residual, r_i is the orthogonal deviation of measurement number i. It is not always easy to find a formula for the residual. For line fitting a little linear algebra shows that

$$r_i(a, b, c) = \frac{ax_i + by_i + c}{a^2 + b^2} \begin{pmatrix} a \\ b \end{pmatrix}. \tag{7.2}$$

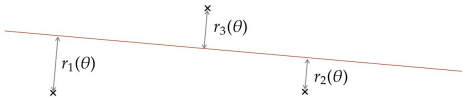


Figure 7.2: For line fitting, we define residual vectors, $r_i(\theta)$ as the orthogonal deviation.

7.1 Least squares

The residual concept gives us a way to measure how well a set of parameter values fits to a measurement, but we get one residual for each measurement. What is the best way to combine these to a single measure? We start with a one-dimensional example.

Example 2. *Let's say that we asked ten people when Olof Palme was murdered. We haven't looked at the answers yet but want to come up with a rule to estimate the actual year.*

It might seem somewhat contrived to talk about a model in this case, but try to accept it. Let y_i be the answer from the ith person. The model is that all the answers are noisy measurements of the true year θ,

$$x_i = \theta + e_i. \tag{7.3}$$

Given an θ, we compute the residuals as

$$r_i(\theta) = x_i - \theta. \tag{7.4}$$

If the e_i's are Gaussian, the maximum likelihood estimate is the one minimizing the sum of squared residuals, that is, the ML estimate is equal to the

least squares estimate, being the solution to

$$\min_\theta \sum_{i=1}^n r_i^2(\theta). \tag{7.5}$$

Let us look closer at this loss function

$$L(\theta) = \sum_{i=1}^n r_i^2(\theta) = \sum_{i=1}^n (x_i - \theta)^2 = \sum_{i=1}^n \left(x_i^2 - 2\theta x_i + \theta^2\right) = \sum_{i=1}^n x_i^2 - 2\theta \sum_{i=1}^n x_i + n\theta^2. \tag{7.6}$$

Since all the x_i's are known, this is just a quadratic function of θ. We know that these look like a very happy mouth. To find the θ that minimizes the loss we differentiate

$$L'(\theta) = -2\sum_{i=1}^n x_i + 2n\theta, \tag{7.7}$$

and solve for the stationary point by setting $L'(\theta) = 0$. We get

$$\hat{\theta} = \frac{1}{n}\sum_{i=1}^n x_i, \tag{7.8}$$

i.e., the ML estimate is just the mean of the measurements.

Example 3. *Let's see what this does in the Palme example. As shown above we assume that nine people answered* 1986 *and one answered* 1918*. In fact, the latter answer is not altogether wrong. The historian Olof Palme was killed in 1918 in the Finnish civil war. In fact, it is often the case that the outlier data is not incorrect, but just not what we were expecting. To return to the question at hand, the mean, and hence the least squares estimate for this problem, is* 1979.2*.*

Before moving on, we will just mention that the least squares estimate is often referred to as the ℓ^2 estimate since it is equivalent to minimizing the ℓ^2-norm of the vector that contains all residuals

$$\bar{r} = \begin{pmatrix} r_1(\theta) \\ \vdots \\ r_n(\theta) \end{pmatrix}. \tag{7.9}$$

7.2 Least absolute residuals (ℓ^1) $\star$

Roughly speaking, an outlier is a measurement that deviates greatly from the *correct* model. The problem with least squares is that it is very sensitive to so called outliers, and the problem with image analysis is that outliers are everywhere. Researchers working in the field *robust statistics* have tried to find other loss functions which are less sensitive to large errors.

One such loss function is the sum of absolute residuals,

$$\sum_{i=1}^{n} |r_i(\theta)| = \sum_{i=1}^{n} |x_i - \theta|. \tag{7.10}$$

Let us try this on our one-dimensional estimation. It turns out to be a little harder to derive the solution in this case. First of all note that we have a finite sum of continuous functions so the sum is also continuous. Now use this trick

$$L(\theta) = \sum_{i=1}^{n} |x_i - \theta| = \sum_{x_i > \theta} (x_i - \theta) + \sum_{x_i < \theta} (\theta - x_i). \tag{7.11}$$

This shows that as long as we stay away from the x_i, L is linear and we can differentiate.

$$L'(\theta) = \sum_{x_i > \theta} (-1) + \sum_{x_i < \theta} 1. \tag{7.12}$$

If the number of $x_i > \theta$ is larger than the number of $x_i < \theta$ then we get a negative derivative. Hence we get a negative derivative for small θ. At each x_i the derivative increases; see Figure 7.3. The minimal loss is attained when as many $x_i > \theta$ as $x_i < \theta$, i.e., at

$$\hat{\theta} = \text{median}\{x_1, \ldots, x_n\}. \tag{7.13}$$

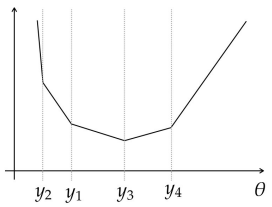


Figure 7.3: Loss function for least absolute residuals. If the number of measurements is even, there might be a perfectly flat part and the minimizer is not unique.

Least absolute residuals is often referred to as ℓ^1 as it is equivalent to minimizing the ℓ^1-norm of the residual vector in (7.9).

Example 4. *Returning to the Palme example we see that the median of the ten measurements is* 1986. *In fact as long as more than half of the measurements are correct, we will get the correct answer. But if we ask people from around the world we might get even more incorrect answers. Let's say we ask 1000 people from around the world. They all understand that we haven't had a prime minister for that long so the incorrect answers are spread between* 1850 *and* 2016. *Apart from that* 10% *actually know the answer and say* 1986. *What happens? Typically the median is around* 1940.

7.3 Outlier count (ℓ^0)

To deal with the type of data mentioned in Example 4, we need an even more robust loss function than the sum of absolute residuals. We introduce the notion of an outlier, being a measurement such that

$$|r_i(\theta)| > \tau \tag{7.14}$$

for some fixed threshold τ. The next loss function that we consider is the number of outliers, sometimes referred to the ℓ^0 loss.

Example 5. *In the Palme example, a reasonable value for the outlier threshold is something like* 0.5*. Hence a residual is an outlier to a certain* θ *if* $|r_i(\theta)| > 0.5$*. Using this loss function, it is very unlikely that another year will have as many votes as* 1986*.*

7.4 Ransac

An issue with using the outlier count as loss is that it are very hard to minimize. Figure 7.4 shows the number of outliers as a function of θ in a one-dimensional estimation problem similar to the Palme example. As you can see it is a non-continuous function with multiple local minima.

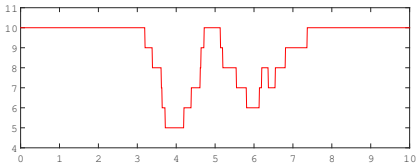


Figure 7.4: The outlier count is very hard to optimize as it has many local minima.

To handle this difficult type of estimation problems, we will use a method called random sample consensus, or Ransac in short. It is an iterative method. In each iteration the following steps are performed:

1. Randomly select a subset with K of the measurements.
2. Fit model parameters θ_{hyp} to this subset.
3. Compute the loss $L(\theta_{hyp})$ using the residuals of *all* the measurements.

These steps are repeated (typically thousands of times) while keeping track of the parameters that yielded the lowest loss. The idea is that, sooner or later, the random subset will only contain good measurements. Figure 7.5 illustrates one Ransac iteration for line fitting.

The hard part in applying Ransac to a new problem is normally to work out a solver for Step 2 in this list. Fitting a line to two points is easy enough, but estimating the camera motion from the two images in Figure 3.12 is something much harder. In that case we need to pick at least five points and the exact estimation requires solving a 10th-degree polynomial.

(a) (b)

Figure 7.5: A Ransac iteration for line fitting. (a) Two points are selected randomly and a line that fits perfectly to these points is estimated. (b) Residuals are computed for all points and the loss is computed.

Performance analysis

Let us examine how the performance of Ransac depends on number of randomly selected points, K, and the rate of outliers in the data. Let p be the probability of picking an outlier when randomly selecting a measurement. If for example 90% of the measurements are outliers, then $p = 0.9$.

To get a good model in Step 2, we need the set of K selected measurements to be outlier-free. The probability of this happening is $(1-p)^K$. For $p = 0.9$ the probability of getting a good solution decreases rapidly with K. Clearly we should select the smallest K that still makes it possible to estimate the model parameters. This explains why solvers for Step 2, are often called *minimal solvers*.

