CHAPTER 4

Patterns, Smoothing and Filters

In this chapter, we introduce methods for obtaining descriptions of the appearance of a small group of pixels. These methods can be used to find patterns, to suppress noise, and to control aliasing.

4.1 LINEAR FILTERS AND CONVOLUTION

Section 2.3.2 showed improvements in downsampling obtained by replacing each image pixel with a weighted average of pixels. This useful trick is easily generalized into an important idea. The weights in that section were either uniform, or large at the pixel of interest, and falling off at distant pixels. Changing the weights leads to interesting outcomes.

4.1.1 Convolution

For the moment, think of an image as a two dimensional array of intensities. Write $\mathcal{I}_{ij}$ for the pixel at position i, j. Construct a small array of weights (a *mask* or *kernel*) $\mathcal{W}$, and compute a new image $\mathcal{N}$ from the original image and the mask, using the rule

$$\mathcal{N}_{ij} = \sum_{uv} \mathcal{I}_{i-u,j-v} \mathcal{W}_{uv}$$

equivalently

$$\mathcal{N} = \mathcal{W} * \mathcal{I}.$$

In some sources, you might see $\mathcal{W} * * \mathcal{I}$ (to emphasize the fact that the image is 2D). This operation is known as *convolution*, and $\mathcal{W}$ is often called the *kernel* of the convolution. You should look closely at the expression; the "direction" of the dummy variable u (resp. v) has been reversed compared with what you might expect (unless you have a signal processing background).

What you might expect – sometimes called *correlation* or *filtering* – would compute

$$\mathcal{N}_{ij} = \sum_{uv} \mathcal{I}_{i+u,j+v} \mathcal{W}_{uv}$$

equivalently

$$\mathcal{N} = \texttt{filter}(\mathcal{I}, \mathcal{W}).$$

This difference isn't particularly significant, but if you forget that it is there, you compute the wrong answer.

Ignore the range of the sum, and assume that the sum is over a large enough range of u and v that all nonzero values are taken into account. Furthermore, assume that any values that haven't been explicitly specified are zero; this means that we can model the kernel as a small block of nonzero values in a sea of zeros. An

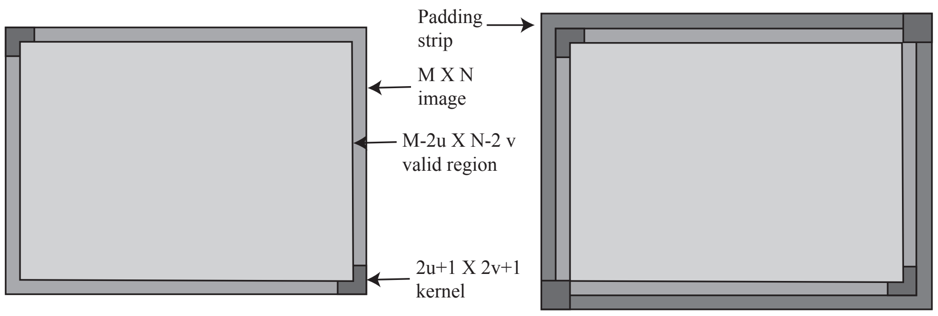

FIGURE 4.1: *The mid-gray box represents an $M \times N$ image, and the darker gray box a $2u+1 \times 2v+1$ kernel. The valid region is lighter gray. It can be constructed by placing the kernel at the top left and bottom right corners of the image, then constructing the box that joins their centers (***left***). A version of this construction reveals how the image should be padded to produce an $M \times N$ result. Place the center of the kernel at the bottom left and top right of the image, and construct the box that joins* their *outer corners (***right***).*

important property of convolution is that the result depends on the local pattern around a pixel, but not where the pixel is. Define the operation $\texttt{shift}(\mathcal{I}, m, n)$ which shifts an image so that the i, j'th pixel is moved to the $i-m$, $j-n$'th pixel, so

$$\texttt{shift}(\mathcal{I}, m, n)_{ij} = \mathcal{I}_{i-m,j-n}.$$

Ignore the question of the range, as `shift` just relabels pixel locations. Check that:

- Convolution is linear in the image, so

$$\begin{aligned} \mathcal{W} * (k\mathcal{I}) &= k(\mathcal{W} * \mathcal{I}) \\ \mathcal{W} * (\mathcal{I} + \mathcal{J}) &= \mathcal{W} * \mathcal{I} + \mathcal{W} * \mathcal{J}. \end{aligned}$$

- Convolution is linear in the mask, so

$$\begin{aligned} (k\mathcal{W}) * \mathcal{I}) &= k(\mathcal{W} * \mathcal{I}) \\ (\mathcal{W} + \mathcal{V}) * \mathcal{I}) &= \mathcal{W} * \mathcal{I} + \mathcal{V} * \mathcal{I} \end{aligned}$$

- Convolution is associative, so

$$\mathcal{W} * (\mathcal{V} * \mathcal{I}) = (\mathcal{W} * \mathcal{V}) * \mathcal{I}.$$

- Convolution is shift-invariant, so

$$\mathcal{W} * (\texttt{shift}(\mathcal{I}, m, n)) = \texttt{shift}(\mathcal{W} * \mathcal{I}, m, n).$$

