CHAPTER 2

Upsampling, Smoothing and Downsampling

2.1 IMAGES AS SAMPLED FUNCTIONS

Your first encounter with an image as something to compute with (rather than look at) is likely as an array for an intensity image, or set of three arrays for a color image. Knowing how the image ended up in this form is important if you want to interpret it. A quite detailed model of the geometry and physics underlying images appears in Part IX. A simple model will have to do for the moment.

The image you see as three arrays starts as a spectral energy field – energy E as a function of position $\mathbf{X}$ in 3D, direction ω and wavelength λ, so $E(\mathbf{x}, \omega, \lambda)$. This energy field is created by light leaving light sources, reflecting from surfaces, and eventually arriving at the entrance to the camera (Figure 2.1). This is usually but not always a lens. Various processes in lens and camera map some of the light that arrives to some *sensor* at the back of a camera. The sensor is made up of a grid of *receptors*, each of which transduces the energy that arrives into a number (or some numbers). Each receptor on the sensor corresponds to a single *pixel* (or spatial location) in the array that is read from the camera.

The lens arranges that light arriving at $\mathbf{x}$ on the sensor all arrived from one point on a surface in 3D ($\mathbf{X}$ in Figure 2.1). The vast majority of sensors in current use are linear, so doubling the amount of light arriving at the camera will double the output. The pixel at i, j on the grid is a *sample* of a function of position (Figure 2.2).

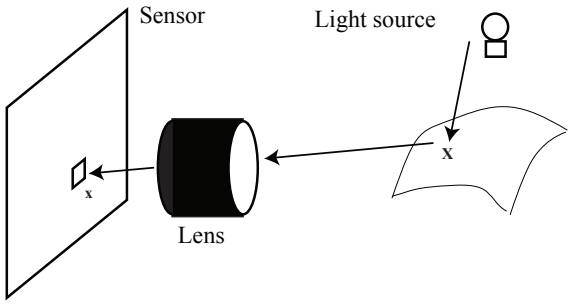

FIGURE 2.1: *A high-level model of imaging. Light leaves light sources and reflects from surfaces. Eventually, some light arrives at a camera and enters a lens system. Some of that light arrives at a photosensor inside the camera.*

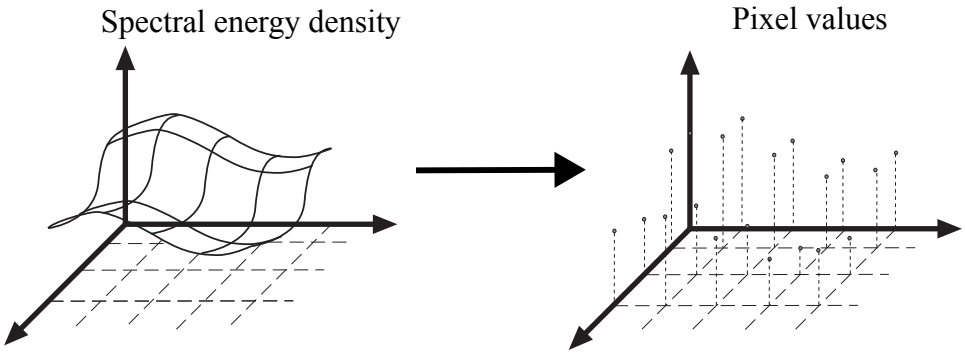

FIGURE 2.2: *Because each pixel in the sensor averages over a small range of directions and positions, the process mapping the input spectral energy distribution to pixel values can be thought of as sampling. On the* **left**, *is a representation of the energy distribution as a continuous function of position. The value reported at each pixel is the value of this function at the location of the pixel (***right***).*

2.1.1 Color Images

Humans see color by comparing the response of different kinds of photoreceptor at nearby locations (Chapter **??**). The main difference between these kinds of photoreceptor is in the sensitivity of the sensor with wavelength. Roughly, one type of sensor responds more strongly to longer wavelengths, another to medium wavelengths, and a third to shorter wavelengths (there are other kinds of sensor, and other differences).

Cameras parallel this process. The sensors used for the R (or red) layer of an RGB image respond more strongly to longer wavelengths; for the G (or green) layer, to medium wavelengths; and the B (or blue) to shorter wavelengths. Cameras must be engineered to produce the response of three different types of sensor *at the same place*. The usual strategy is to use one imaging sensor, and arrange that different pixels respond differently to wavelength. Typically, there are three types of pixel (R, G, and B), interleaved in a *mosaic* (Figure 2.3). This means that at many locations the camera does not measure R (or G, or B) response, and it obtains a value by interpolation. Generally, mosaic patterns have more G pixels than R or B pixels. This is because G pixels are sensitive to a wider range of visible wavelengths than R and B pixels, and so the interpolation yields better results. Regular mosaic patterns can create effects in images, and there are *demosaicing* algorithms to remove these effects. An alternative is to use three imaging sensors and arranging for each sensor to receive the same light (lenses, mirrors, that sort of thing). Such *multiccd cameras* tend to be larger, heavier and more expensive than single sensor cameras.