Another question is how to choose the number of Ransac iterations. Let's look at the line fitting example again. For a line the minimal number of points is $K = 2$ as it is not possible to estimate at line from a single measurement. Assuming that the probability of picking an outlier is $p = 0.9$, we have

$$P(\text{outlier-free subset}) = (1 - 0.9)^2 = 0.01. \tag{7.15}$$

So on average we need 100 Ransac iterations to get one outlier-free subset and one reasonable set of parameters! Also note that, in general, an outlier-free subset does not imply that we get a good solution, as the inliers are also affected by noise. Hence we should choose the number of iteration significantly larger than 100. As a rule of thumb, you could use

$$\text{number of iterations} \approx \frac{100}{P(\text{outlier-free subset})} \tag{7.16}$$

7.5 Huber loss*

The Huber loss is very similar to the sum of absolute residuals, but as it is differentiable everywhere it it easier to work with. The idea is to use squared residuals if they are small, say less than δ and then switch to something that increases linearly for large residuals, see Figure 7.6. The Huber loss given a set

of residuals is

$$L(\theta) = \sum_{i=1}^{n} h(r_i^2(\theta)) \tag{7.17}$$

where

$$h(s) = \begin{cases} s & \text{if } s < \delta \\ 2\delta\sqrt{s} - \delta^2, & \text{otherwise.} \end{cases} \tag{7.18}$$

The offset and slope of the second part are carefully chosen such that h is differentiable even at $s = \delta^2$.

Example 6. *Figure 7.7 shows the Huber loss for θ's between 1984 and 1988 (and $\delta = 0.5$) for the Palme example. The lowest loss is obtained at 1985.9 so very close to the correct answer. If we include answers from around the world, the Huber loss fails just as badly as the ℓ^1 loss.*

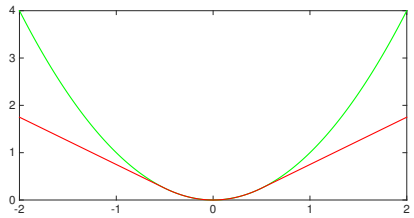


Figure 7.6: The Huber loss for for one residual (red) compared to the squared loss (green).

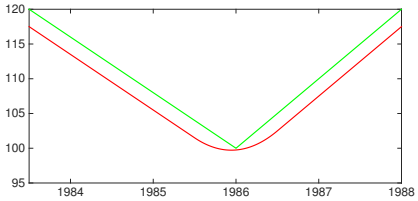


Figure 7.7: Palme example: The Huber loss with $\delta = 0.5$ (red) and the ℓ^1 loss (green) as a function of θ. The lowest Huber loss is attained at 1985.9.

7.6 Iteratively reweighted least squares ⋆

Intuitively the problem with least squares is that too much weight is assigned to a few of the measurements. In many cases the weighted least squares problem,

$$\sum_{i=1}^{n} w_i r_i^2(\theta) \tag{7.19}$$

has a simple solution. For example, the one-dimensional Palme example, yields the solution

$$\theta \sum_{i=1}^{n} w_i = \sum_{i=1}^{n} w_i x_i, \tag{7.20}$$

i.e., a weighted average of the measurements. Iterative reweighting means that we update the weights at each iteration in order to minimize a different loss function than the sum of squared residuals. Let's say we want to minimize the Huber loss

$$L_h(\theta) = \sum_{i=1}^{n} h(r_i^2(\theta)). \tag{7.21}$$

To get the same minima, we want to define a reweighting scheme such that

$$L_w(\theta) = \sum_{i=1}^{n} w_i r_i^2(\theta) \tag{7.22}$$

and $L_h(\theta)$ has the same derivatives. To simplify notation we introduce $f_i(\theta) = r_i^2(\theta)$.

$$\nabla L_h(\theta) = \sum_{i=1}^{n} h'(f_i(\theta))\, \nabla f_i(\theta). \tag{7.23}$$

and

$$\nabla L_w(\theta) = \sum_{i=1}^{n} w_i\, \nabla f_i(\theta) \tag{7.24}$$

We can achieve this, by setting

$$w_i = h'(f_i(\theta)). \tag{7.25}$$

Note that this is done iteratively. For each estimate $\theta^{(k)}$, we compute a new set of weights and solve a new weighted least squares problem. For the Huber loss, the weighting scheme is

$$h'(s) = \begin{cases} 1 & \text{if } s < \delta^2 \\ \frac{\delta}{\sqrt{s}}, & \text{otherwise.} \end{cases} \tag{7.26}$$

Example 7. *We return to the Palme example (with $\delta = 0.5$) and assume that we start in the least squares solution, ≈ 1979. Selecting weights according to* (7.26) *are $0.5/7$ for the correct answers and $0.5/61$ for the incorrect answer. This leads to a new estimate of 1985.1. The new weights are roughly 0.58 for the correct measurements and 0.007 for the incorrect leading to a new estimate of 1985.6. Now we get weight 1 for the correct measurements and 0.007 for the incorrect and an estimate of 1985.9.*

Chapter 8

Image registration

8.1 Coordinate transformations and warping

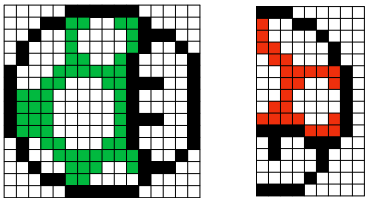


Figure 8.1: Example of a source and a target image. Our goal is to create a warped variant of the source image with the same size as the target image.

Let us say that we want to align one image, called the source, or I_s to another image, called the target image, or I_t; cf. Figure 8.1. The purpose of this is normally to relate information in the two images or use a well-known image to tell us something about a new, unknown image. Thus it is very practical if the warped image, I_w, has the same size as the target image, so we start by forming an *empty* image of the same size as the target. What we need now is a rule for how to fill this image with pixel intensities from the source. This rule could be something like: Set $I_w(x, y)$ to the value of $I_s(\tilde{x}, \tilde{y})$, where

$$\begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} = \mathcal{T} \begin{pmatrix} x \\ y \end{pmatrix}. \tag{8.1}$$

Note that this coordinate transformation might feel backwards. To warp I_s to I_t we need a transformation mapping coordinates in I_t to coordinates in I_s. What type of coordinate transformation to use, depends on the application. Figure 8.2 shows an example. In this case we have a coordinate transformation

that is a rotation and a translation, namely

$$\begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 9 \end{pmatrix}, \tag{8.2}$$

where $(\tilde{x}, \tilde{y})$ refers to coordinates in the source image and (x, y) refers to coordinates in the target image. In the figure, we have filled the first column with pixel values from the source image and are working our way down the second column. To determine the value of the current pixel, being the pixel at $(2, 5)$. We use the coordinate transformation,

$$\begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} = R \begin{pmatrix} 2 \\ 5 \end{pmatrix} + t = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} 2 \\ 5 \end{pmatrix} + \begin{pmatrix} 0 \\ 9 \end{pmatrix} = \begin{pmatrix} 5 \\ 7 \end{pmatrix}. \tag{8.3}$$

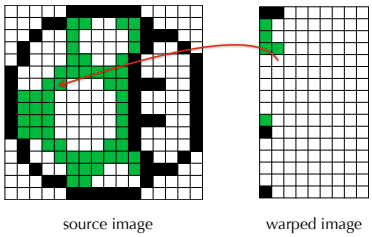


Figure 8.2: Warping: For each pixel, we use the coordinate transformation to find the coordinates in the source image at which to collect the colour information. In this case we have finished with the first column and just started filling the second column of the warped image.

8.2 Transformation types

Rigid transformation. For images taken under very controlled circumstances, for example microscopy with fixed settings, a rigid transformation might be suitable. This allows rotating and translating the source image, but not changing the scale. If $(x \quad y)^T$ are coordinates in the target image and $(\tilde{x} \quad \tilde{y})^T$ coordinates in the source image we have the relationship

$$\begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} = \begin{pmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix} = R \begin{pmatrix} x \\ y \end{pmatrix} + t. \tag{8.4}$$

Similarity transformation. Next in line is the similarity transformation which is a rigid transformation plus scaling with a factor s.

$$\begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} = s \begin{pmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix} \tag{8.5}$$

Affine transformation. A less restrictive transformation is the affine which—apart from rotation and translation—allows stretching of the image in an arbitrary direction,

$$\begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix} = A \begin{pmatrix} x \\ y \end{pmatrix} + t. \tag{8.6}$$

Note that reflections of the image are also affine transformations but normally not desirable in image applications. To give an example of a useful affine transformation let's consider the registration of an x-ray of a tall person (210 cm) to that of a short person (140 cm). The following is a transformation from the *short coordinates* to the tall ones, i.e., exactly what we need for the warping

$$\begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} = \begin{pmatrix} 1.2 & 0 \\ 0 & 1.5 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}. \tag{8.7}$$

Here I assumed that the difference i *width* of the two persons was only 20%, whereas the increase in height from 140 cm to 210 cm is 50%. Note that the translation vector in this example was zero so it can be excluded.