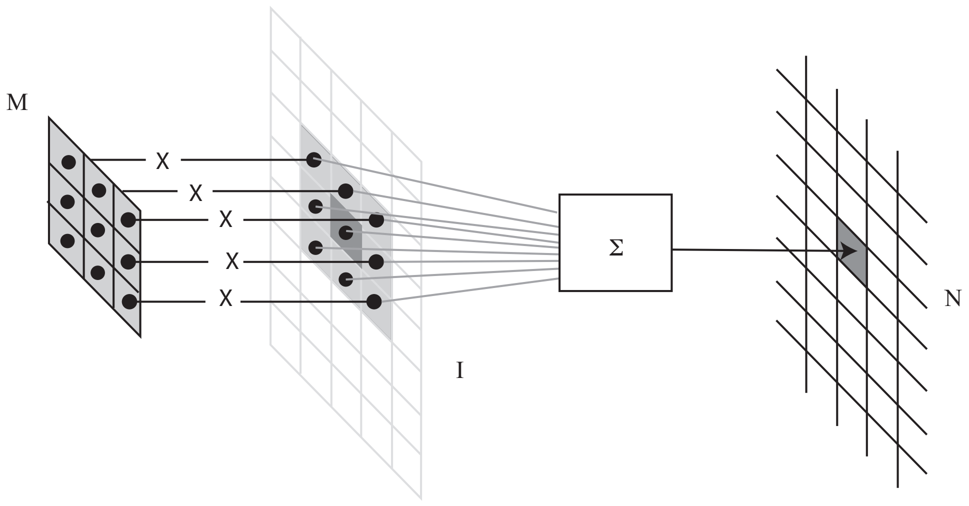

FIGURE 4.2: *To compute the value of $\mathcal{N} = \mathcal{W} * \mathcal{I}$. at some location, you shift a copy of $\mathcal{M}$ (the flipped version of $\mathcal{W}$) to lie over that location in $\mathcal{I}$; you multiply together the non-zero elements of $\mathcal{M}$ and $\mathcal{I}$ that lie on top of one another; and you sum the results. To compute the value of $\mathcal{N} = \texttt{filter}(\mathcal{I}, \mathcal{W})$ at some location, just omit flipping $\mathcal{W}$.*

4.1.2 Inconvenient Details

Now consider convolving an $M \times N$ image with a $2u + 1 \times 2v + 1$ kernel. Strips of the result of width v on the left and the right side and strips of height u at top and bottom contain values that are affected by pixels outside the image (Figure 4.1). Convolution could report only the values not affected by pixels outside the image – sometimes called the *valid region* of the convolution. This would turn an $M \times N$ image into an $M - 2u \times N - 2v$ image.

Attaching strips of width u on the left and right and height v on top and bottom would produce an image of size $M + 2u \times N + 2v$ – this is *padding*. Assuming the pixel values in these strips can be obtained somehow, convolving this padded image with the kernel would produce a $M \times N$ valid region. Padding like this is convenient, because there is no need to keep track of how much images have shrunk, but padding can have consequences (Section 41.2). Typically, API's make a variety of kinds of padding easy. One is *zero padding* (outside strips are zero); another is *reflection padding* (outside strips obtained by reflecting around the outer boundaries of the image).

The outer boundaries of an image mean that, in practice, convolution is not shift-invariant. This is because, in practice, shifting an image is not just a matter of relabelling pixels. If one (say) pans a camera, some pixels “fall off” one edge of the image *and are lost*, and other new pixels with *new values* appear at the other edge. Convolution cannot be invariant to this operation, because the value of the convolution cannot be computed for unknown pixels.

4.1.3 Convolution as Pattern Detection

You should think of the value of $\mathcal{N}_{ij}$ as a dot-product. To see this, flip $\mathcal{W}$ in both directions to form $\mathcal{M}$ and notice that

$$\begin{aligned}\mathcal{N} &= \mathcal{I} * \mathcal{W} \\ &= \texttt{filter}(\mathcal{I}, \mathcal{M})\end{aligned}$$

This means that you can think about convolution like this. To compute the value of $\mathcal{N}$ at some location, you place $\mathcal{M}$ (the flipped version of $\mathcal{W}$) at some location in the image; you multiply together the elements of $\mathcal{I}$ and $\mathcal{M}$ that lie on top of one another, ignoring everything in $\mathcal{I}$ outside $\mathcal{M}$; then you sum the results (Figure 4.2). Reindex the two windows to be vectors, and this is a dot product. This view explains why a convolution is interesting: it is a very simple pattern detector.

Recall that the dot-product of two unit vectors $\mathbf{u}$ and $\mathbf{v}$ is largest when they are the same, and smallest when $\mathbf{u} = -\mathbf{v}$. Using the dot-product analogy, for $\mathcal{N}_{ij}$ to have a large and positive value, the piece of image that lies under $\mathcal{M}$ must "look like" $\mathcal{M}$. Similarly, to have a large and negative value, the piece of image must "look like" $\mathcal{M}$, but with contrast reversed.

4.1.4 ReLU's

Write $\mathcal{W}$ for a kernel representing some pattern you wish to find. Assume that $\mathcal{W}$ has zero mean, so that the filter gives zero response to a constant image. Notice that $\mathcal{N} = \mathcal{W} * \mathcal{I}$ is strongly positive at locations where $\mathcal{I}$ looks like $\mathcal{W}$, and strongly negative when $\mathcal{I}$ looks like a contrast reversed (so dark goes to light and light goes to dark) version of $\mathcal{W}$. Usually, you would want to distinguish between (say) a light dot on a dark background and a dark dot on a light background. Write

$$\texttt{relu}(x) = \begin{cases} x & \text{for } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

(often called a *Rectified Linear Unit* or more usually *ReLU*). Then $\texttt{relu}\mathcal{W} * \mathcal{I}$ is a measure of how well $\mathcal{W}$ matches $\mathcal{I}$ at each pixel, and $\texttt{relu}{-\mathcal{W}} * \mathcal{I}$ is a measure of how well $\mathcal{W}$ matches a contrast reversed $\mathcal{I}$ at each pixel. The ReLU will appear again.

