Lecture Notes, Week 2: Linear & Non-Linear Filtering

1 Images as Functions

In the past few lectures we studied how images are formed. The ultimate result of this process is light that is incident on the imaging sensor. This light (number of photons) is measured by each of the sensor elements creating a grid of quantized values. We will discuss the quantization process later, but for the time being it is sufficient to assume that a sensor that has a grid of $M \times N$ elements will produce an $M \times N$ array of integer values (or an image) typically represented as `unit8`, *i.e.*, 0 to 255, with 0 corresponding to few and 255 to many photons incident on a given element. A color image will get three channels for red, blue and green portion of the spectra, each of which would be $M \times N$ grid of integer values itself.

As such, the image $I(X, Y)$ can naturally be treated as a function. The *domain* of this function are dimensions of the grid itself, *i.e.*, $(X, Y) \in ([1, M], [1, N])$ where M is the *width* of the image and N is the *height*. The *range* of this function $I(X, Y) \in [0, 255] \in \mathbb{Z}$. Operation on the domain of the function will lead to warping of an image (*e.g.*, image rotation, scaling, distortion). One example of this would be image rectification that would remove lens distortions and ensure that lines in the world appear as lines in the image. We will not talk about this at this stage, but will come back to this later. Changes to the range of this function, on the other hand, allow us to model many important concepts in vision ranging from image enhancement (*e.g.*, sharpening, increasing contrast) to solving specific problems (*e.g.*, finding certain patterns or even objects in an image).

Specifically, there are two classes of operators to consider: *point* operators and *neighborhood* operators. Point operations are modeled by functions that are applied to individual pixels in an image. Specifically, for point operations $I'(X, Y) = g(I(X, Y))$, where $I(X, Y)$ is the input and $I'(X, Y)$ an output image; the $g(x)$ is the operation being applied. For example $g(x) = x - c$ where c is a positive constant will darken an image; $g(x) = x/c$ where $c > 1$ will reduce the contrast, while with $c < 1$ will increase the contrast. Point operators are limited, however, and neighborhood operators are strictly more powerful (*e.g.*, any point operator can be expressed as a neighborhood operator with a neighborhood of 1×1). Neighborhood operators ensure that the output at a particular pixel is a function of a, typically symmetric, $M \times M$ neighborhood around that pixel in the input. In other words, for neighborhood operators

$$I'(X, Y) = g(I(X - k : X + k, Y - k : Y + k)), \text{where } k = (M - 1)/2. \tag{1}$$

Whether we are dealing with point or neighborhood operators, such operators can either be *linear* or *non-linear*. A linear operator is one where the output is a weighted combination of the inputs, while non-linear operator would correspond to more complex interactions between the inputs (*e.g.*, a product of all pixels in the neighborhood would not be linear). We will mostly focus on linear operators which will give rise to linear filtering, but will consider a few non-linear operators as well.

2 Linear Filtering

Many important effects can be modeled by a simple class of neighborhood operators, mainly linear operators or *linear filters*. To apply a linear filter one must first construct a filter or *kernel* to apply – $F(X, Y)$, which is an array, similar to an image. We will assume that $F(X, Y)$ has a size of $M \times M$ (where M is an odd integer) with an indexing coordinate system origin defined at the center of the filter. The values in the filter will correspond to the contribution (or weight) a neighborhood pixel will have towards the output. For example, averaging of the neighborhood pixels can be obtained by letting $F(i, j) = \frac{1}{M^2}$ for all (i, j).

The filter/kernel is applied on an image by superimposing it in each image location and computing the following equation:

$$I'(X, Y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} F(i, j) I(X + i, Y + j) \qquad \textbf{Discrete Correlation}$$

which simply computes a filter weighted sum of the pixels in the neighborhood; $k = (M-1)/2$. Note that the values in the kernel $F(i,j)$ will modulate the output of the linear filtering defined in this way. For example, some kernels will blur an image, others will make it sharper; some will allow us to quantify how localizable the neighborhood may be, or even find whether the neighborhood contains a specific object (more on this later). The equation above characterizes mathematical construct known as *correlation*. One interpretation of correlation is similarity of the filter to the neighborhood it is applied to; this is easy to see if one vectorizes the filter and the patch which would result in equation above implementing a dot product.