8.3 Registration using Ransac

A minimal solver is the solver used in Ransac to fit a model to a minimal (or at least small) subset of the measurements. Let's consider the similarity transformation as an example. Dealing with sines and cosines in equations is a little tricky so we replace $s\cos\varphi$ with a and $s\sin\varphi$ with b. This does not alter the problem. For correspondence j we two equations

$$\begin{pmatrix} \tilde{x}_j \\ \tilde{y}_j \end{pmatrix} = A \begin{pmatrix} x_j \\ y_j \end{pmatrix} + t = \begin{pmatrix} a & -b \\ b & a \end{pmatrix} \begin{pmatrix} x_j \\ y_j \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix} \tag{8.8}$$

This is equivalent to two standard equations,

$$\tilde{x}_j = x_j\, a - y_j b + t_x \tag{8.9}$$

$$\tilde{y}_j = y_j\, a + x_j b + t_y \tag{8.10}$$

If we collect these equations for two different pairs of corresponding points, let's say number 3 and 7, we get the system

$$\underbrace{\begin{pmatrix} x_3 & -y_3 & 1 & 0 \\ y_3 & x_3 & 0 & 1 \\ x_7 & -y_7 & 1 & 0 \\ y_7 & x_7 & 0 & 1 \end{pmatrix}}_{M} \underbrace{\begin{pmatrix} a \\ b \\ t_x \\ t_y \end{pmatrix}}_{\theta} = \underbrace{\begin{pmatrix} \tilde{x}_3 \\ \tilde{y}_3 \\ \tilde{x}_7 \\ \tilde{y}_7 \end{pmatrix}}_{v} \tag{8.11}$$

As you can see we have as many equations as unknown, so this system is easy to solve. In MATLAB you just run `M\v`. For Lab 2 you will have to work out the same thing for an affine transformation

In the case of a rigid transformation there is no simple linear parameterization, i.e., there is no simple way to get rid of the sines and cosines. There are many tricks to deal with this specific case. A reasonably general technique is to write everything using polynomial equations. For example we can replace a $\sin\varphi$ and $\cos\varphi$ with a and b, if we add the extra constraint $a^2 + b^2 = 1$.

Residuals and loss

The second thing we need to use Ransac is a way to measure residuals. In case of registration, there is a very natural choice, simply set

$$r_i(\theta) = A \begin{pmatrix} x_j \\ y_j \end{pmatrix} + t - \begin{pmatrix} \tilde{x}_j \\ \tilde{y}_j \end{pmatrix}. \tag{8.12}$$

Figure 8.3 illustrates the idea. Note that the residual is a vector. In many cases we are only interested in the length of this vector, called the *absolute residual*. One example is when we define outliers. Correspondence i is an outlier if the absolute residual

$$|r_i(\theta)| > \tau. \tag{8.13}$$

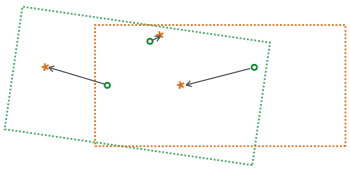


Figure 8.3: The residuals for three pairs of corresponding points.

8.4 Linear least squares

Having estimated an outlier-free solution with Ransac, we can remove the incorrect correspondences permanently from the problem and continue working with the inliers.

Consider the estimation of an affine transformation. Given a pair of corresponding points $(x, y)^T$ and $(\tilde{x}, \tilde{y})^T$, and a transformation as defined in (8.6), we can write the residual vector

$$r(\theta) = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix} - \begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} \tag{8.14}$$

where θ is a vector with the unknowns, a, b, t_x, c, d and t_y. An alternative way

to write this is

$$r(\theta) = \begin{pmatrix} x & y & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x & y & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ t_x \\ c \\ d \\ t_y \end{pmatrix} - \begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} \tag{8.15}$$

If we stack the residuals from n point correspondences into a long vector $\bar{r}$, we get

$$\bar{r}(\theta) = \begin{pmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ \vdots & & \vdots & & \vdots & \\ x_n & y_n & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_n & y_n & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ t_x \\ c \\ d \\ t_y \end{pmatrix} - \begin{pmatrix} \tilde{x}_1 \\ \tilde{y}_1 \\ \vdots \\ \tilde{x}_n \\ \tilde{y}_n \end{pmatrix} = M\theta - v \tag{8.16}$$

The matrix M here has $2n$ rows but only six columns, so we are dealing with an overdetermined system.

As you can see, the residuals are linear in the parameters. This is a nice special case, where it is actually cheap to estimate the least squares solution. Note that $\bar{r}^T\bar{r}$ is exactly the sum of squared residuals, and

$$\bar{r}^T\bar{r} = |M\theta - v|^2. \tag{8.17}$$

Writing `theta = M\v` in Matlab will produce the least squares solution to this problem.

8.4.1 A geometric interpretation ⋆

As (8.17) shows, finding the *best* affine transformation is equivalent to choosing θ such that the ordinary Euclidean distance between $M\theta$ and v is minimized! Let us look closer at this equation. Let m_i be the ith column of M. Then,

$$\bar{r} = m_1\theta_1 + m_2\theta_2 + \ldots + m_6\theta_6 - v. \tag{8.18}$$

Figure 8.4 shows a geometric interpretation. We have a set of $2n$-dimensional m_i's. In our case there are six of them, one for each θ_i. These span a six-dimensional subspace marked in grey in the figure. Then we have a vector, v, that will normally not lie in this subspace. The task is to find a linear combination of the m_i's with a minimal squared distance to v,

$$|\bar{r}|^2 = |m_1\theta_1 + m_2\theta_2 + m_3\theta_3 + \ldots - v|^2. \tag{8.19}$$

From linear algebra we know that the closest point in the subspace is the orthogonal projection. But how do we find it? One way is to study the deviation

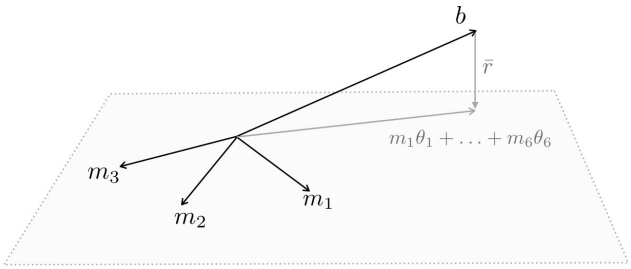


Figure 8.4: Linear least squares.

$\bar{r}$. We just stated that it should be orthogonal to the subspace. That means it must be orthogonal to all the m_i's. Orthogonality means that the scalar product,

$$m_i^T \bar{r} = m_i^T (M\theta - v) = 0. \tag{8.20}$$

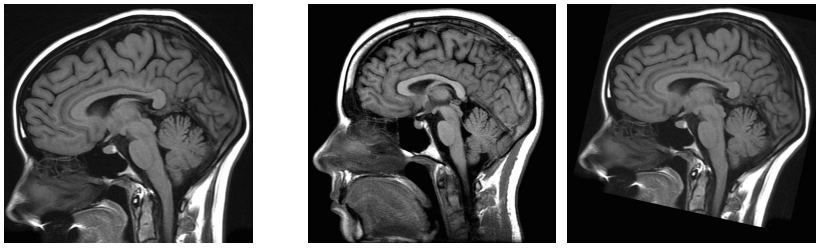
This hold for all i. Stacking these constraints we get the

$$M^T \bar{r} = M^T (M\theta - v) = 0 \quad \Leftrightarrow \quad M^T M\theta = M^T v. \tag{8.21}$$

The matrix $M^T M$ is a square 6×6-matrix, so we are no longer dealing with an overdetermined system!

Remark. So what is the purpose of knowing the normal equations? Well, in cases with a many points, M is a huge matrix. So just allocating space for it is expensive. In certain settings it is a better solution to form $M^T M$ and $M^T v$ directly. Another purpose will appear in the next chapter.