Figure 4.3 give some examples. The filters are shown on the far left, each in the top left hand corner of a field of zeros the same size as the image; this gives some sense of spatial scale. The lightest value is the largest value of the filter, the darkest is the smallest. The left two frames show the positive and negative components of the response to the filter. The positive responses occur where (rather roughly) the image "looks like" the filter. Similarly, negative responses occur where the image "looks like" a contrast reversed version of the filter. In that figure, Notice how the filters really are pattern detectors (the big dark blob gets responses from big dark blobs, and the small bright blob gets responses from small bright blobs), but they are not very good pattern detectors. Something that causes a bar filter to response will often also get a response from a blob filter. Further, the region of small bright leaves on the bottom of the image produces strong positive responses. The filter

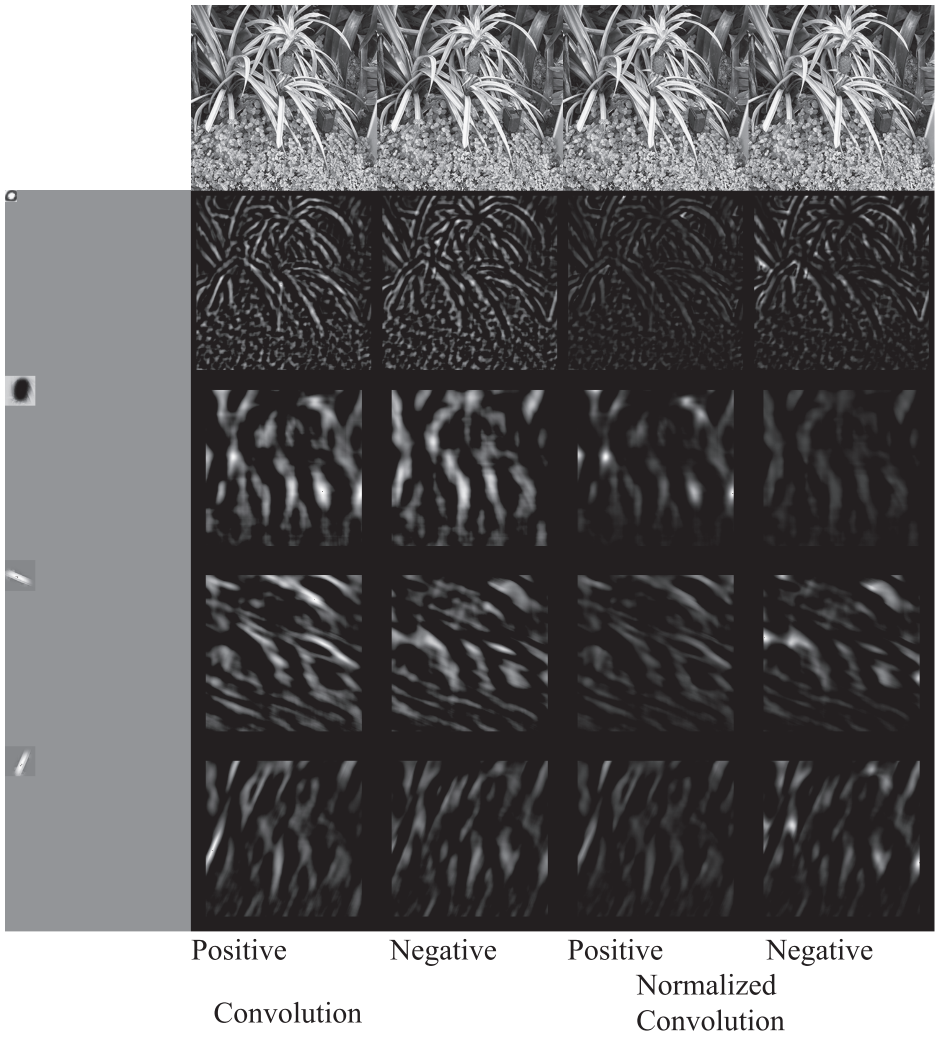

FIGURE 4.3: *Various zero-mean filters applied to a monochrome image of a pineapple plant (shown in the* **top row***, for reference), to show filters are simple pattern detectors. Details in the text.* Image credit: *Figure shows my photograph of a pineapple in the Singapore botanical garden.*

is linear, so bright patterns that don't look like the filter tend to give responses as strong as dark patterns that do. It can be useful to suppress small responses, and it is easy to do so by subtracting a small constant from the response before applying the ReLU (**exercises**).