2.2 UPSAMPLING AND IMAGE INTERPOLATION

To *upsample* an image you increase the number of pixels in a grid. Some cases are easy. To go from, say a 100×100 image to a 200×200 image, you could simply

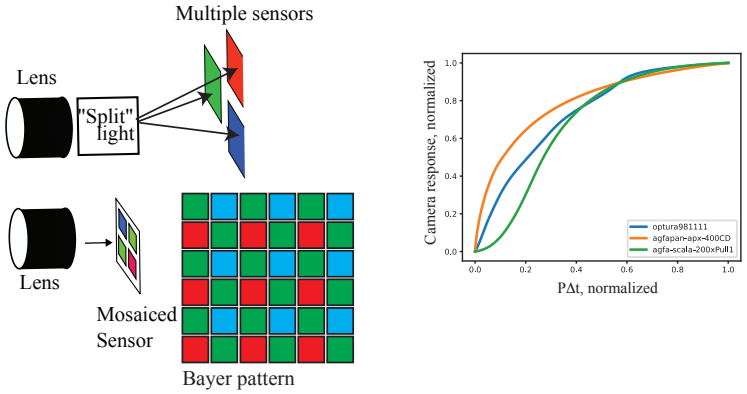

FIGURE 2.3: *There are two main ways to obtain color images. One can (as in* **top left***) build a multiccd camera with three imaging sensors. Each has a different response to wavelengths. The cheaper and lighter alternative is to use one imaging sensor (***bottom left***) but have a mosaic of pixels with different responses. This can be achieved by placing a small filter on each sensor location.* **Right** *shows one traditional such pattern of filters, a Bayer pattern. Camera response functions for three different cameras, plotted from the comprehensive dataset available at* `https://cave.cs.columbia.edu/repository/DoRF`*. The horizontal axis is the "input" – the $P\Delta t$ observed by the camera, scaled to $0-1$. The vertical axis is the "output" – the response of the camera, again scaled to $0-1$. Notice that locations that would be quite dark for a linear sensor will be lighter; but as the linear sensor gets very bright, the output recorded by the camera grows slowly. This means that the range of outputs is smaller than the range of inputs, which is helpful for practical cameras. This response function is typically located deep in the camera's electronics. Typical consumer cameras apply a variety of transforms before reporting an image, though one can often persuade cameras to produce an untransformed, linear response image (a RAW file).*

replace each pixel with a 2×2 block of pixels, each having the same value as the original. This isn't a particularly good strategy, and the resulting images tend to look "blocky" (try it!). But upsampling by a factor that isn't an integer is more tricky.

Consider going from 100×100 to 127×127. One way to do this is to duplicate 27 rows, then duplicate 27 columns in the result; to do so requires determining which columns to duplicate. You might consider scanning the source (smaller - $\mathcal{S}$) image and, for each pixel, determining where it goes in the target (larger - $\mathcal{T}$) image. But there are more pixels in the target than in the source, so this approach must lead to holes in the predicted image.

The correct alternative is to scan the target image and, for each pixel, determine what value it should receive. This is known as *inverse warping*. In the example, the i, j'th location of $\mathcal{T}$ must get the value of the $i/1.27$, $j/1.27$'th location of $\mathcal{S}$. In fact, most values requred are at locations that are not integer values. To produce these values, construct a continuous function out of the image, then

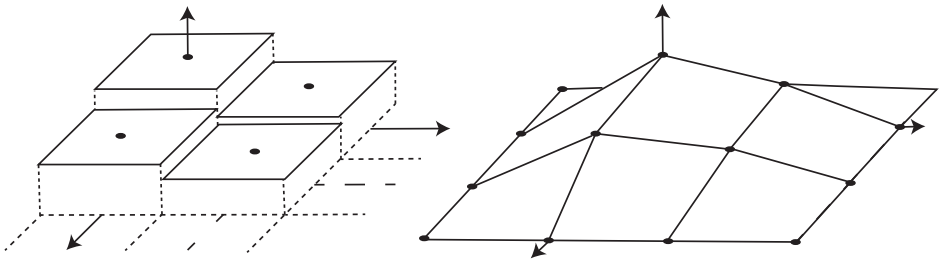

FIGURE 2.4: *On the* **left**, *a function interpolating a* 2×2 *image using nearest neighbors. The dashed lines pass through grid points, and the dotted lines are halfway between grid points. The function is zero away from the four boxes shown. Image values are shown as filled circles. On the* **right**, *a bilinear interpolate of the same data.*