8.5 Free-form registration *



(a) (b) (c)

Figure 8.5: (a) and (b) show brain MRI's from different subjects and (c) shows the result of an affine registration of (a) to (b). Note the remaining shape variation.

Figure 8.5a-b shows magnetic resonance images (MRI) from different subjects. In Figure 8.5c the right image has been warped with an affine transformation to the left image, but due to shape variations between the two subjects, the alignment is far from perfect. This also limits the quality that can be obtained with atlas-based segmentation. In this section we will describe a richer class of transformations that is very useful for medical images and atlas-based segmentation.

Recall that image warping is done backwards. To find the pixel intensity of a pixel in the warped image we take its coordinates (x, y) and apply the coordinate transformation to obtain $(\tilde{x}, \tilde{y})$. We then assign the

$$I_w(x, y) = I_s(\tilde{x}, \tilde{y}). \tag{8.22}$$

To describe free-form transformations that allow local deformations of the image, we will write the coordinate transformation as

$$\begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} = T(x, y) = \begin{pmatrix} x \\ y \end{pmatrix} + \Delta(x, y). \tag{8.23}$$

If $\Delta \equiv 0$, we will create a copy of the source image. Hence, deformation function or deformation field is a good name for Δ. In most cases it is sufficient to estimate Δ for each pixel in the warped image. This means that we can visualize Δ as two matrices/images, one for the row deformation and one for the column deformation. Figure 8.6 shows an example where all the deformation lies in the row direction, that is, the second dimension of $\Delta(x, y)$ is identically zero.

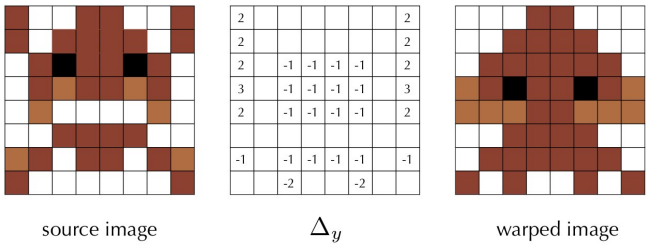


Figure 8.6: Example free-form transformation with $\Delta_x \equiv 0$. Non-zero values of Δ_y are shown in the middle. For clarity the zeros have been left out.

Figure 8.7 shows a larger example with deformation in both row and column directions. For this example, I manually selected a deformation field Δ. For medical applications, we need a way to parameterize Δ. We will write it as a sum of basis functions

$$\Delta_x(x, y) = \sum_{k=1}^{n} \alpha_k B_k(x, y) \quad \text{and} \quad \Delta_y(x, y) = \sum_{k=1}^{n} \beta_k B_k(x, y). \tag{8.24}$$

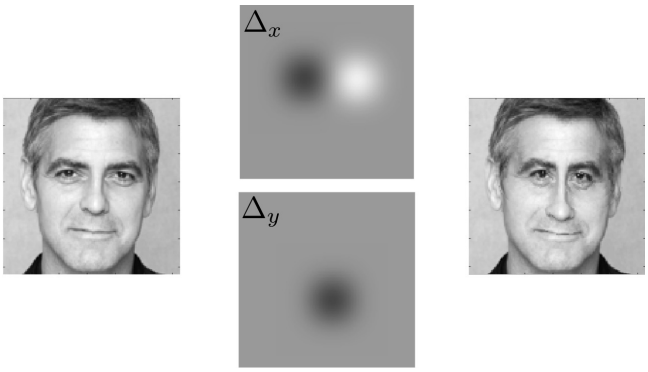


Figure 8.7: Example free-form transformation. Because of the size the deformation is shown as an image rather than with numbers. Note that grey corresponds to zero, white to positive values and black to negative values.

The process of determining a transformation boils down to finding appropriate values for the α_k's and the β_k's. But first we need to choose a class of basis functions. A few desired characteristics can be used to guide our choice.

- The ability to describe local transformations, such prolonging George Clooney's nose without changing the rest of his face.
- Smoothness, so we don't create discontinuities in the image.
- Mathematically simple, so we can optimize it.

To meet the first requirement, we use basis functions which are zero everywhere except in a small patch of the image; cf. Figure 8.8. To obtain smoothness and mathematical simplicity, we use splines. Splines are polynomials glued together to form a smooth function. We leave out the details.

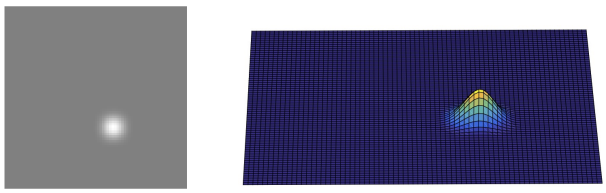


Figure 8.8: Basis function with local support, viewed as an image (left) and as a surface plot (right).

The next question is how to find the α_k's and β_k's of (8.24). By now you will recognize the approach. We choose a goal function and try to optimize. Naturally the appearance of the warped image, I_w depends on

$$\theta = (\alpha_1, \beta_1, \alpha_2, \beta_2, \ldots). \tag{8.25}$$

The task in registration is to create a warped image that is similar to the target image. Chapter 2 discussed a few different measures of image similarity. A common one for medical images is correlation. Hence we want to find θ, such that

$$\mathrm{Corr}\left(I_t, I_w(\theta)\right) \tag{8.26}$$

is maximized. Naturally, this is a complex function of θ and we cannot hope to find a global optimum, but if the deformation is moderate, we can often get very good results with local optimization such as gradient descent, Gauss-Newton or Levenberg-Marquardt.

8.6 Multi-atlas segmentation

We have already talked about image segmentation and how it can be solved as a classification problem using a convolutional neural network. One drawback of this approach is that we might need hundreds of images to train a good network, and especially for 3D medical images, manual segmentation is a very tedious process. An alternative approach that can work fairly well even for a few training images is multi-atlas segmentation. An atlas is an image together with a manual segmentation. By aligning this image to a new image, we can also transfer the segmentation. We simply warp the label image using the computed transformation.

Having multiple atlases, we can align all to the new image and perform majority voting for each pixel. A pixel is deemed as foreground if a majority of the atlases thinks so.

Chapter 9

Camera geometry

If light passes through a small hole into a dark room it will project an image of the outside world on the wall of the room; see Figure 9.2. This room is the veiled chamber, the *camera obscura*, that has given our modern cameras their name. A pinhole camera uses the same principle to produce a photograph. A sensor (or photographic plate) is placed inside a small box and a tiny *pinhole* is made in the wall of the box. Figure 9.1 shows the principle. Light from the tree passes through the pinhole and projects and image on the camera wall.

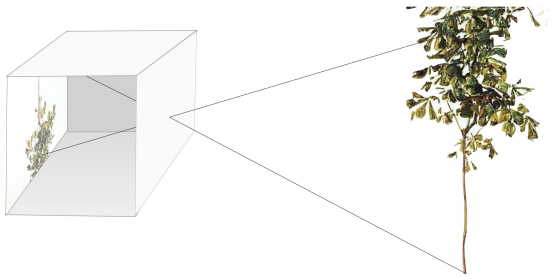


Figure 9.1: Principle of a pinhole camera.

The main purpose of this chapter is to derive an equation that relates a point in the three-dimensional world to the pixel coordinates of its projection in a pinhole camera. This simple model with prove to be very useful even for modern-day cameras using complex lenses.

In a real pinhole camera, the projected image is upside down, but mathematically, it is more practical to place the image plane in front of the pinhole as in Figure 9.3. Let W be the vector from the camera centre to a point in the scene, and w the vector to its projection in the image. Then

$$\lambda w = W \quad \text{with} \quad \lambda > 0. \tag{9.1}$$

Figure 9.2: A camera obscura.

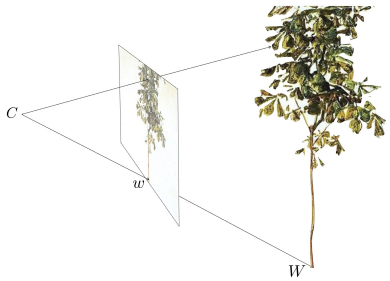


Figure 9.3: Model of a pinhole camera.

This simple observation is the first step towards a useful camera equation. We will refer to λ as the depth of point W.

9.1 Camera calibration

Our next task is to relate w to actual pixel coordinates in the image. To do so, we need to look closer at the geometry of the camera. First, we define the camera coordinate system. Its origin is at the camera centre and two axes are parallel to the image plane. The third axis is perpendicular to the image plane; cf. Figure 9.4.