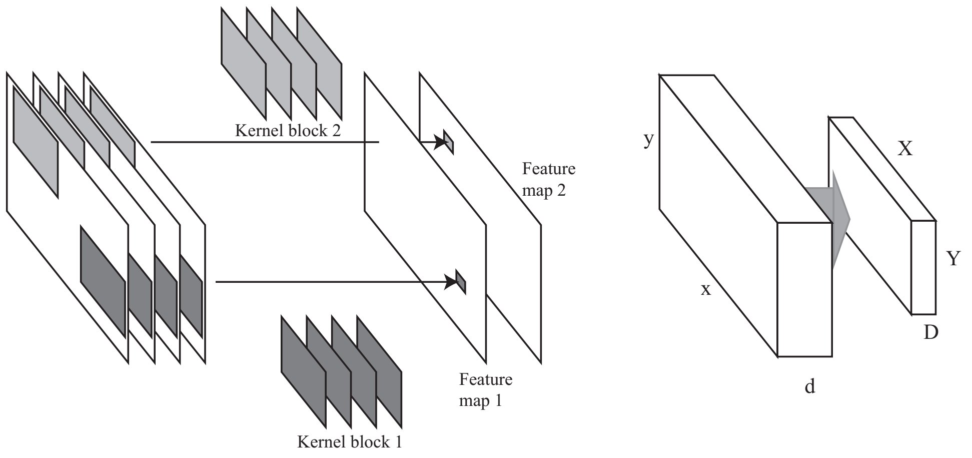

FIGURE 4.4: *On the* **left***, two kernels (now 3D, as in the text) applied to a set of feature maps produce one new feature map per kernel, using the procedure of the text (the bias term isn't shown). Abstract this as a process that takes an $x \times y \times d$ block to an $X \times Y \times D$ block (as on the* **right***).*

4.1.5 Normalized Convolution

The dot-product analogy reveals some reasons that convolution is not a particularly good pattern detector. Assume that the mean of the kernel is not zero. In this case, adding a constant offset to the image will change the value of the convolution, so you cannot rely on the value. This can be dealt with by subtracting the mean from the kernel.

If the mean of the kernel is zero, scaling the image will scale the value of the convolution. One strategy to build a somewhat better pattern detector is to normalize the result of the convolution to obtain a value that is unaffected by scaling the image. For $\mathcal{W}$ a zero mean kernel, $\mathcal{G}$ a gaussian kernel, and ϵ a small positive number compute

$$\frac{\mathcal{W} * \mathcal{I}}{\mathcal{G} * \mathcal{I} + \epsilon}.$$

Here the division is element by element, ϵ is used to avoid dividing by zero, and $\mathcal{G} * \mathcal{I}$ is an estimate of how bright the image is. This strategy, known as *normalized convolution* produces an improvement in the detector. Figure 4.3 compares normalized convolution to convolution. The right two frames show the positive and negative components of the normalized convolution (divide the filter responses by an estimate of image intensity). The normalized convolution is more selective. Responses are shown on a scale where zero is dark and a strong response is light. It is now more usual to manage these difficulties by learning kernels that behave well (Section 41.2).

4.1.6 Multi-Channel Convolution

The description of convolution anticipates monochrome images, and Figure 4.3 shows filters applied to a monochrome image. Color images are naturally 3D objects with two spatial dimensions (up-down, left-right) and a third dimension that chooses a slice or *channel* (R, G or B for a color image). Color images are sometimes called *multi-channel images*. Multi-channel images offer a natural for representations of image patterns, too — two dimensions that tell you where the pattern is and one that tells you what it is. For example, the results in Figure 4.3 can be interpreted as a block consisting of eight channels (four patterns, original contrast and contrast reversed). Each slice is the response of a pattern detector *for a fixed pattern*, where there is one response for each spatial location in the block, and so are often called *feature maps* (it is entirely fair, but not usual, to think of an RGB image as a rather uninteresting feature map).

For a color image $\mathcal{I}$, write $\mathcal{I}_{k,ij}$ for the k'th color channel at the i, j'th location, and $\mathcal{K}$ for a color kernel – one that has three channels. Then interpret $\mathcal{N} = \mathcal{I} * \mathcal{K}$ as

$$\mathcal{N}_{ij} = \sum_{kuv} \mathcal{I}_{k,i-u,j-v}\mathcal{K}_{kuv}$$

which is an image with a single channel. This $\mathcal{N}$ is a single channel image that encodes the response to a single pattern detector. Much more interesting is an encoding of responses to multiple pattern detectors, and for that you must use multiple kernels (often known as a *filter bank*). Write $\mathcal{K}^{(l)}$ for the l'th kernel, and obtain a feature map

$$\mathcal{N}_{l,ij} = \sum_{kuv} \mathcal{I}_{k,i-u,j-v}\mathcal{K}^{(l)}_{kuv}.$$

This notation is quite clunky, because it isn't a three dimensional convolution (look at the directions of the indices). This never matters for our purposes. Another clunky feature of the notation is that applying the same kernel to each layer of a color image requires a fairly odd set of kernels (**exercises**). It has two enormous virtues. First, convolution can be used to detect colored patterns (Figure **??**). Second, convolution becomes an operation that turns a three dimensional object – a stack of channels, a multi-channel image or a feature map, according to taste – into another such object, so you can apply a convolution to the results of a convolution.

