Linear Filters and Convolutions

Lecturer: Pedro Felzenszwalb
Scribe: Eric Purdy

April 2, 2008

1 Local Averages

Consider a noisy image, in which each value are perturbed a little bit. At each pixel, we add a small random value to the image intensity. See Figure 1.

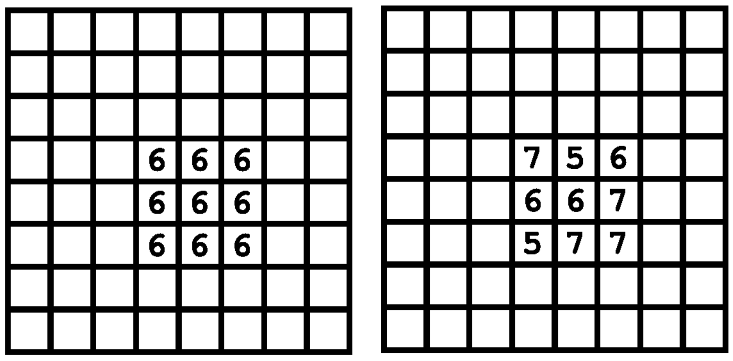

Figure 1: An image, left, and a noisy version of the same image, right.

We want to get rid of this noise somehow. How can we do that? One way is to consider local averages. Most pixels in an image tend to have roughly the same color as their neighbors, since images tend to be composed of somewhat homogenously colored objects. So, local averaging could be a good idea.

This leads to the following idea: for every pixel p, compute the average value in a $k \times k$ window around p, and use this for our predicted value at p. Actually, we want this window to be symmetric around p, so we will use a $2k+1 \times 2k+1$ window. See Figure 2.

We can code this up as follows:

```
procedure local-average(image I)
    for $y = k$ to height $-k$ do
        for $x = k$ to width $-k$ do
            sum := 0
            for $v = -k$ to $k$ do
```

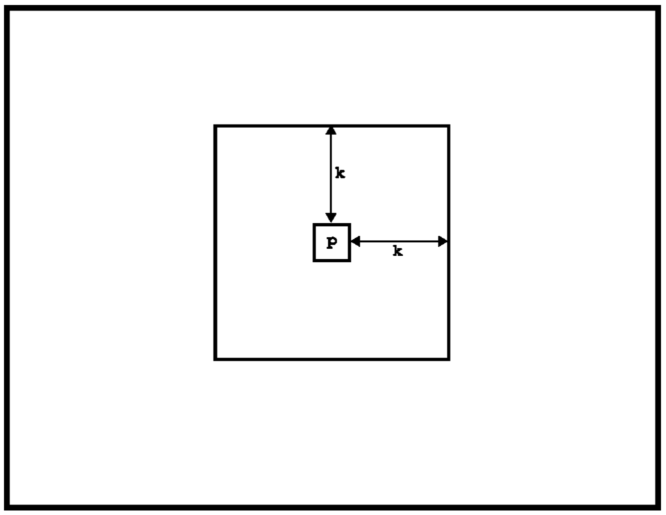

Figure 2: A $k \times k$ window.

```
            **for** $u = -k$ to $k$ **do**
                sum := sum + $I[x+u, y+v]$
            **end**(**for**)
        **end**(**for**)
        $O[x,y] := sum/(2k+1)(2k+1)$
    **end**(**for**)
**end**(**for**)
**return** O
```

To simplify matters, we have left off what happens near the borders - we would have to change things a bit so that $(x+u, y+v)$ is still inside the image.

As an aside, when iterating over images, you should always do it the way we did it in the previous pseudocode, where we iterated over y first and then over x. Doing it this way makes our code run much faster, because we benefit from cache coherence. Images (and 2-D arrays in general) are usually stored in row major order, which means that we implement a $w \times h$ 2-D array by creating a normal array of length wh, which has the first row of our 2-D array, and then the second, and so forth. Iterating over x first causes our algorithm to jump all over the image, while iterating over y first causes us to move mostly in a straight line. This layout is shown in Figure 3.

How fast is the code we just wrote? It runs in time $O(\text{width}\times\text{height}\times k^2)$. This can get a little slow if we want to use a large k. We would like to have an algorithm that runs in time independent of k. The real work is in computing the sums (the number of divisions does not depend on k), so really we just want to be able to take the sum of the image intensities over a $k \times k$ box in time independent of k. It turns out we can do this.