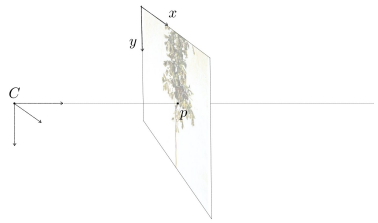


Figure 9.4: Camera coordinate system with C marking the camera centre. This is the origin of the camera coordinate system. The axes of this system are also indicated. In the image the principal point has been marked with a p.

Focal length. The distance from the camera centre to the image plane is the *focal length*, f, of the camera. Clearly the third coordinate $w_3 = f$. In the exif tags of the image, you will often find a rough measure of the focal length in millimetres. It is also possible to estimate the focal length from data.

Principal point. What about the first two coordinates of w? How do they relate to the column and row coordinates, (x, y), measured in pixels, that we have used in earlier chapters to specify a position in an image? First of all, we

measure pixel coordinates starting at the upper left corner of the image, but now we want to start at the camera centre, so we need to move the origin. The orthogonal projection of the camera centre into the image plane, is called the *principal point*; see Figure 9.4. Let (p_x, p_y) be its pixel coordinates. Camera manufacturers do their best to centre the image sensor with respect to the camera centre, so it is normally safe to assume that the principal point lies exactly in the middle of the image.

Pixel size. Finally, as we know the focal length in millimetres and image positions are measured in pixels, we need the relationship between these two. Again, we can compute this from information about the camera. In the exif tags, you can find the camera type and on the internet you will find the sensor size in millimetres. Dividing that with the number of pixels produces the pixel size, ϱ, in millimetres.

The calibration matrix. Knowing the focal length, f, the principal point (p_x, p_y) and the pixel size ϱ, we can compute w of (9.1) as

$$w = \begin{pmatrix} w_1 \\ w_2 \\ f \end{pmatrix} = \begin{pmatrix} (x - p_x)\varrho \\ (y - p_y)\varrho \\ f \end{pmatrix}. \tag{9.2}$$

Often, this equation is written in *homogeneous coordinates* by adding an extra 1 to the pixel coordinates (x, y)

$$w = \begin{pmatrix} (x - p_x)\varrho \\ (y - p_y)\varrho \\ f \end{pmatrix} = \underbrace{\begin{pmatrix} \varrho & 0 & -\varrho\, p_x \\ 0 & \varrho & -\varrho\, p_y \\ 0 & 0 & f \end{pmatrix}}_{K^{-1}} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = K^{-1}u. \tag{9.3}$$

This equation also defined the calibration matrix, K. Here, we wrote all coordinates in millimetres, but due to the arbitrary $\lambda > 0$ in (9.1), we can always rescale K with a non-negative number. It will work just as well with

$$K^{-1} = \begin{pmatrix} 1 & 0 & -p_x \\ 0 & 1 & -p_y \\ 0 & 0 & f/\varrho \end{pmatrix} \tag{9.4}$$

that corresponds to having all coordinates in pixels.

9.2 In global coordinates

Now we can rewrite (9.1) as

$$\lambda\, K^{-1}u = W, \quad \text{or} \quad \lambda u = KW \tag{9.5}$$

with W in the camera coordinate system. Using camera coordinates is fine as long as we are only dealing with one, stationary camera, but already with two cameras we need to introduce a global coordinate system.

Let U be the coordinates of a 3D point in the global coordinates, C the camera centre and R a rotation mapping global coordinates to camera coordinates. Together C and R, define the *pose* of the camera, i.e., the position and orientation. Returning to (9.5), we can replace W with $R(U - C)$, yielding

$$\lambda u = KR(U - C). \tag{9.6}$$

As with (9.3), this equation is often written in homogeneous coordinates. We simply add a 1 at the end of X, making it into a 4-vector,

$$\lambda\, u = \underbrace{K \left[\; R \;\middle|\; -RC \;\right]}_{P} \begin{pmatrix} U \\ 1 \end{pmatrix}. \tag{9.7}$$

The 3×4 matrix P is called the camera matrix.

Often people will use U both for the 3-vector and for the homogeneous 4-vector, leaving to the reader to work out which is intended. Doing so yields the most familiar version of the camera equation

$$\lambda\, u = PU \quad \text{with} \quad \lambda > 0. \tag{9.8}$$

9.3 Two variants of the camera equation

If the calibration matrix, K, is known, we can move effortlessly between the two variants of the camera equation. The first is the variant in (9.7) and (9.8)

$$\lambda u = \lambda \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \left[\; KR \;\middle|\; -KRC \;\right] U = PU. \tag{9.9}$$

The advantage of this version is that the image point u is just a homogeneous variant of the actual pixel coordinates (x, y). The other way to write the camera equation is

$$\lambda u_c = \lambda K^{-1} u = \left[\; R \;\middle|\; -RC \;\right] U = P_c U. \tag{9.10}$$

This version is preferable if the camera matrix is unknown, since we know that the left 3×3-submatrix of P_c is a rotation. Keep in mind that you cannot use this variant directly with pixel coordinates, but first you have to calibrate the coordinates by multiplication with K^{-1}.

If you are given a camera matrix and want to determine whether it is calibrated or uncalibrated, just check if the left 3×3-submatrix is a rotation. Note that the actual value of the depth, λ, depends on the choice of equation.

9.4 Lens distortion ⋆

Consider an image of a linear structure in 3D. Points on a line has coordinates

$$U = V + sW \tag{9.11}$$

where s is a parameter and V 3-vector. Now consider the projection of a point on this line.

$$u = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \frac{1}{\lambda} PU = \frac{1}{\lambda} \left(PV + sPW \right) = \frac{1}{\lambda} \left(\tilde{V} + s\tilde{W} \right), \tag{9.12}$$

where $\tilde{V}$ and $\tilde{W}$ are just new 3-vectors. If we take the scalar product with $\tilde{V} \times \tilde{W}$, we get

$$\left(\tilde{V} \times \tilde{W} \right) \cdot u = 0. \tag{9.13}$$

Noting that $\tilde{V} \times \tilde{W}$ is just a 3-vector, we see that this can be rewritten as

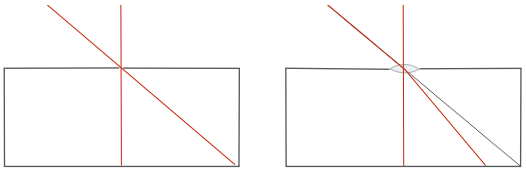
$$ax + by + c = 0. \tag{9.14}$$

This means that a line in 3D is mapped onto a line in the image, but then how come we can get images like Figure 9.7a? The reason is that most images are not captured with pinhole cameras and that most cameras do not work like an ideal pinhole camera. The deviation is caused by the lens and thus called lens distortion.

(a) (b)

Figure 9.5: (a) Image with radial lens distortion. (b) Distortion removed.

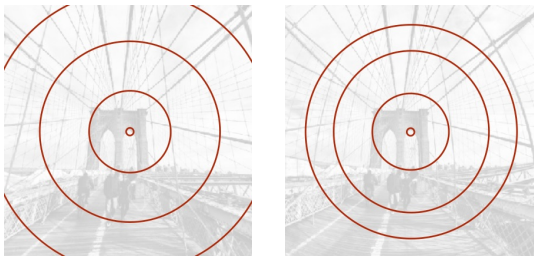
Figure 9.6 illustrates the principle. In a pinhole camera a light ray will pass through the pinhole without changing direction, but in a lens camera—depending on the type of lens or lenses—the ray might change direction. Camera lenses tend to be circularly symmetric so this effect will also be symmetric



(a) (b)

Figure 9.6: (a) Light rays in a pinhole camera. (b) In a lens camera the rays change direction due to refraction.

about the image centre—or more precisely the principal point. Figure 9.7 illustrates the principle. Each circle corresponds to the light rays with a certain angle of incidence compared to the image plane. In a pinhole camera these the relationship between angle of incidence and circle radius is easily determined, but for



(a) (b)

Figure 9.7: Each circle corresponds to the light rays with a certain angle of incidence. (a) A pinhole camera. (b) The corresponding circles in a lens camera with significant radial distortion.

Naturally we could make an accurate physical model of the lens and include that in our camera model, but this would make the geometric estimation problems that we will look at in the next chapter difficult and it would also make us dependent on the type of camera that we are using. Instead, we will use a simple polynomial model to revert the lens distortion. Having such a model, we can always undistort the image before starting with the model estimation. After that, we can work with any image as if it were from an ideal pinhole camera.