4.2 APPLICATIONS

4.2.1 Image Gradients with Finite Differences

For an image $\mathcal{I}$, the gradient is

$$\nabla \mathcal{I} = (\frac{\partial \mathcal{I}}{\partial x}, \frac{\partial \mathcal{I}}{\partial y})^T,$$

which we could estimate by observing that

$$\frac{\partial \mathcal{I}}{\partial x} = \lim_{\delta x \to 0} \frac{\mathcal{I}(x+\delta x, y) - \mathcal{I}(x,y)}{\delta x} \approx \mathcal{I}_{i+1,j} - \mathcal{I}_{i,j}.$$

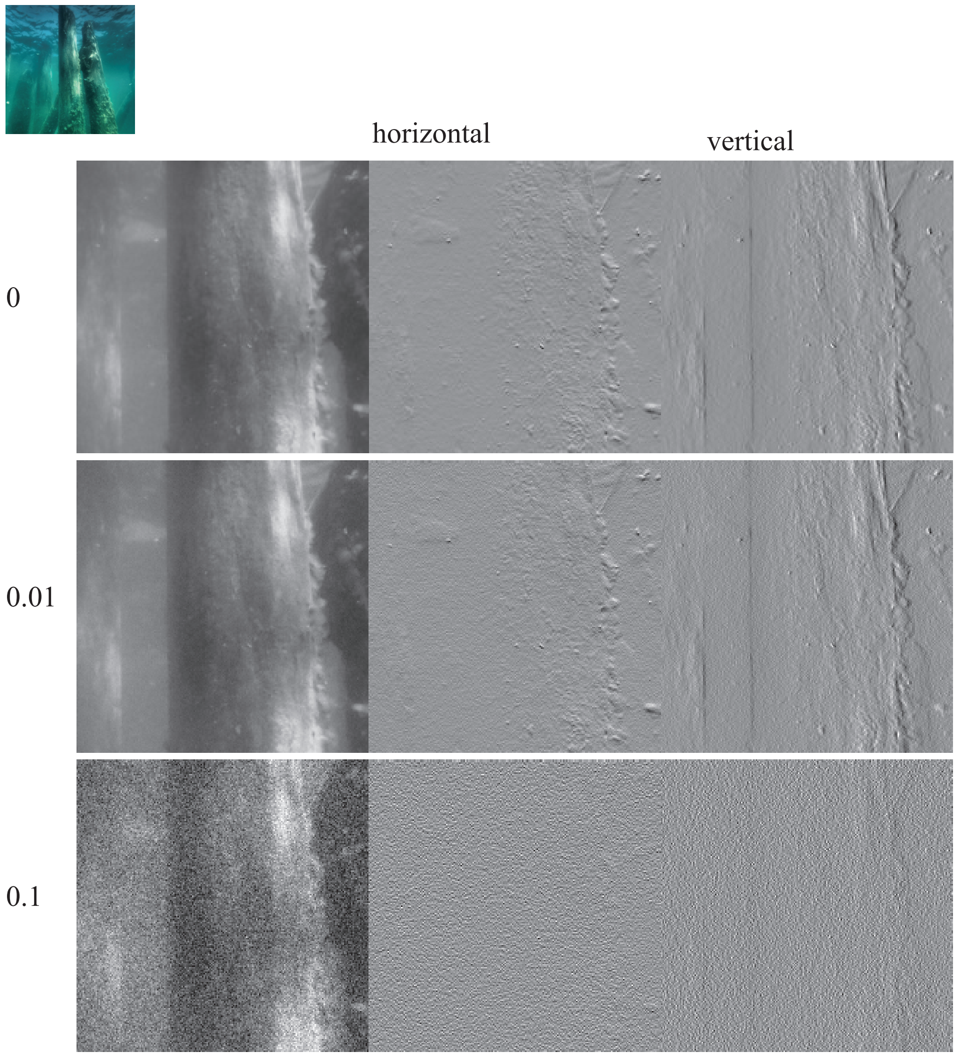

FIGURE 4.5: *Finite differences yield reasonable derivative estimates, but are strongly affected by noise.* **Top left** *shows the original image, from which a detail window is extracted and turned monochrome.* **Rows** *show image, horizontal derivative and vertical derivative, where derivatives are estimated by finite differences.* **First row** *is noise free image; others have additive Gaussian noise added, with standard deviation shown. Notice how this noise affects derivatives. The derivatives are scaled so that positive values are bright, negative values are dark, and 0 is mid-range. However the scale is chosen* per row, *which means the figure understates the effect of noise. In the noisy rows, the largest magnitude derivatives are much larger than in the clean row, which is why you can hardly see significant derivatives in the bottom row.* Image credit: *Figure shows Robert Forsyth's photograph of historical dock pilings in Lake Michigan.*