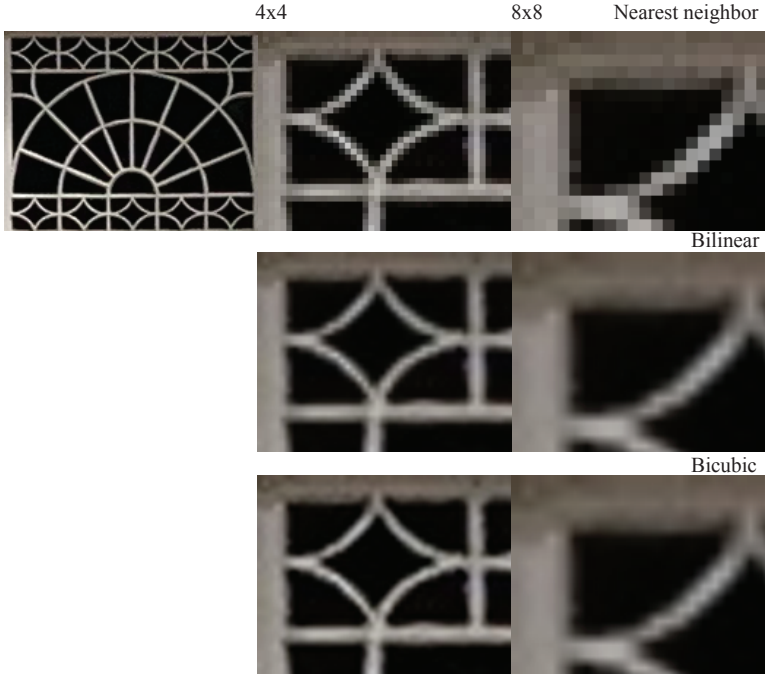

FIGURE 2.5: *The choice of interpolate when upsampling can make a real difference.* **Top left** *shows a detail from a picture. I have upsampled the image, then cropped the upsamples (showing the top left corner) and zoomed them so you can see the details.* **Center column** *shows a cropped 4x4 upsample using three different interpolation methods and* **right column** *shows 8x8 upsamples by various methods. Notice the significant blockiness in nearest neighbor interpolates (***top row***). Bilinear interpolates (***second row***) are much better, and bicubic interpolates (***third row***) are different to bilinear interpolates, but not a major improvement.* Image credit: *Figure shows my photograph of a facade in Stellenbosch.*

evaluate that function at the (likely non-integer) points. This procedure is known as *interpolation* and the function – the *interpolate* – (a) must have the same value as the original image at the original integer grid points (b) can be evaluated at any point rather than just the integer grid points. Write $\mathcal{I}(x, y)$ for an interpolate of an image $\mathcal{I}$.

The simplest interpolate is *nearest neighbors* – take the value at the integer point closest to location whose value you want. Break ties by rounding up, so you would use the value at $2, 2$ if you wanted the value at $1.5, 1.5$. As Figure 2.4 shows, this strategy has problems – the upsampled image looks blocky.

Writing nearest neighbors in a different way can be informative. For nearest neighbors, define

$$b_{nn}(u, v) = \begin{cases} 1 & \text{for } -1/2 \leq u < 1/2 \text{ and } -1/2 \leq u < 1/2 \\ 0 & \text{otherwise} \end{cases}$$

which has the convenient property that $b_{nn}(0, 0) = 1$, but $b_{nn} = 0$ for every other set of integer coordinates. The fitted function is

$$\mathcal{I}(x, y) = \sum_{i,j} \mathcal{I}_{ij} b_{nn}(x - i, y - j).$$

and it is a simple exercise to show that it has the properties required for an interpolate. This fitted function looks like a collection of boxes, and is not continuous (Figure 2.4).

Most widely used is *bilinear interpolation*. For this, construct a function

$$b_{bi}(u, v) = \begin{cases} (1-u)(1-v) & \text{for } 0 < u \leq 1 \text{ and } 0 < v \leq 1 \\ u(1-v) & \text{for } -1 \leq u \leq 0 \text{ and } 0 < v \leq 1 \\ uv & \text{for } -1 \leq u \leq 0 \text{ and } -1 \leq v \leq 0 \\ (1-u)v & \text{for } 0 < u \leq 1 \text{ and } -1 \leq v \leq 0 \\ 0 & \text{otherwise} \end{cases}$$

which is continuous, and again has the convenient property that $b_{bi}(0, 0) = 1$, but $b_{nn} = 0$ for every other grid point (and looks a bit like a hat). The interpolate is

$$\mathcal{I}(x, y) = \sum_{i,j} \mathcal{I}_{ij} b_{bi}(x - i, y - j).$$

and it is a simple exercise to show that it has the properties required for an interpolate. Notice that this interpolate is continuous (Figure 2.4) and has a variety of interesting properties (**exercises**).