Let u_c be calibrated image coordinates, i.e., measured with the origin at the principal point. We can model radial distortion as

$$u_d = u_c f(|u_c|). \tag{9.15}$$

Note that $|u_c|$ here is the distance to the principal point—typically at the centre of the image. This means that the function f describes how much the rays are

offset at different distances to the principal point. We won't attempt to use a correct physical model for f. Instead we assume that a simple polynomial will be accurate enough. Normally, a polynomial with only even-order monomials is used, for example,

$$f(|u_c|) = 1 + \kappa_1 |u_c|^2 + \kappa_2 |u_c|^4. \tag{9.16}$$

Figure 9.7b shows the image of Brooklyn bridge after undistorting it with this type of model.

Chapter 10

3D reconstruction

10.1 Three basic problems

3D reconstruction is the process of automatically building a 3D model from a set of images. Except in some very special cases it is not possible to solve for the whole model in a single step. Instead, an iterative process is used, breaking down the problem into small basic problems that can be solved with Ransac.

The first step of this process is to estimate the *relative pose* of two cameras. The input is a set of point-to-point correspondences between the images, obtained by feature matching. Possibly we also know the calibration matrices of the two cameras. Having estimate the relative pose of these two cameras we can start iterating the following steps

- Estimate 3D points using *triangulation*.
- Estimate the pose of a new camera.

This process is the reason why these three problems hold a central place in 3D reconstruction:

- Relative pose estimation.
- Triangulation.
- Camera pose estimation.

In this course, we will only consider the case of known calibration matrices, K_i.

10.2 Scale ambiguity

Consider the general 3D reconstruction problem with multiple images/cameras and multiple 3D points. Assume that somehow, we have found a solution, that

is we have found an orientation, R_k and a position C_k for each camera and coordinates U_i for each 3D point, that satisfy the calibrated camera equations

$$\lambda_{k,i} u_{k,i} = R_k(U_i - C_k) \quad \lambda_{k,i} > 0. \tag{10.1}$$

Now consider, multiplying each equation with $s > 0$,

$$\underbrace{s\lambda_{k,i}}_{=\lambda'_{k,i}} u_{k,i} = 2R_k(U_i - C_k) = R_k(\underbrace{sU_i}_{=U'_i} - \underbrace{sC_k}_{C'_k}), \tag{10.2}$$

so we have

$$\lambda'_{k,i} u_{k,i} = R_k(U'_i - C'_k) \quad \lambda'_{k,i} > 0. \tag{10.3}$$

This shows that there is another solution with the same camera orientations, but camera positions C'_k and 3D points U'_i that also satisfies the camera equations. The difference between the two solutions being a scale factor. Note that this does not depend on the number of equations and unknowns. The problem is built in to the structure of the camera equation. We simply cannot recover the absolute scale solely from this type of image measurements.

10.3 Triangulation using Ransac

Assume that we have a number of images of the same scene, let us say three, and that we already know the camera matrices for these cameras, P_1, P_2 and P_3. Moreover, we have computed feature points for all three images and matched them yielding a triplet of corresponding points, u_1 in the first image, u_2 in the second and u_3 in the third. The assumption is that these three points are the projections of a single 3D point U; cf. Figure 10.1. We will now try to estimate the coordinates of this U.

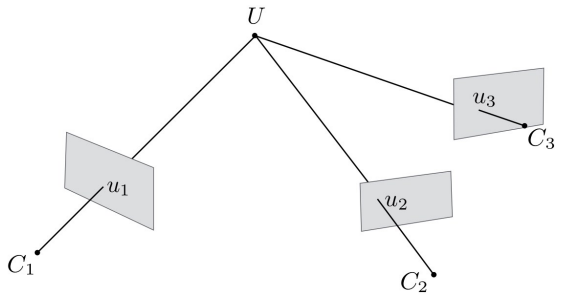


Figure 10.1: The setup in triangulation, except that with real measurements, due to noise, the rays will not intersect at a single point.

Using (9.8) for each of the cameras yields a set of equations,

$$\begin{aligned} \lambda_1 u_1 &= P_1 U \quad \text{with} \quad \lambda_1 > 0 \\ \lambda_2 u_2 &= P_2 U \quad \text{with} \quad \lambda_2 > 0 \\ \lambda_3 u_3 &= P_3 U \quad \text{with} \quad \lambda_3 > 0. \end{aligned} \tag{10.4}$$

Suppose we want to use this system to find U. If we are not sure that all three measurements are correct, we want to use Ransac together with a robust loss function.

A minimal solver. The first thing we need for Ransac is a minimal solver. There are three unknown coordinates in U, so normally three equations would do. However, each set of equations in (10.4) introduces an extra unknown λ_i. This means that we will need to select two views (or cameras) for the minimal solver. Let's say we got number two and three. We get the system

$$\begin{aligned} \lambda_2 u_2 &= P_2 U \\ \lambda_3 u_3 &= P_3 U, \end{aligned}$$

with six equations and five unknowns. Clearly this is overdetermined. An easy way to change that is to simply throw away one equation to yield a quadratic system. Note that the equations are linear so after some rearranging to get them on the form $M\theta = b$ we can solve the system in Matlab using `theta = M\b`. As always, θ, is a vector with all the unknown parameters, in this case the three coordinates in U together with λ_1 and λ_2.

A loss function. For Ransac, we also need a loss function and before that a way to measure residuals. This is the topic of the next section.

10.4 Reprojection errors

Working with camera geometry, our measurements will always be image points given pixel coordinates, typically SIFT points. Through (9.8), these points are used to estimate the pose of a camera or a 3D point as we saw in the previous section. Given such an estimate, we should project it back in all the images to see how well it fits with the original measurements.

Let us continue with the example from the last section. To see if a Ransac hypothesis, $\hat{U}$, is any good, we project it into each camera and measure how well it fits with the measured image points. To find a formula, we need to work with the rows of the camera matrix. Let

$$P = \begin{pmatrix} \leftarrow & a^T & \rightarrow \\ \leftarrow & b^T & \rightarrow \\ \leftarrow & c^T & \rightarrow \end{pmatrix}, \tag{10.5}$$

where the arrows just indicate that a^T is the whole row. If $\hat{u}$ is the reprojection of $\hat{U}$ into this camera, it should satisfy the camera equation

$$\lambda\hat{u} = \lambda \begin{pmatrix} \hat{x} \\ \hat{y} \\ 1 \end{pmatrix} = P\hat{U} = \begin{pmatrix} a^T\hat{U} \\ b^T\hat{U} \\ c^T\hat{U} \end{pmatrix} \quad \text{with} \quad \lambda > 0 \tag{10.6}$$

To find the actual pixel coordinates we need to divide by λ. The third row of (10.6) gives us

$$\lambda = c^T\hat{U}, \tag{10.7}$$

so we get

$$\hat{x} = \frac{a^T\hat{U}}{c^T\hat{U}} \quad \text{and} \quad \hat{y} = \frac{b^T\hat{U}}{c^T\hat{U}}, \quad \text{if} \quad c^T\hat{U} > 0 \tag{10.8}$$

To see how well this fits to the measured image coordinates, (x, y), we compute the residual vector

$$r(\theta) = \begin{pmatrix} \hat{x} \\ \hat{y} \end{pmatrix} - \begin{pmatrix} x \\ y \end{pmatrix}, \quad \text{if} \quad c^T\hat{U} > 0. \tag{10.9}$$

We will refer to this as *reprojection residuals*. The absolute residuals $|r(\theta)|$ are often called reprojection errors. The *re* in reprojection comes from the fact that we start with measured image coordinates, estimate 3D points from these measurements, and then project these points back into the images. Note that the formula only holds for points with positive depth. Points with negative depth can be said to have infinite reprojection error.

Given many measurements (as in the Ransac example) we compute the reprojection residual, $r_i(\theta)$, for each measurement and throw them all into one of the robust loss function discussed in Chapter 4. Just as in the case of registration, the following pipeline is recommended:

- Use Ransac to obtain a rough estimate of the parameters (U).
- Remove all measurements which are outliers with respect to these parameters.
- Estimate the least squares parameters using the remaining measurements.

This last point is the topic of the next section.