Example. The 1D "image" being correlated with a 1D 3×3 kernel will result in the following computation:

$$\begin{aligned} I'(X,Y) &= \frac{1}{4}[1,2,1] \otimes [9,5,2,1,3,4,6,2,4] \\ &= [\ \cdot\ , 21, 10, 7, 11, 17, 18, 14,\ \cdot\ ]. \end{aligned}$$

To see that this operation is linear it can be expressed as a matrix-vector multiplication with kernel replicated as follows:

$$I'(X,Y) = \frac{1}{4} \begin{bmatrix} 1,2,1,0,0,0,0,0,0 \\ 0,1,2,1,0,0,0,0,0 \\ 0,0,1,2,1,0,0,0,0 \\ 0,0,0,1,2,1,0,0,0 \\ 0,0,0,0,1,2,1,0,0 \\ 0,0,0,0,0,1,2,1,0 \\ 0,0,0,0,0,0,1,2,1 \end{bmatrix} \cdot \begin{bmatrix} 9 \\ 5 \\ 2 \\ 1 \\ 3 \\ 4 \\ 6 \\ 2 \\ 4 \end{bmatrix}.$$

Notably, the above equation is a discrete version of correlation defined on the continuous domain:

$$I'(X,Y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} F(i,j) i(X+i, Y+j)\ \mathrm{d}i\ \mathrm{d}j. \qquad \textbf{Continuous Correlation}$$

A closely related mathematical construct is *convolution*, which we will denote by $\otimes$. Correlation and convolution differ only in indexing for the image. Notably:

$$\begin{aligned} I'(X,Y) &= \sum_{i=-k}^{k} \sum_{j=-k}^{k} F(i,j) I(X-i, Y-j) \qquad \textbf{Discrete Convolution} \\ &= \sum_{i=-k}^{k} \sum_{j=-k}^{k} F(-i,-j) I(X+i, Y+j). \end{aligned}$$

Again, this is a discrete version of continuous convolution defined as follows:

$$I'(X,Y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} F(i,j) I(X-i, Y-j)\ \mathrm{d}i\ \mathrm{d}j. \qquad \textbf{Continuous Convolution}$$

For convenience we will actually mostly talk about convolution as opposed to correlation, because convolution has better mathematical properties. However, it is pretty easy to show that: (1) convolving an image with a filter (or kernel) $F(X,Y)$ is equivalent to correlating an image with a filter $F'(X,Y)$ where $F'(X,Y)$ is simply $F(X,Y)$ rotated by 180 degrees; and (2) for a filter, or kernel, that is 180 degree symmetric the convolution and correlation will result in the same output. In practice, it is most convenient to perform all symbolic operations as convolutions, but then apply the resulting filter as correlation (by rotating it) since correlation is more intuitive to apply (*i.e.*, you are likely to make fewer mistakes) in practice.

Lecture Notes, Week 2: Linear & Non-Linear Filtering

2.1 Border Effects

Convolution (and correlation) as defined above, is not defined for placements of the filter close to the border of the image. In particular, superimposing a filter at a pixel that is closer than $\frac{(M-1)}{2}$ to the left/right/top/bottom border will make the filter extend beyond the domain of the image, making the operation undefined. There are a number of ways to handle such cases. Mainly:

- **Ignore Borders.** This will leave the computations at the border "undefined" and result in an output image that is smaller than the input. In particular convolving an image of size $N \times N$ in this way with a filter that is $M \times M$ results in an output that is $(N - M + 1) \times (N - M + 1)$. This is inconvenient.
- **Zero Padding.** Perhaps the most common approach (and one that has mathematical backing) is to pad the image with enough zero columns and rows on each side to ensure that the size of the output image is the same as the input. This requires padding the image with $\frac{(M-1)}{2}$ zero rows / columns on each side. The downside of this is that this may introduce artifacts at the border, *e.g.*, darkening in the output for smoothing filters. Note, this strategy is prevalent in CNNs.
- **Assuming Periodicity.** Assuming periodicity effectively ensures that filter that extends past the image wraps around from left-to-right and top-to-bottom respectively. This alleviates averaging with zero elements which happens with zero padding but may create other artifacts if color/texture of the image between the ends of the image does not match well.
- **Border Reflection.** This strategy copies rows/columns locally by reflecting them over the edge of the image. Similarly to periodicity it alleviates artifacts at the border. Another way to think about it, is that some pixels in the border neighborhood will contribute $2\times$ as much to the weighted sum, because they will be counted both in the original and reflected counterpart.

2.2 Properties of Convolutions

Linear filters and convolutions have a number of important properties that one should consider.

- **Superposition:** Assuming F_1 and F_2 are two filters of the same size (if they are not, then the smaller of the two can be padded with zeros to match size of the larger) then,

$$(F_1 + F_2) \otimes I(X, Y) = F_1 \otimes I(X, Y) + F_2 \otimes I(X, Y). \tag{2}$$

  This can be convenient and efficient when constructing filters that are combinations of other filters (see sharpening in the slides which is a difference between a delta filter and a smoothing filter).

- **Scaling:** Assuming F is a filter and k is a scalar,

$$(kF) \otimes I(X, Y) = F \otimes (kI(X, Y)) = k(F \otimes I(X, Y)). \tag{3}$$

  Again this can be convenient and efficient. For example, implementing box filter by pulling out the normalizing constant reduces computation of convolution from $\mathcal{O}(M^2N^2)$ to $\mathcal{O}(N^2)$.

- **Shift Invariance:** Output of linear filtering is local and does not depend on the position of the filter, only on the values of pixels under it. This also implies sift *equivariance*, *i.e.* moving pixels left/right/up/down will result in similar move of outputs after filtering. Shift variant operation would be the one that takes into account, for example, position of where the filter is applied itself.

Convolution will have additional properties, not shared with correlation. Mainly:

- **Associativity:** Assuming F_1 and F_2 are two filters then

$$F_2 \otimes (F_1 \otimes I(X,Y)) = (F_2 \otimes F_1) \otimes I(X,Y). \tag{4}$$

  This is a very convenient property that allows one to pre-convolve filters before applying them to an image. This property also allows us to apply separable 1D filters in succession.

- **Symmetry:** Similarly, assuming F_1 and F_2 are two filters then

$$(F_1 \otimes F_2) \otimes I(X,Y) = (F_2 \otimes F_1) \otimes I(X,Y). \tag{5}$$

Characterization Theorem: Any linear, shift invariant operation can be expressed as convolution. This shows that a large class of operators can be expressed as convolutions.

3 Efficient Implementation of Convolution

Naive implementation of convolution (or correlation) requires $\mathcal{O}(M^2N^2)$ operations. We saw above that for certain filters the computation can be reduced. Here we consider another class of filters for which this would be the case, as well as the general case that would work for all filters.

Separability: A 2D (or 3D) filter that can be expressed as an outer-product of 1D filters is separable. Similarly, in continuous space, the function of multiple variables is separable if it can be expressed as a product of functions of each of the variables involved. Separable filters, or kernels, can be applied efficiently by filtering the image successively by the corresponding 1D filters. This stems directly from *associativity* property. Doing so, results in drastic reduction of computations from $\mathcal{O}(M^2N^2)$ to $\mathcal{O}(MN^2)$, more specifically to $2MN^2$ computation for an $N \times N$ image being filtered by $M \times M$ filter. Note, not all filters are separable, *e.g.,* box and Gaussian are, while, pillbox is not.