The basis function construction above is a good way to think about interpolation (and can be used to build more complicated interpolates, **exercises**), but it is not the best way to think about bilinear interpolation. To find a value for $\mathcal{I}(i + \delta i, j + \delta j)$, where i and j are integers; $0 < \delta i < 1$; and $0 < \delta j < 1$, use

$$\mathcal{I}(i + \delta i, j + \delta j) = \begin{bmatrix} \mathcal{I}_{ij}(1 - \delta i)(1 - \delta j) + \\ \mathcal{I}_{i+1,j}(\delta i)(1 - \delta j) + \\ \mathcal{I}_{i,j+1}(1 - \delta i)(\delta j) + \\ \mathcal{I}_{i+1,j+1}(\delta i)(\delta j) \end{bmatrix}.$$

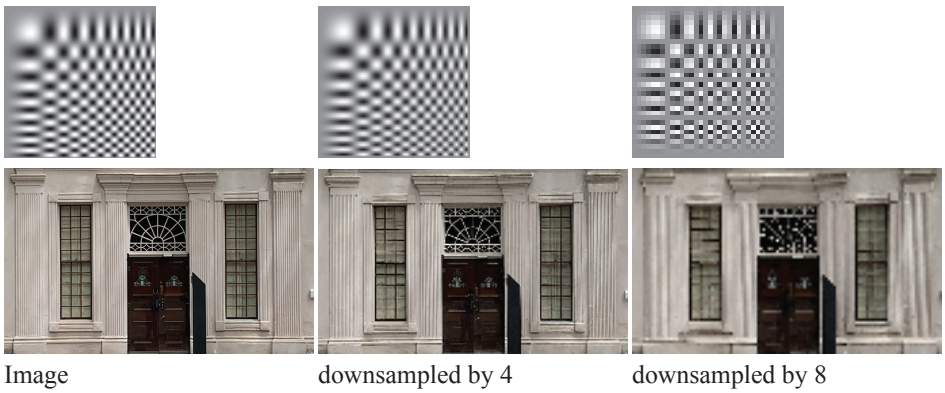

FIGURE 2.6: *Downsampling by just taking every k'th pixel in each direction reliably leads to problems. The* **top row** *shows some effects on a stylized image, and the* **bottom row** *shows results on a real photograph. The* **left** *image is the original;* **center** *is a downsampled image obtained by taking every 4'th pixel, then printing the image with larger pixels;* **right** *the original downsampled by taking every 8'th pixel. Notice how detail is lost in the resampling process. For the stylized image, some small boxes disappear (look on the edges of the image); others turn into large boxes (lower right quarter of the downsampled by 8 image). For the real image, notice the behavior of the details in the window above the door, and on either side of the door.* Image credit: *Figure shows my photograph of a facade in Stellenbosch.*

It is an exercise to check that this formula yields the value that the basis function approach would yield. By a little manipulation, you can show that this procedure boils down to: predict a value for $\mathcal{I}(i+\delta i, j)$ using a linear interpolate; predict a value for $\mathcal{I}(i+\delta, j+1)$ using a linear interpolate; now linearly interpolate between these two to get a value for $\mathcal{I}(i+\delta i, j+\delta j)$. Modern hardware is particularly efficient at bilinear interpolation, and any reasonable software environment will be able to do this for you.

The choice of interpolate can make a real difference to the quality of the result (Figure 2.5). More complicated interpolation procedures are possible. In *bicubic interpolation*, the interpolate is cubic in δx and δy and depends on other neighboring pixels (**exercises**). Again, any reasonable software environment will be able to do this for you. While this procedure is more complicated and slower, in some applications the small improvements are justified. One occasionally important difference between bicubic interpolation is that for a bilinear interpolate, the local maxima are always at grid points, but for a bicubic interpolate, they may not be (**exercises**). Constructing more complicated interpolates is straightforward (any function b such that $b(0,0) = 1$ and b is zero at every other grid point will do it; **exercises**) but seldom worthwhile. Another application for interpolation is demosaicing: one could interpolate, and then sample the interpolating function. The interpolation procedures above need some minor adjustments because the unknown values are at grid points (details in **exercises**).

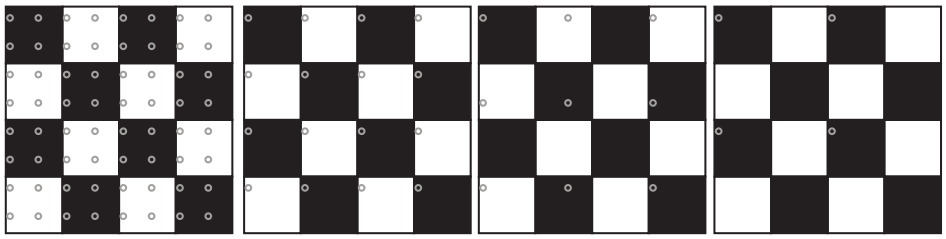

FIGURE 2.7: *A visualization of how sampling problems arise. The underlying image is a checkerboard, which is sampled at each of the circles. The checkerboards on the* **left** *and* **center left** *illustrates a sampling procedure that appears to be successful. Whether it is or not depends on some details that we will deal with later – but the count of checks will be correct in each case. The sampling procedures shown on the* **center right** *and* **right** *are unequivocally unsuccessful. The samples suggest that there are fewer checks than there are in the original patterns. This illustrates two important phenomena: first, a successful sampling scheme must sample data often enough; and second, unsuccessful sampling schemes cause high-frequency information to appear as lower-frequency information. For example, on the* **right**, *the sampling procedure represents a checkerboard as a single dark region.*