10.5 Nonlinear least squares

We will continue with the triangulation problem. Assume that we have already performed Ransac and removed outliers. Hence the sum of squared residuals

is reasonable loss for the remaining measurements. Let $\bar{r}(\theta)$ be a vector with the residuals of remaining measurements.

$$\bar{r}(\theta) = \begin{pmatrix} r_{1,x}(\theta) \\ r_{1,y}(\theta) \\ r_{2,x}(\theta) \\ r_{2,y}(\theta) \\ \vdots \end{pmatrix} \tag{10.10}$$

In Section 8.4, this vector was a linear function of the model parameters,

$$\bar{r}(\theta) = M\theta - v, \tag{10.11}$$

and we could find the least squares solution by solving the normal equations or using `M\v` in Matlab. For reprojection residuals, the relationship is not linear. This makes it considerably harder to find a least squares solution. What if we linearize residual vector? Recall from multivariate calculus, that a function $f(\theta)$ can be approximated at a point $\theta = \alpha$ with its first order Taylor expansion,

$$f(\theta) \approx f(\alpha) + \nabla f(\alpha)^{\mathbf{T}}(\theta - \alpha). \tag{10.12}$$

The approximation is good in a neighbourhood of α. For a vector-valued function, such as $\bar{r}(\theta)$, we simply apply this to each dimension. Writing this down gets slightly less tedious if we introduce the Jacobian matrix,

$$J = \begin{pmatrix} \frac{\partial r_{1,x}}{\partial \theta_1} & \frac{\partial r_{1,x}}{\partial \theta_2} & \cdots \\ \frac{\partial r_{1,y}}{\partial \theta_1} & \frac{\partial r_{1,y}}{\partial \theta_2} & \cdots \\ \frac{\partial r_{2,x}}{\partial \theta_1} & \frac{\partial r_{2,x}}{\partial \theta_2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}. \tag{10.13}$$

Using this definition we can write the Taylor expansion of the vector-valued $\bar{r}(\theta)$ as

$$\bar{r}(\theta) \approx \bar{r}(\alpha) + J(\alpha)\left(\theta - \alpha\right). \tag{10.14}$$

Note that approximation is linear. We have,

$$\bar{r}(\theta) = M\theta - v, \tag{10.15}$$

with $M = J(\alpha)$ and $v = \bar{r}(\alpha) - J(\alpha)\alpha$. According to Section 8.4.1, we get the least squares solution by solving can solve the normal equation

$$M^{\mathbf{T}}M\theta = M^{\mathbf{T}}v \tag{10.16}$$

which in this case is equivalent to

$$J^{\mathbf{T}}J\left(\theta - \alpha\right) = J^{\mathbf{T}}J\alpha - J^{\mathbf{T}}\bar{r}, \tag{10.17}$$

to get least squares solution (to the linear approximation in (10.14)). (Note that $J = J(\alpha)$ and $\bar{r} = \bar{r}(\alpha)$.)

The solution is

$$\theta = \alpha - (J^T J)^{-1} J^T \bar{r}. \tag{10.18}$$

Note that this is only a least squares solution to the linearized problem. But if α is close enough to the true optimum, the linear approximation will be accurate. Hence, we will use this trick iteratively. We set $\theta^{(0)}$ to the parameters estimated with Ransac and then proceed iteratively:

Given an parameter estimate $\theta^{(k)}$, we compute the residuals and the Jacobian at $\theta^{(k)}$ and use (10.18) to obtain a better estimate

$$\theta^{(k+1)} = \theta^{(k)} - \underbrace{(J^T J)^{-1} J^T \bar{r}}_{\text{evaluated at } \theta^{(k)}}. \tag{10.19}$$

This is iterated a fixed number of times or until the parameter update is small enough. This method is called the Gauss-Newton algorithm.

10.6 Minimal solvers and polynomial equations ⋆

The most difficult part of implementing Ransac for a new problem is often to construct a minimal solver. To introduce some useful methods for this process, we will consider camera pose estimation.

Camera pose estimation is the process of estimating the position and orientation of a camera with respect to a set of known 3D model. The starting point for this is a set of correspondences between image points and 3D points. We will assume that the calibration matrix, K, is known, for example from the exif tags. This means we can use the calibrated camera equation

$$\lambda_i u_i = \lambda_i K^{-1} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = R(U_i - C) = RU_i + t \tag{10.20}$$

The main difficulty will be to ensure that the

$$R = \begin{pmatrix} \uparrow & \uparrow & \uparrow \\ a & b & c \\ \downarrow & \downarrow & \downarrow \end{pmatrix}, \tag{10.21}$$

that we find is a rotation matrix. Using knowledge from linear algebra, we can put the following constraints

$$a^T a = b^T b = c^T c = 1, \tag{10.22}$$

and

$$a^T b = 0 \quad \text{and} \quad a \times b = c. \tag{10.23}$$

Using these six equations together with the camera equation for three pairs, (u_i, U_i), it is theoretically possible to solve for all the unknowns. We use this example to show how you can work with this type of problem.

Choosing coordinate system

We are always free to change coordinate system. In this case it is always possible to change coordinates such that

$$\begin{pmatrix} 0 & 0 & 1 \end{pmatrix} U_i = 0 \tag{10.24}$$

This means that the third column c in (10.21) will never turn up in the camera equations and we can solve for a and b independently. We can also move the origin to U_1. After also rescaling the unknowns to eliminate λ_1, we get the equations

$$\begin{aligned} u_1 &= t \\ \lambda_2 u_2 &= aU_{2,1} + bU_{2,2} + t \\ \lambda_3 u_3 &= aU_{3,1} + bU_{3,2} + t \\ a^T a &= b^T b \\ a^T b &= 0 \end{aligned} \tag{10.25}$$

This is not untypical. We have a set of nine linear equations and two polynomial equations. We can use the linear equations to eliminate nine unknowns, leaving us with just two unknowns and two quadratic equations. This can be done manually or using numerical algorithms to compute an SVD or a QR factorization. Hence, a relevant question in this context is how to solve small systems of low-order polynomial equations. Note that u here are the a homogeneous vector with the pixel coordinates (x, y).

Systems of polynomial equations

Using modern numerical linear algebra, it is possible to solve systems of linear equations with thousands of variables. For systems of polynomial equations we need to work very hard even for four to five equations. Still, these small systems happen to be very useful when working with camera geometry.

A powerful technique for solving polynomial systems in floating point arithmetic uses something called the action matrix, but going through this in detail is out of our scope. For really small systems such as the one arising in camera pose estimation, we can often reduce the system to just one equation in one unknown. A systematic way to do this is using the resultant. Consider the following system of two equations,

$$x^2 + y^2 - 2 = 0 \tag{10.26}$$

$$xy - 1 = 0. \tag{10.27}$$

We can rewrite this as

$$\begin{pmatrix} 1 & 0 & x^2 - 2 \\ 0 & x & -1 \end{pmatrix} \begin{pmatrix} y^2 \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}. \tag{10.28}$$

This is a linear system in the unknowns y and y^2, but there are not enough equations to solve it. If we multiply the original equations with y we create new linearly independent equations,

$$y(x^2 + y^2 - 2) = y^3 + x^2y - 2y = 0 \tag{10.29}$$

$$y(xy - 1) = xy^2 - y = 0. \tag{10.30}$$

If we add these to the linear formulation we get

$$\begin{pmatrix} 1 & 0 & x^2-2 & 0 \\ 0 & 1 & 0 & x^2-2 \\ 0 & x & -1 & 0 \\ 0 & 0 & x & -1 \end{pmatrix} \begin{pmatrix} y^3 \\ y^2 \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}. \tag{10.31}$$

Using Gaussian elimination we can rewrite this

$$\begin{pmatrix} 1 & 0 & x^2-2 & 0 \\ 0 & 1 & 0 & x^2-2 \\ 0 & 0 & -1 & 2x-x^3 \\ 0 & 0 & 0 & 2x^2-x^4-1 \end{pmatrix} \begin{pmatrix} y^3 \\ y^2 \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}. \tag{10.32}$$

Viewed as a linear system, $M\theta = 0$, one solution should be apparent, namely $\theta = 0$. But this cannot be the one we seek the sought-for vector contains a 1. This means that the equation system must have multiple solutions and hence,

$$0 = \det(M) = x^4 - 2x^2 + 1. \tag{10.33}$$

By solving this equation we get a set of x's that we insert to the original equations to find the corresponding values of y.

Remarks. Instead of computing the determinant by hand, you can always use a symbolic software such as WolframAlpha. Also note that for larger systems, this method tends to get into numerical problems. Methods using action matrices are more stable.

10.7 Relative pose estimation and the essential matrix *

In relative pose estimation, we have a set of point-to-point correspondences between two views and want to work out the camera matrices for both cameras. We will consider the calibrated case, where the calibration matrices for both cameras are known.