The Convolution Theorem: Convolution theorem states that convolution in FFT space reduces to complex multiplication. In other words, convolution can be applied by first transforming both an image and a filter to FFT space, performing multiplication there, and then applying inverse FFT mapping to obtain the result. This works for *any* filter and computation in FFT space is independent of filter size! Considering that fastest FFT and inverse FFT has complexity $\mathcal{O}(N^2 \log N)$ the overall complexity of this process is approximately $\mathcal{O}(N^2 \log N + M^2 \log M + N^2) = \mathcal{O}(N^2 \log N)$. So as long as $\log N < M^2$ efficiencies will result. Specifically, performing convolution in this way will be efficient for large filters and may actually be slower for smaller ones.

4 Smoothing/Blurring Filters

Smoothing or blurring filters are good for removing unwanted noise from images. They will also prove useful when resizing images, allowing to reduce aliasing artifacts. We will consider three such linear filters. Different filters, in general, would be optimal for different types of noise characteristics. For example, salt and pepper noise would generally be better removed by non-linear median filter; while Gaussian noise by a linear Gaussian filter.

Box Filter. Box filter, which we saw before, of size $M \times M$ has value of $\frac{1}{M^2}$ in each of the array cells. The effect is that box filter uniformly averages pixel values in a squared neighborhood. Larger size of the filter (*i.e.*, larger M) results in more blurring. In terms of the properties, box filter is: (1) not rotationally

symmetric[1], and (2) is separable, allowing efficient implementation. Finally, it can be easily shown that box filter cannot well approximate geometric defocus, or blur, that happens in out-of-focus imaging. The reason for this is that a single point in focus will be blurred in squared shape using a box filter.

Pillbox Filter. Pillbox filter is similar to the box filter in that it also uniformly blurs the image pixels in the neighborhood, but is doing so in circular patter (instead of a square pattern). Pillbox filter can be defined by first defining a continuous function:

$$f_r(x,y) = \frac{1}{\pi r^2} \begin{cases} 1 & \text{if } x^2+y^2 \leq r^2 \\ 0 & \text{otherwise} \end{cases}$$

and then sampling and evaluating it at the center of each cell coordinate within a filter. Specifically a 3×3 pillbox filter with $r = 1$ is obtained as follows:

$$\begin{bmatrix} f_1(-1,-1) & f_1(0,-1) & f_1(1,-1) \\ f_1(-1,0) & f_1(0,0) & f_1(1,0) \\ f_1(-1,1) & f_1(0,1) & f_1(1,1) \end{bmatrix} = \begin{bmatrix} 0 & \frac{1}{\pi} & 0 \\ \frac{1}{\pi} & \frac{1}{\pi} & \frac{1}{\pi} \\ 0 & \frac{1}{\pi} & 0 \end{bmatrix}.$$

Pillbox filter is (1) rotationally symmetric; (2) is not separable, and (3) does model geometric out-of-focus blurring well. Similar to a box filter, larger filter size will result in more blurring. Note that the extent of the function in the continuous space is $2r \times 2r$ this means that for a given filter of size $M \times M$ it should be set to $r = \frac{M}{2}$. Alternatively, for a given value of parameter r it is easy to find a filter size that would capture the extent, or support, of the pillbox filter. Mainly, we can set $M = 2r$ rounded to the next odd integer. For example, for $r = 5$, M should be 11, resulting in pillbox filter that is 11×11. Note that while the continuous function is normalized to sum to 1 over the continuous domain, sampling and evaluating it a discrete set of points corresponding to filter array cell centers will generally produce values that over the filter will not sum to 1 (see example above which will sum to $\frac{5}{\pi}$). To ensure that filter does not change the overall brightness of the image being filtered, it is therefore necessary to normalize the filter to sum to 1.

Gaussian Filter. Similar to the pillbox filter, the Gaussian filter is defined by a continuous function that is sampled and evaluated at the filter cell locations. The function is defined as follows:

$$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2} \exp\left[-\frac{x^2+y^2}{2\sigma^2}\right].$$

Gaussian filter is perhaps the most common filter in vision; we will be using it a lot. The reason for this is that it is as close to the perfect low-pass filter as we have. The parameter σ is standard deviation and controls the spread of the Gaussian function. Important thing to note about the Gaussian is that contributions of the pixels in the neighborhood will depend on the distance from the center pixel. Mainly, there is exponential drop off, meaning that pixels in the neighborhood that are further away from the center pixel, for which filter response is computed, will contribute less. How much less depends on σ.