2.3 DOWNSAMPLING AND SMOOTHING

2.3.1 Aliasing: Errors Caused by Downsampling

Sampling a function can produce something that represents the function very poorly indeed. This is most apparent when you *downsample* an image – reduce its size in each dimension. To see this, take an image whose dimensions are divisible by two (or four, or eight, and so on) then halve (or quarter, and so on) the size. To do this, you can simply take every second (fourth, eighth, and so on) pixel in each direction. Figure 2.6 shows effects that occur when you downsample by an integer number of pixels. Fine details can disappear or worse turn into coarse details. Figure 2.7 sketches a partial explanation — if there are too few samples, patterns in the image can fall between the samples.

Downsampling by an amount that isn't an integer is straightforward. Just like upsampling, the correct procedure is to scan the target image and, for each pixel, determine what value it should receive using interpolation. The errors produced by downsampling are not the result of interpolation, though a better choice of interpolate can help. The general term for the kind of errors seen here is *aliasing*. In Chapter 41.2, we will be much more precise about these issues.

As Figure 2.7 illustrates, the key question is how many samples you draw compared to how much detail there is in the function you are sampling. The figure suggests a rough explanation for what is going wrong when one subsamples an image. Samples might be poorly aligned with the underlying data, and so misrepresent it.

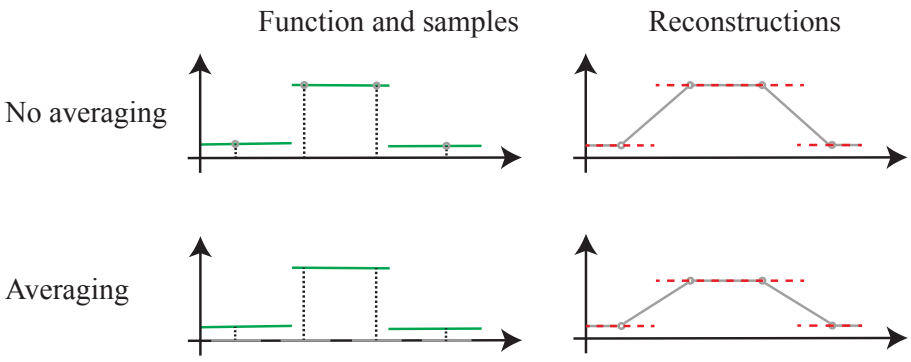

FIGURE 2.8: *Averaging can improve the representation produced by sampling. The* **top row** *shows an example of a simple function (a set of checks in one dimension,* **green***), sampled at four points (the vertical* **dotted** *lines show the locations of the samples). On the* **top right**, *two reconstructions of the function using interpolation. The* **red** *shows a nearest neighbors reconstruction, and the* **gray** *a linear reconstruction. Note how the center check has been made wider because there are two samples on it. The* **bottom row** *shows the effect of averaging. The average is taken over the* **gray** *regions on the horizontal axis.* **Bottom left** *shows the reconstructions. While the center check is still too wide, averaging has reduced its height. The representation is somewhat improved.*

2.3.2 Smoothing

The downsampler needs to compute a value for the target image at i, j. This location corresponds to the location u, v in the source image (so, for example, in downsampling by two, $u = 2i$ and $v = 2j$). Call the point u, v the query point. Using the (possibly interpolated) value of the source image at this location may not be a particularly good idea, because there might be an important detail close to, but not at, the query point. An alternative is to use an average of the source image function about the query point.

In the easiest case, downsample an image by a factor of two. At every second pixel location in each direction, compute (say) an average of the $(2k+1) \times (2k+1)$ window of pixels centered at that location and report that average rather than the pixel value. A simple argument suggests that this should help: now the value of the pixel in the subsampled image is affected by its neighbors in the original image, so details that were missed by just taking every second pixel have a chance to appear in the result. Figure 2.8 is a picture of this argument applied to a function in one dimension (a case that is easier to draw).

2.3.3 Gaussian Smoothing

As Figures 2.8 and 2.10 show, just averaging nearby values helps, because small structures that might otherwise have been missed will contribute to the downsam-