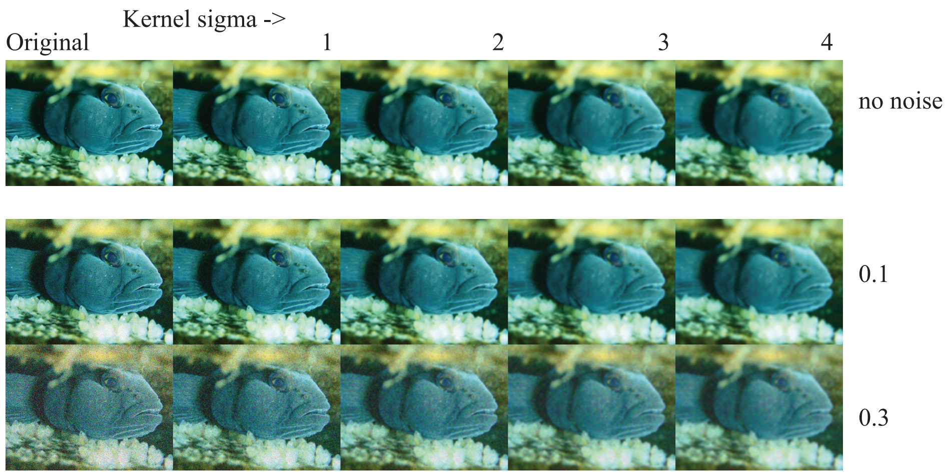

FIGURE 4.6: *Smoothing an image with a gaussian kernel is an effective way to suppress additive Gaussian noise.* **Left column** *the original image, followed by versions smoothed with a gaussian kernel with $\sigma = 1$, $\sigma = 2$, $\sigma = 3$ and $\sigma = 4$.* **Top row** *shows results on a noise free image;* **middle row** *shows results on an image with additive stationary gaussian noise with standard deviation 0.01, where the value of a pixel ranges from 0 to 1;* **bottom row** *shows results on an image with additive stationary gaussian noise with standard deviation 0.1. Notice how (a) smoothing blurs the original image; (b) more smoothing leads to more blur; (c) smoothing suppresses noise (so a smoothed version of a noisy image is close to the smoothed version of the original); and (d) more smoothing suppresses more noise.* Image credit: *Figure shows Robert Forsyth's photograph of a goby on its nest in Lake Michigan.*

This means a convolution with

$$-1 \quad 1$$

will estimate $\partial\mathcal{I}/\partial x$ (nothing in the definition requires convolution with a square kernel). Notice that this kernel "looks like" a dark pixel next to a light pixel, and will respond most strongly to that pattern. By the same argument, $\partial\mathcal{I}/\partial y \approx \mathcal{I}_{i,j+1} - \mathcal{I}_{i,j}$. These kinds of derivative estimates are known as *finite differences.* most unsatisfactory estimate of the derivative. This is because finite differences respond strongly (i.e., have an output with large magnitude) at fast changes, and fast changes are characteristic of noise. Roughly, this is because image pixels tend to look like one another. For example, if we had bought a discount camera with some pixels that were stuck at either black or white, the output of the finite difference process would be large at those pixels because they are, in general, substantially different from their neighbors. All this suggests that some form of noise suppression is appropriate before differentiation.

4.2.2 Image Noise Models

The simplest model of image noise is the *additive stationary Gaussian noise* (or *Gaussian noise*) model, where each pixel has added to it a value chosen independently from the same normal (Gaussian – same Gauss, different sense) probability distribution. This distribution almost always has zero mean. The standard deviation is a parameter of the model. Figure 4.6 shows some examples of additive stationary Gaussian noise.

Images can be quite effectively denoised because "pixels look like their neighbors". This important and very reliable slogan is in scare quotes because, while it is an extremely important practical guide, making it precise is neither easy nor particularly useful. Generally, pictures show objects which are span a large number of pixels, and where the shading changes relatively slowly over the surface of the object. This means that the value at a pixel is likely to be close to the value at its neighbor. Although this isn't true of every pixel – otherwise there wouldn't be edges in images – it is true of most pixels. If you have a pixel whose value is unknown, looking at its neighbors will almost always yield a good estimate. A pixel that does not look like its neighbors is suspect.

4.2.3 Gaussian Smoothing to Suppress Noise

The downsampling strategy of Section 2.3.3 involves first forming a weighted average of a window centered on each pixel then downsampling the image. Here, for a $2k-1 \times 2k-1$ window, the weights are:

$$w_{i,j} = \frac{\exp - \left(\frac{(i-k)^2+(j-k)^2}{2\sigma^2} \right)}{C}$$

where C is chosen so the weights sum to one. Check that forming the weighted averages is equivalent to convolving an image with a kernel with the given weights. This procedure is called *gaussian smoothing* or very often just *smoothing*. It turns out that this procedure is very good at suppressing many kinds of image noise. Figure 4.6 shows examples of suppressing additive Gaussian noise, and the **exercises** explore some details. Gaussian smoothing can suppress the effects of other noise processes, too (Figure 4.7).

The choice of σ (or scale) for the Gaussian follows from the following considerations:

- If the standard deviation of the Gaussian is very small—say, smaller than one pixel—the smoothing will have little effect because the weights for all pixels off the center will be very small.
- For a larger standard deviation, the neighboring pixels will have larger weights in the weighted average, which in turn means that the average will be strongly biased toward a consensus of the neighbors. This will be a good estimate of a pixel's value, and the noise will largely disappear at the cost of some blurring.
- Finally, a kernel that has a large standard deviation will cause much of the image detail to disappear, along with the noise.