A larger σ will result in more smoothing and more gradual fall off of contributions of pixels with the distance. A smaller σ will result in less smoothing and sharper drop off of contribution from the neighbors. A very small σ will result in no blurring/smoothing at all. Similar to the pillbox filter above, because we are discretizing a continuous function, we need to choose a filter size that would capture majority of function support (meaning majority of function domain where it has non-zero values). We also need to

[1] A filter is rotationally symmetric implies that amount of smoothing perfromed by the filter will be the same in all directions. A rotationally symmetric filter will not introduce directional edges to the original image.

normalize the filter after sampling, for the same reasons discussed above, *i.e.*, a normalized continuous function sampled at a set of discrete positions will generally not sum up to 1. Finding an appropriate size of the filter for a given value of σ is relatively easy, because we know that majority of Gaussian support (99.7% to be exact) lays between -3σ and $+3\sigma$; hence most of the domain of the Gaussian function on which it is non-zero can be captures by 6σ. The filter size M can hence be obtained by 6σ rounded to the next odd integer. For example, Gaussian with $\sigma = 1.2$ would require a filter of size $1.2 \times 6 = 7.2$ rounded to the next odd integer, so 9×9.

5 Relationship to CNNs

Convolutional Neural Networks (CNNs) is a class of models that has recently been achieving state of the art results on variety of computer vision tasks from object detection and pixel-level segmentation, to generating descriptions of images in natural language form. CNNs are built of convolutional layers – functional operators that take the form of linear filtering. These layers are applied in succession and ultimately result in predictions from the CNN network. Each CNN layer takes the form of linear filter typically applied with zero-padding to the input. The main difference with what we studied in class so far is that (a) the values in these filters are learned (instead of being defined by a function) and (b) that after computing the output of the filtering at each layer a simple non-linearity is applied. Usually the non-linearity takes the form of Rectified Linear Unit (ReLU). ReLU simply zeros out elements of the output array that result in value below 0. Note that while we have not seen this so far, filter values that are < 0 can result in output value that would be below 0. We will study why non-linearities are necessary in CNNs as well as how we can potentially learn parameters of the CNNs, comprising in part of the filters.

6 Non-linear Filters

The ReLU operation discussed above makes CNN layer non-linear (even though the core operation of filtering within it linear). Other non-linear filtering operators and functions exists. We will mainly study two of them in this class: Median and Bilateral.

Median. Median is perhaps the simplest non-linear denoising filter. It receives as input a neighborhood of $M \times M$ pixel values centered at a pixel. However, rather than taking a weighted combination of these values, it simply computes a median statistic on them. Specifically it sorts the values from the neighborhood from lowest to highest and takes one from the middle of the sorted array as an output. Note that while median filter is shift invariant it cannot be implemented as convolution. Median filter is particularly useful for removing salt and paper or speckle noise.

Bilateral Filter. Bilateral is an edge-preserving non-linear smoothing filter. Similar to a Gaussian filter, the filter weights for the Bilateral filter depend on spatial distance from the center pixel. Hence pixels closer to the center pixel have higher contribution to the output as compared to pixels further away. However, unlike the Gaussian filter, Bilateral filter weights also depend on the greyscale/color difference between center pixel and neighborhood pixels. In other words, pixels that are equally far away with similar value to the center pixel will contribute more than pixels with dissimilar value. This can be expressed as a product of two Gaussian kernels (a *domain* and a *range* kernel). Therefore a Bilateral filter can be written as follows:

$$B_{\sigma_d,\sigma_r}(x, y, I(X, Y)) = \frac{1}{2\pi\sigma^2} \exp\left[-\frac{x^2 + y^2}{2\sigma_d^2}\right] \cdot \frac{1}{2\pi\sigma^2} \exp\left[-\frac{(I(X + x, Y + y) - I(X, Y))^2}{2\sigma_r^2}\right].$$

Worth noting that the kernel now is a function of the image itself and will change for each superposition.