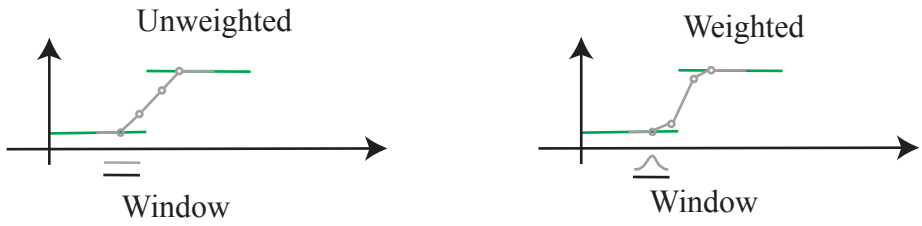

FIGURE 2.9: *Sampling with a weighted average makes significant changes in the representation. On the* **left**, *sampling using an average that is not weighted. The sample points are the centers of the* **gray** *circles, and the* **green** *shows the function.* **Bottom left**, *the* **gray** *shows a linear interpolate of the sample values. Note the relatively slow gradient from dark to light. This occurs because values far from the center of a sample are weighted the same as those close to the center. The small inset shows the weighting function, which is uniform in this case. On the* **right**, *the sample values are now* weighted *averages, using the weighting function shown in the inset, which emphasizes points closer to the sample point. Weights are chosen to: (a) be positive; (b) be large at the center and small at the boundary; and (c) to sum to one. Notice that now the representation has improved somewhat, as the gradient is sharper and is about in the right place.*

pled image. But if the window is, say, a 5x5 window, small structures that are two grid points away from the query point will have the same effect as small structures that are one grid point away. This can be fixed by weighting the average, so that points near the sample point have a higher weight than points far from the sample point (Figure 2.8). The weighted average is formed as above, but the i, j'th pixel in $\mathcal{N}$ is now the weighted average of a $(2k+1) \times (2k+1)$ window of pixels in $\mathcal{S}$, centered on i, j.

A traditional weighting scheme is given by a one parameter family of functions, derived from the normal distribution and widely called *gaussians*. The parameter σ is sometimes called the *scale* and more usually called the *sigma* of the weights. In a $2k-1 \times 2k-1$ window, the weights will be:

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

where C is chosen so the weights sum to one. Figure 2.10 shows a 5×5 window of these weights, and the considerable improvement in subsampling that can result from using a set of weights. For downsampling by a factor between one and two, $\sigma = 1$ or $\sigma = 1.5$ are fair choices.

Now imagine the downsampling requires a value that *isn't* on the source grid. This value could be interpolated, but it isn't clear what to do about the smoothing. A straightforward trick applies. Take the source image $\mathcal{S}$, and form a new image $\mathcal{N}$ from that source. The i, j'th pixel in $\mathcal{N}$ is now the average of a $(2k+1) \times (2k+1)$ window of pixels in $\mathcal{S}$, centered on i, j. There are some problems when i or j are

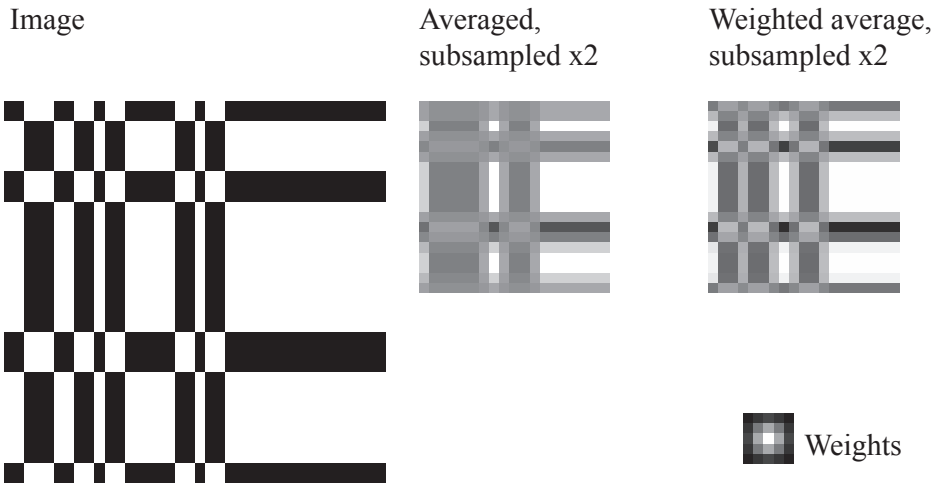

FIGURE 2.10: *The effects shown in Figure 2.9 are quite visible in images. On the* **left**, *an image of stripes ranging from fine to coarse.* **Center**, *a version of the image that has been subsampled by 2, and the value of each sample is an average within a 5x5 window centered on the relevant pixel. Notice how the unweighted average has caused multiple lines to merge into a gray bar, and the relatively "slow" gradient of the lines, which is most obvious on the horizontal lines.* **Right**, *the average in the sample is weighted with the set of weights show on the* **bottom right** *(these weights have been rescaled so the largest weight is light). Notice how some – though not all – of the vertical lines on the left have been resolved, and the faster gradient at the top and bottom of the horizontal lines.*

too big or too small and so the window leaves the source image. Deal with these by *padding* the source image with k rows of zeros at the top and bottom and k columns of zeros on either side. Now downsample the smoothed image $\mathcal{N}$, interpolating as required.