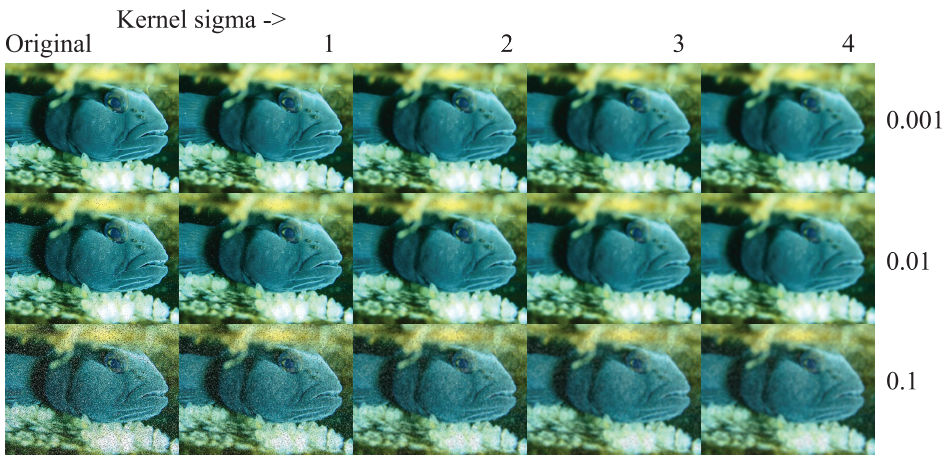

FIGURE 4.7: *Images at various noise levels smoothed with various gaussian kernels. The noise here involves picking pixel locations uniformly at random in the image, then flipping them either full light or full dark. The number on the* **far right** *shows the probability of a pixel being flipped (so at* 0.001*, a* 30×30 *window should have about one flipped pixel in it; at* 0.01*, a* 10×10 *window should have about one flipped pixel in it; and at* 0.1*, a* 3×3 *window should have one flipped pixel in it.* **Left** *the original image, followed by versions smoothed with* $\sigma = 1$*,* $\sigma = 2$*,* $\sigma = 3$ *and* $\sigma = 4$*. Notice how (a) smoothing blurs the original image; (b) more smoothing leads to more blur; (c) smoothing suppresses noise (so a smoothed version of a noisy image is close to the smoothed version of the original); and (d) more smoothing suppresses more noise. The noise-free image is top left in Figure 4.6.* Image credit: *Figure shows Robert Forsyth's photograph of a goby on its nest in Lake Michigan.*

4.2.4 The Median Filter

Most image noise tends to result in pixels not looking like their neighbors. However, gaussian smoothing is not always effective at estimating the true value of noisy pixels. For example, look closely at Figure 4.7. The noise process – a *Poisson noise process*, sometimes called *salt and pepper noise* – picks pixel locations uniformly at random in the image, then flips the result either full light or full dark. This means that a noisy pixel contains no information, and might be very different from its neighbors. If you compute a weighted average in a region that contains a noisy pixel, that weighted average might be severely disrupted by the noise, even if the center is a clean pixel. For example, think of a dark neighborhood on the goby where noise has turned one pixel bright – the bright pixel will dominate the average unless it contains a very large number of pixels with quite large weights. And in that case, the image will be blurry.

This suggests the entirely natural alternative of computing a median in a neighborhood as an estimate of the value at a pixel. As Figure **??** shows, this can be very effective at suppressing noise. Notice an attractive feature of the median filter – it tends not to blur edges, even when it strongly smoothes the interior of

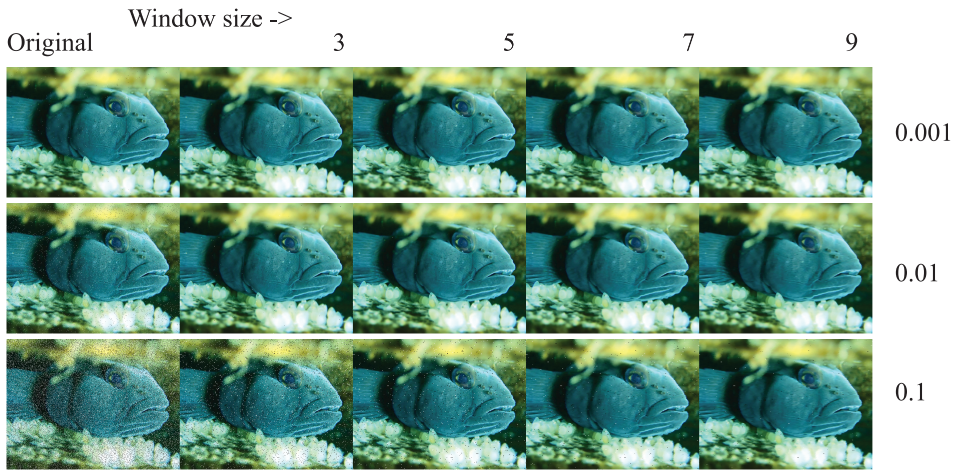

FIGURE 4.8: *Images at various noise levels smoothed with a median filter. The noise here involves picking pixel locations uniformly at random in the image, then flipping them either full light or full dark. The number on the* **far right** *shows the probability of a pixel being flipped (so at* 0.001*, a* 30×30 *window should have about one flipped pixel in it; at* 0.01*, a* 10×10 *window should have about one flipped pixel in it; and at* 0.1*, a* 3×3 *window should have one flipped pixel in it.* **Left** *the original image, followed by versions where the median is taken in windows of different sizes. Notice how (a) the median filter preserves edges rather well, even over big windows; (b) bigger windows lead to more noise suppression; and (c) texture details are suppressed by the median, with bigger windows suppressing more.* Image credit: *Figure shows Robert Forsyth's photograph of a goby on its nest in Lake Michigan.*

image regions. Median filters are somewhat more expensive computationally than smoothing, but deal fairly well with additive gaussian noise as well as salt and pepper noise (Figure 4.9)

4.2.5 Representing Images with Filter Banks

In the image in Figure 4.3, the leaves of the pineapple plant look like disorganized thick stripes. The leaves of the plant at its base are quite different, and look more like repeated small spots. These are examples of textures – somewhat unstructured patterns that are quite characteristic. Textures are widespread and quite distinctive – a field of pebbles looks quite different from a stand of corn; a cluster of pine needles looks very different from an expanse of bark; and so on.

