CS 280 Computer Vision
Spring 2018 Efros, Malik, Yu Lecture 7

1 Lecture

Image Processing

By looking at histograms, we can see if an image uses all of its pixel values. We can perform histogram equalization by taking the current cumulative histogram and using it as a mapping from input to output. This will perform a stretching to use the whole $[0, 255]$ range.

Histogram matching: instead of making the bins equal, we can make the histogram have the same shape as some other image's histogram. We can do this by "undoing" the first cumulative histogram, then mapping it to the second cumulative histogram.

- Histogram matching can be applied to color transfer.

Note: point processing methods like histogram equalization know nothing about the spatial layout of the image. They treat every pixel independently and don't care about the spatial position of each pixel. In a way, the beauty of the image is lost! Therefore, we progress to *spatial* processing.

But first, some signal processing.

Signal Processing

Everything is continuous *until* we get to the CCD array in the back of our camera. The world is continuous (for all intents and purposes), but we have a discrete number of pixels! So at some point, we have to go from continuous to discrete. Then, sometimes, we might want to consider our discrete signal as a continuous signal. This is when we need to know something about sampling and reconstruction.

Sampling

Sampling is when we have a finite, discrete number of pixels and an infinite precision signal that gets sampled at even intervals using our discrete space. Even after we do this, our values are infinite precision (floats). So we have to **quantize** the values in order to make them representable as bytes (i.e. integers in the range $[0, 255]$). We then end up with values that are regular in both x and y.

Sadly, we're losing information. We're representing an infinite precision signal as a set of discrete samples! How can we perform reconstruction after losing information? This is where all the complications arise.

Reconstruction

If we undersample, we will probably reconstruct things incorrectly due to having missed things. Our signal might be indistinguishable from a lower (or higher) frequency! This problem is called *aliasing*, where signals "travel in disguise" as other frequencies (classic example: wheels appearing to rotate backward).

In order to defeat aliasing, we can *sample more often* (which can't go on forever) or we can simply *get rid of high frequencies* ("make the signal less wiggly"). In other words, we can smooth our signals with a lowpass filter, leaving only safe, low frequencies. This will occur *before* the sampling happens. Then, during reconstruction, we can disambiguate by choosing the lowest frequency. If we sample at twice the highest frequency of the signal, we can then perform lossless reconstruction.

Linear Filters

A lowpass filter is an example of a whole family of *linear filters*: transformations of signals that operate on neighborhoods and have nice linear properties. For a linear filter, we have $\text{filter}(f + g) = \text{filter}(f) + \text{filter}(g)$ (the linearity property). And then the main property is that they are *shift-invariant*: what we do in one part of the signal is the same as what we do in another part of the signal. We are doing the same thing to all of the neighborhoods, meaning we can parallelize all of our computation!

Smoothing, blurring, etc. are all examples of linear filters. These can be modeled mathematically by *convolution*. Basic idea: to smooth (or do whatever to) a signal, we can average it over a sliding window. We can then add weights to our moving average, performing a dot product of our signal with a long vector which uses 0s for non-included entries. Essentially, we're doing a series of dot product as we move along the signal.

In 2D, we do the same thing. We shift a mask around a grid of values and perform a lot of dot products. In an image signal, smooth changes are low frequencies and sharp changes are high frequencies.

If F is the image, H is the kernel (of size $2k + 1 \times 2k + 1$), and G is the output image, then we define the **cross-correlation** $G = H \otimes F$ as

$$G[i, j] = \sum_{u=-k}^{k} \sum_{v=-k}^{k} H[u, v]F[i + u, j + v]$$

This translates to taking our kernel and our image, and computing a dot product for every shift of the kernel H around the image F. It's a shifted average / dot product.

What is the difference between cross-correlation and convolution? Convolution is just cross-correlation where the filter is flipped both horizontally and vertically before being applied to the image:

$$G[i, j] = \sum_{u=-k}^{k} \sum_{v=-k}^{k} H[u, v]F[i - u, j - v]$$

It is written as $G = H * F$. We do convolution instead of cross-correlation because it has nicer properties: it is both commutative and associative.

Using cross-correlation, we see that $\begin{bmatrix}1 & 2 & 3\end{bmatrix} \otimes \begin{bmatrix}0 & 1 & 0\end{bmatrix}$ is not equivalent to $\begin{bmatrix}0 & 1 & 0\end{bmatrix} \otimes \begin{bmatrix}1 & 2 & 3\end{bmatrix}$.

In fact convolution $b = c * a$ has many nice properties. It's like multiplication, in that

- it exhibits commutativity: $a * b = b * a$
- it exhibits associativity: $a * (b * c) = (a * b) * c$
- it distributes over addition: $a * (b + c) = a * b + a * c$
- scalars factor out: $\alpha a * b = a * \alpha b = \alpha(a * b)$
- it has an identity, the unit impulse: $e = \begin{bmatrix}..., 0, 0, 1, 0, 0, ...\end{bmatrix}$ for which $a * e = a$

Note: for a lot of filters, i.e. symmetric ones, there is no difference between convolution and cross-correlation. Also, conceptually there is no distinction between the filter and the signal.

Associativity often comes in handy! Applying many filters one after another is equivalent to applying one composed filter:

$$(((a * b_1) * b_2) * b_3) = a * (b_1 * b_2 * b_3)$$

The latter is much faster, since filters are smaller than images.

Gaussians play very well with convolutions. They are lowpass filters, and therefore remove high-frequency components from images. Conveniently, convolving a Gaussian with a Gaussian produces another Gaussian, just a fatter one.

The Convolution Theorem

Convolutions connect the spatial domain with the Fourier domain. In 2D, Fourier transforms allow us to look at a signal in terms of its frequency content (what frequencies are in the signal, kind of like a histogram of frequencies). A Fourier transform is similar to a change of basis: instead of looking at pixels, we're counting frequencies.

As it happens, the Fourier transform of the convolution of two functions is the same as the product of their Fourier transforms:

$$F[g * h] = F[g]F[h]$$

And the inverse Fourier transform of the product of two Fourier transforms is the convolution of the two inverse Fourier transforms:

$$F^{-1}[gh] = F^{-1}[g] * F^{-1}[h]$$

Convolution in the spatial domain is equivalent to multiplication in the frequency (Fourier) domain!

Accordingly, if we take the Fourier transform of our image and multiply it with the Fourier transform of our filter, then take the inverse Fourier transform, we get the properly filtered image! This is very cool.

An aside: in Fourier domain visualizations, the low frequencies are in the center, and as we go away from the center, we basically count the higher and higher frequencies.