2.3.4 The Gaussian Pyramid

Now consider downsampling by a factor of four. You *could* (but shouldn't) smooth with a gaussian with large σ, then downsample. This is not a good idea, because the support of the gaussian is infinite, meaning that working with a $2k-1 \times 2k-1$ window involves some truncation. As σ gets bigger, k will need to get bigger to keep this truncation reasonable, so the smoothing process will be expensive. The more efficient alternative is to smooth, downsample by two, then smooth the result and downsample *that* by two.

A useful construction follows. In some applications (Section 13.1.2 and Chapter 41.2), it will be useful to have versions of an image downsampled by different factors. A *gaussian pyramid* is a collection of smoothed and downsampled representations of an image. Downsampling is usually by a factor of either two or the square root of two (so two rounds of downsampling halves the edge length of the

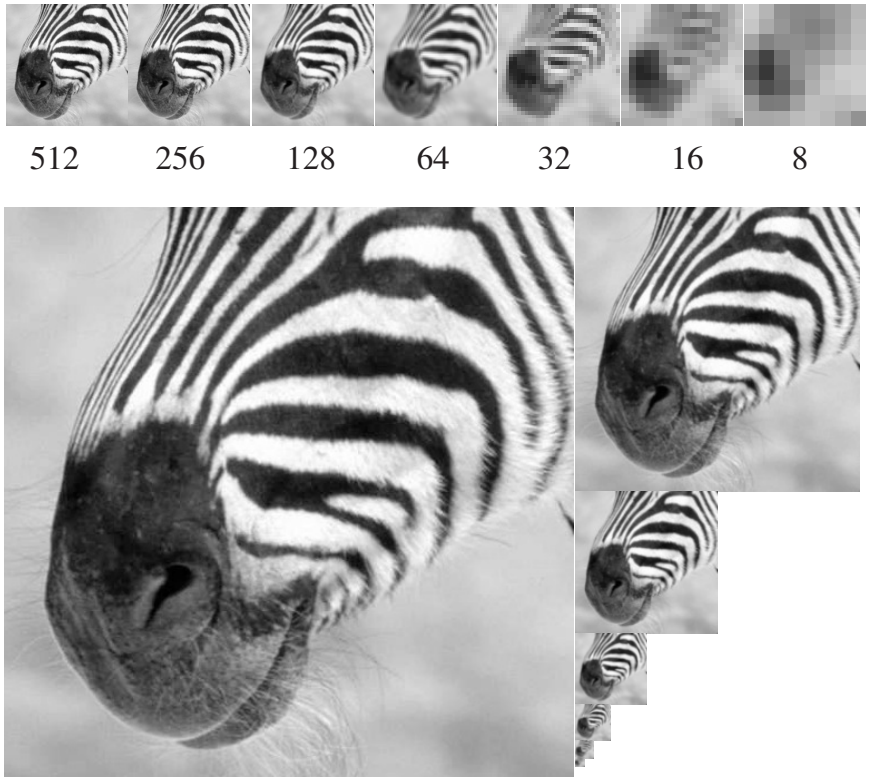

FIGURE 2.11: *A Gaussian pyramid of images running from 512x512 to 8x8. On the top row, I have shown each image at the same size (so that some have bigger pixels than others), and the lower part of the figure shows the images to scale. Notice that an 8x8 pixel block at the finest scale might contain a few hairs; at a coarser scale, it might contain an entire stripe; and at the coarsest scale, it contains the animal's muzzle.*

image). The name comes from a visual analogy. If we were to stack the layers on top of each other, an inverted pyramid would result. The smallest image is the most heavily smoothed. The layers are often referred to as *coarse scale* versions of the image that forms the top layer.

2.3.5 The Laplacian Pyramid

One thing should trouble you about the gaussian pyramid of 2.3.4. There is redundant information in the representation. Write D_σ for the operation that smoothes an image with a gaussian of scale σ then downsamples it; U for the operation that upsamples an image; and G_k for the k'th layer of a gaussian pyramid. This notation suppresses by how much the image is downsampled, and what particular interpolation you use in upsampling, because these aren't important here. An N