Figure 4.3 also suggests a way to represent textures, and so images. Think of a texture as a collection of small patterns, arranged in some distinctive way. An image region showing a field of pebbles would have many spots, some small, some large and most medium, but very few thin bars. In contrast, an image region showing a cluster of pine needles would have many thin bars, pointing in about the

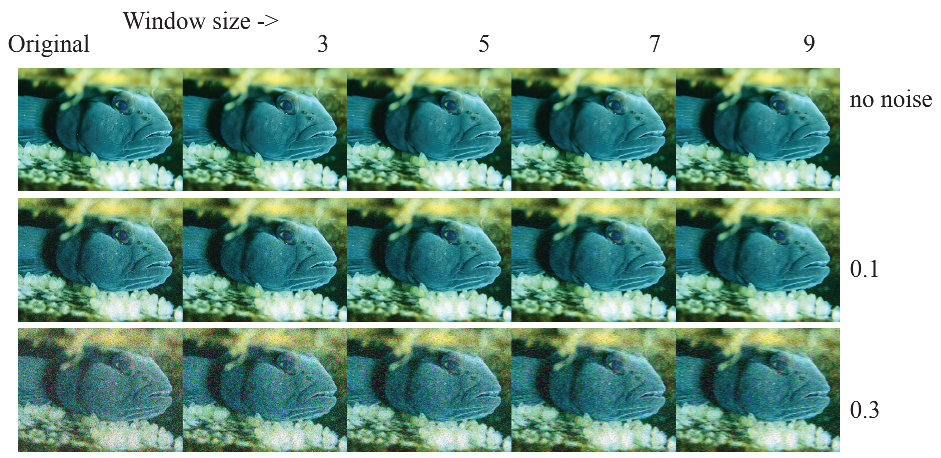

FIGURE 4.9: *Images at various noise levels smoothed with a median filter. The noise here is additive Gaussian noise.* **Left** *the original image, followed by versions where the median is taken in windows of different sizes.* **Top row** *shows results on a noise free image;* **middle row** *shows results on an image with additive stationary gaussian noise with standard deviation 0.01 (where the value of a pixel ranges from 0 to 1);* **bottom row** *shows results on an image with additive stationary gaussian noise with standard deviation 0.1. Notice how (a) the median filter preserves edges rather well, even over big windows; (b) bigger windows lead to more noise suppression; and (c) texture details are suppressed by the median, with bigger windows suppressing more.* Image credit: *Figure shows Robert Forsyth's photograph of a goby on its nest in Lake Michigan.*

direction, but very few small or large spots. Then to build an image representation: (a) construct a vocabulary of patterns; (b) find out which patterns are present at which pixel; and then (c) building a summary of which patterns are present in a region.

Because the patterns are likely so variable, an elaborate or detailed pattern detector is likely to be unhelpful – something that precisely detects a pine needle would need to be tuned to exactly the right angle, which would be a nuisance – so it is natural to use filters as pattern detectors. However, it is helpful to distinguish between, say, light thin bars on dark backgrounds (possible pine needles) and dark thin bars on light backgrounds (possible gaps between needles).

For the moment, assume the vocabulary of patterns is given, represented as a filter bank. Then the next two steps are straightforward. To find the patterns in an image, construct the response of all the filters at all points and apply a ReLU. Stack these responses into a multi-channel image. To compute a summary, construct a local weighted average of each channel of the multi-channel image at each pixel.

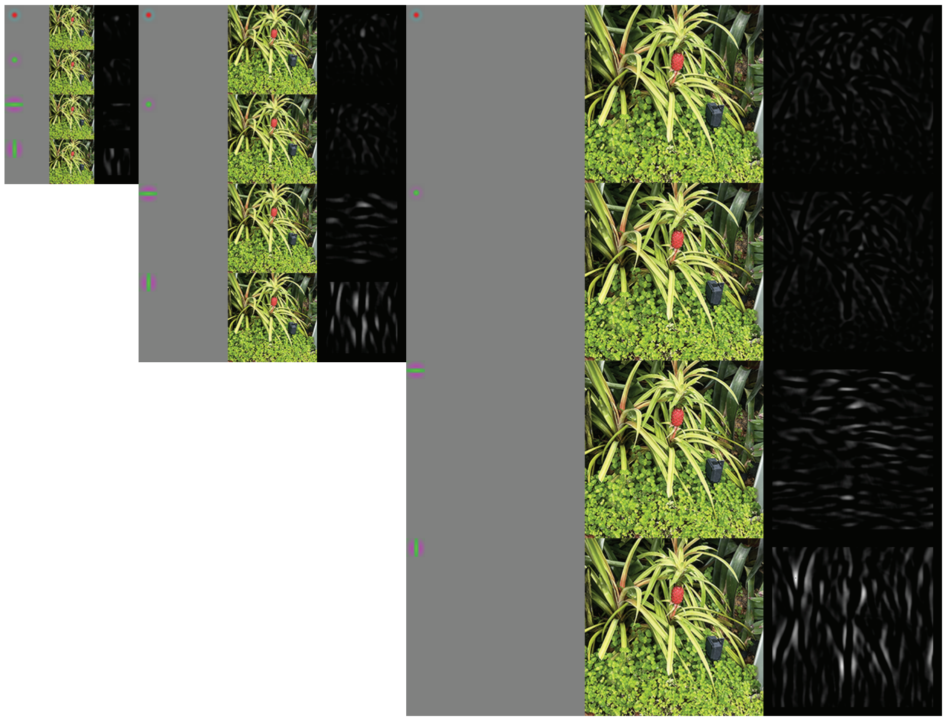

FIGURE 4.10: Image credit: *Figure shows my photograph of a pineapple in the Singapore botanical garden.*