Since the global coordinate system is arbitrary we can always set the first camera at the origin with the canonical orientation

$$P_c = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{10.34}$$

The second camera matrix is arbitrary (but calibrated), so

$$\tilde{P}_c = \begin{pmatrix} R \mid t \end{pmatrix}. \tag{10.35}$$

For a point-to-point correspondence $u \to \tilde{u}$, we get the equations

$$\lambda u = U \tag{10.36}$$

$$\tilde{\lambda}\tilde{u} = RU + t \tag{10.37}$$

Note that

$$\tilde{u}^T (t \times Ru) = \frac{1}{\lambda\tilde{\lambda}}(RU + t)^T (t \times RU) = 0. \tag{10.38}$$

We have found a way to eliminate the 3D points U_i from the problem.

10.7.1 The essential matrix

Consider the vector product of vectors a and w. It is easy to see that it can be written as a matrix multiplication,

$$a \times w = \begin{pmatrix} a_2w_3 - a_3w_2 \\ a_3w_1 - a_1w_3 \\ a_1w_2 - a_2w_1 \end{pmatrix} = \begin{pmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{pmatrix} w = [a]_\times w. \tag{10.39}$$

If we use this for the expression in (10.38), we get the alternative formula

$$\tilde{u}^T [t]_\times Ru = 0 \tag{10.40}$$

The matrix $E = [t]_\times R$ is often called the essential matrix. It captures most of the information in P_c and $\tilde{P}_c$, but is easier to work with if we aim for a minimal solver. Note that (10.40) only involves the measured image points and the essential matrix, so we do not need to parameterize the 3D points U_i.

10.8 Differentiating rotations ⋆

Section 10.6 described how to construct a minimal solver for camera pose estimation. This solver can be used with Ransac to obtain a fair estimate of the camera pose. Having done so, we would like to proceed with Gauss-Newton to get a least squares solution. First we remove the outlier measurements using the Ransac estimate. Let us say that this leaves n measurements, being pairs (u_i, U_i). To use Gauss-Newton, we need to work out the derivatives of the residuals with respect to the unknown parameters. Unfortunately, the parameterization that we used for the minimal solver does not work very well. The problem is that we have two polynomial equations

$$a^T b = 0 \quad \text{and} \quad a^T a = b^T b, \tag{10.41}$$

that have to be fulfilled for the parameters to describe a rotation. Performing a Gauss-Newton step, we cannot enforce this, so our next estimate might not be a rotation. Luckily, there is a better parameterization. First we need a few facts about matrix exponentials.

Matrix exponentials

The matrix exponential function of a 3×3-matrix, X, is defined as

$$\exp U = \sum_{k=0}^{\infty} \frac{X^k}{k!} = I + X + \frac{X^2}{2} + \frac{X^3}{6} + \dots \tag{10.42}$$

and has the following characteristics.

Theorem 1.

$$\frac{d}{d\varphi} \exp(A\varphi) = A \exp(A\varphi). \tag{10.43}$$

Proof.

$$\frac{d}{d\varphi} \exp(A\varphi) = \frac{d}{d\varphi} \left(I + A\varphi + \frac{A^2\varphi^2}{2} + \frac{A^3\varphi^3}{6} + \dots \right) \tag{10.44}$$

$$= 0 + A + \frac{A^2 2\varphi}{2} + \frac{A^3 3\varphi^2}{6} + \dots \tag{10.45}$$

$$= A \left(I + A\varphi + \frac{A^2\varphi^2}{2} + \dots \right) = A \exp(A\varphi). \tag{10.46}$$

□

Theorem 2. *If T is an orthogonal matrix, then*

$$\exp(T^{\mathtt{T}} U T) = T^{\mathtt{T}} \exp(U) T \tag{10.47}$$

Proof. Consider

$$(T^{\mathtt{T}} U T)^k = T^{\mathtt{T}} U T T^{\mathtt{T}} U T T^{\mathtt{T}} \dots U T = T^{\mathtt{T}} U^k T. \tag{10.48}$$

This means that

$$\exp(T^{\mathtt{T}} U T) = \sum_{k=0}^{\infty} \frac{(T^{\mathtt{T}} U T)^k}{k!} = \sum_{k=0}^{\infty} \frac{T^{\mathtt{T}} U^k T}{k!} = T^{\mathtt{T}} \exp(U) T. \tag{10.49}$$

□

Theorem 3. *For any a with $|a| = 1$,*

$$\exp([a]_\times \varphi) \tag{10.50}$$

is a rotation φ radians about a.

Proof. A rotation is defined by a unit-length rotation axis, a and an angle φ. Select b and c such that a, b and c form a right-handed ON-basis for $\mathbb{R}^3$. The orthogonal matrix

$$T = \begin{pmatrix} \leftarrow & a^{\mathtt{T}} & \rightarrow \\ \leftarrow & b^{\mathtt{T}} & \rightarrow \\ \leftarrow & c^{\mathtt{T}} & \rightarrow \end{pmatrix}, \tag{10.51}$$

changes between the standard basis and the abc-basis. In the abc-basis, a rotation about a has the rotation matrix

$$R_\circ = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi \\ 0 & \sin\varphi & \cos\varphi \end{pmatrix}. \tag{10.52}$$

We know from linear algebra that if M is a transformation matrix and T is an orthogonal basis transformation, then $T^{\mathbf{T}}MT$ is the transformation matrix in the new basis, so

$$R = T^{\mathbf{T}} R_\circ T \tag{10.53}$$

is the rotation matrix in the standard basis. A similar relationship holds for the vector product.

$$S = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix} \tag{10.54}$$

performs vector product with a in the abc-basis and

$$[a]_\times = T^{\mathbf{T}} S T \tag{10.55}$$

in the standard basis. It is easy to verify that

$$S^2 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix}. \tag{10.56}$$

Note that this is equivalent to

$$a \times (a \times b) = -b, \quad a \times (a \times c) = -c, \tag{10.57}$$

which can also be seen geometrically. It follows that $S^3 = -S$, $S^4 = -S^2$, $S_5 = S$, $S^6 = S^2$, etc. This means that

$$\exp(S\varphi) = I + \left(\varphi - \frac{\varphi^3}{3!} + \frac{\varphi^5}{5!} + \ldots\right) S + \left(\frac{\varphi^2}{2!} - \frac{\varphi^4}{4!} + \ldots\right) S^2 \tag{10.58}$$

$$= I + \sin\varphi\, S + (1 - \cos\varphi) S^2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi \\ 0 & \sin\varphi & \cos\varphi \end{pmatrix} = R_\circ. \tag{10.59}$$

Now consider the rotation in the standard basis. From (10.53) we have

$$R = T^{\mathbf{T}} R_\circ T = T^{\mathbf{T}} \exp(S\varphi) T \tag{10.60}$$

and using Theorem 2 together with (10.55),

$$R = \exp(T^{\mathbf{T}} S T \varphi). = \exp([a]_\times \varphi) \tag{10.61}$$

Since a was an arbitrary unit vector, we have proven the theorem. □

A minimal parameterization

By considering the normal formula for the vector product, we see that $[a]_x\varphi$ is a general skew-symmetric matrix.

$$\Phi = \begin{pmatrix} 0 & -\varphi_3 & \varphi_2 \\ \varphi_3 & 0 & -\varphi_1 \\ -\varphi_2 & \varphi_1 & 0 \end{pmatrix}. \tag{10.62}$$

Hence, any rotation can be written $R = \exp\Phi$. The really useful formulas however, are that

$$\frac{\partial R}{\partial \varphi_1} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix} R, \qquad \frac{\partial R}{\partial \varphi_2} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{pmatrix} R \tag{10.63}$$

and

$$\frac{\partial R}{\partial \varphi_3} = \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} R. \tag{10.64}$$

These formulas are the keys to differentiating residuals with respect to rotation parameters φ_1, φ_2 and φ_3, thus also allowing us to perform Gauss-Newton on rotations.

Example 8. *Let*

$$R = \begin{pmatrix} \leftarrow & a^T & \rightarrow \\ \leftarrow & b^T & \rightarrow \\ \leftarrow & c^T & \rightarrow \end{pmatrix}. \tag{10.65}$$

Consider differentiating $b^TU + t_y$ with respect to φ_1,

$$\frac{\partial}{\partial \varphi_1}(b^TU + t_y) = \frac{\partial}{\partial \varphi_1}\left(\begin{pmatrix} 0 & 1 & 0 \end{pmatrix} RU + t_y\right) = \tag{10.66}$$

$$\begin{pmatrix} 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix} RU = -c^TU \tag{10.67}$$