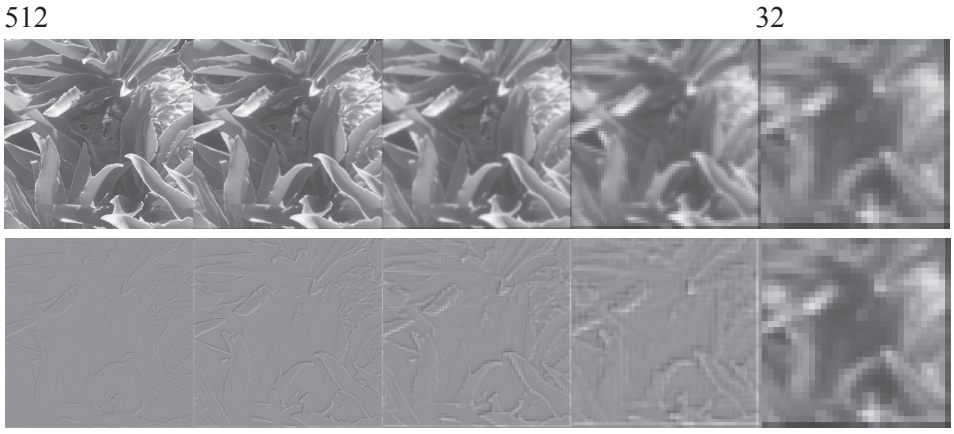

FIGURE 2.12: *A comparison of Gaussian and Laplacian pyramids.* **Top row** *shows a five layer Gaussian pyramid, and* **bottom row** *a Laplacian pyramid derived from it. Each image has been shown at the same size (so the pixels for the* 32×32 *layers are larger). The image is on a scale 0-1 (dark-light). All but the coarsest layer in the Laplacian pyramid have been shown on a scale where mid-gray is 0.5, negative numbers are dark, and positive numbers are light.* Image credit: *Figure shows my photograph of a striped mouse.*

level gaussian pyramid then can be written as:

$$\begin{aligned} G_1 &= \mathcal{I} \\ \dots & \\ G_k &= D_\sigma(G_{k-1}) \\ \dots & \\ G_N &= D_\sigma(G_{N-1}). \end{aligned}$$

Although some information is lost in downsampling and then upsampling, it isn't that much, because $U(G_k)$ looks rather a lot like G_{k-1}. This suggests using a representation where only the residual $G_k - U(G_{k+1})$ is preserved. An N level *laplacian pyramid* can be written as:

$$\begin{aligned} L_1 &= G_1 - U(D_\sigma(G_1)) \\ \dots & \\ L_k &= G_k - U(D_\sigma(G_k)) \\ \dots & \\ L_N &= G_N. \end{aligned}$$

This isn't the most efficient way to build a Laplacian pyramid (**exercises**). Figure 2.12 compares Gaussian and Laplacian pyramids. Each layer of a Laplacian pyramid can be thought of as a representation of image information at a particular

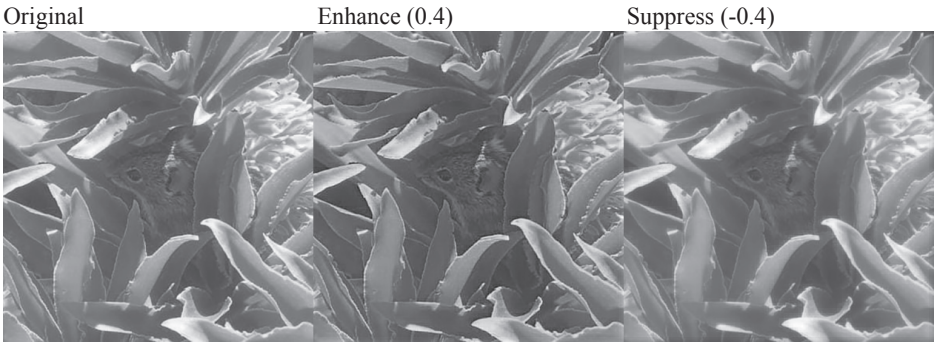

FIGURE 2.13: *Images can be reconstructed from Laplacian pyramids, and weighting components can emphasize or smooth edges. The Laplacian pyramid of Figure 2.12, reconstructed into an image using the method of Section 2.3.5, with* $\alpha = 0$ *(***left***; original image);* $\alpha = 0.4$ *(***center***; emphasizes edges); and* $\alpha = -0.4$ *(***right***; smoothes edges).*

scale. If a pattern in the image is too small for a layer, then it will have been smoothed out; if it is too large, there will be little difference between G_k and $U(D_\sigma(G_k))$ and it will be suppressed by the subtraction.

2.3.6 Reconstruction from Pyramids

It is easy to get an image back from a Gaussian pyramid (take the biggest layer). It is easy to get a gaussian pyramid from a laplacian pyramid, because $G_N = D_\sigma(G_{N-1})$, so $G_{N-1} = L_{N-1} + U(G_N)$, and so on (**exercises** b). In turn, it is easy to recover an image from its laplacian pyramid. Essentially, one just adds the layers. Writing the reconstruction process in an elaborate form exposes some useful tricks in reconstruction. Write

$$\begin{aligned} R_1 &= w(1)L_1 + R_2 \\ \dots & \\ R_k &= w(k)L_k + R_{k+1}\dots \\ R_N &= L_N = G_N. \end{aligned}$$

If all the weights are 1, then $R_1 = \mathcal{I}$ (**exercises**). You can emphasize or deemphasize some effects in the image by upweighting or downweighting the relevant scale by choosing $w(k)$. Using strongly different weights for different scales doesn't usually end well. For the example of Figure 2.13, I used weights obtained by: (a) choosing some largest scale k_x (in this case, $k_x = 3$); (b) choosing a weight α then (c) forming

$$w(k) = \left(1 + \left[\alpha \frac{\max k_x - k0}{k_x}\right]\right).$$

Figure 2.13 shows how various choices of α either sharpen or smooth the image.