Consider the corresponding 1-D problem. We have an array F, and we want

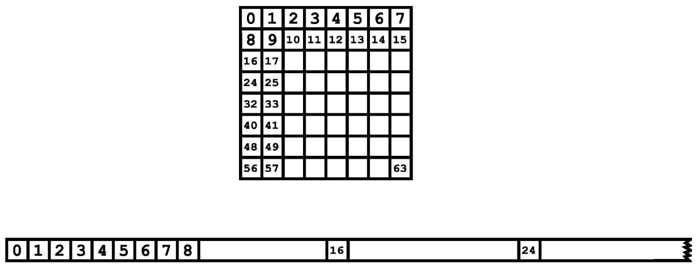

Figure 3: Row-major order.

to compute the sums of all the intervals of length k. We can do this easily using the sliding window approach: we calculate the sum of the first k,

$$O[k-1] = F[0] + \cdots + F[k-1]$$

and then we calculate each interval sum by adding one more value to the end and subtracting the first one off the beginning:

$$O[x] = O[x-1] + F[x] - F[x-k].$$

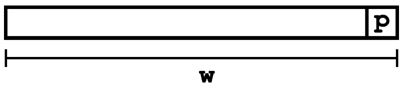

Figure 4: The value at p is the sum of the run of w pixels ending at p.

Can we use this insight in the 2-D case? Yes, because the filter we want to use is *separable* - it factors into two one-dimensional filters. Suppose our box size is $w \times h$. We run our one-dimensional algorithm in each row of the image, summing over all intervals of length w.

This gives us an intermediate image J, in which the value at a pixel p is the sum over the w pixels in p's row that start $w-1$ pixels to the left of p and end at p. Now, if we compute the interval sums over the columns of J, we will get the desired result, the sum over all $w \times h$ boxes. So, we can compute box sums in time independent of box size.

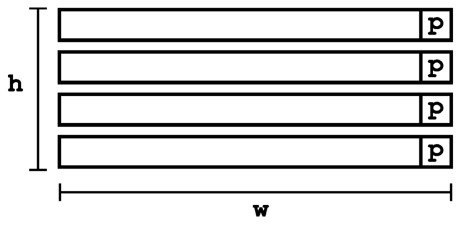

Figure 5: The sum over a $w \times h$ box.

2 The Integral Image

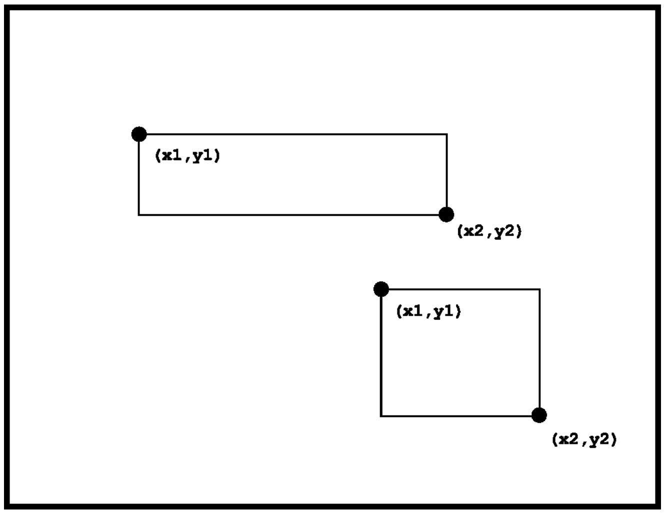

Figure 6: Some possible queries.

What if we wanted different size boxes, as in Figure 6? There is an even more clever approach: the integral image. We want to pre-process an image so that we can quickly compute sums over arbitrarily sized rectangles.

Again, it is helpful to consider the one-dimensional case first. How would we preprocess an array so that we can quickly find the sum of the array over an arbitrary interval? We do this using a cumulative sum:

$$S[x] = \sum_{i=0}^{x} F[i].$$

We can compute this very quickly, by setting

$$S[0] = F[0]$$

and

$$S[x] = S[x-1] + F[x].$$

We can compute this in linear time. Then the sum of F from i to j is just $S[j] - S[i-1]$. Can we do this in two dimensions?

We take as input an image I. We will compute an image S, called the *integral image*:

$$S[x, y] = \sum_{x' \leq x, y' \leq y} I[x', y'].$$

We can compute this quickly, by doing each row in one pass and each column in a second pass. We will have an intermediate image $C[x, y]$, and we will compute:

$$C[0, y] = I[0, y]$$

$$C[x,y] = C[x-1,y] + I[x,y]$$
$$S[x,0] = C[x,0]$$
$$S[x,y] = S[x,y-1] + C[x,y].$$

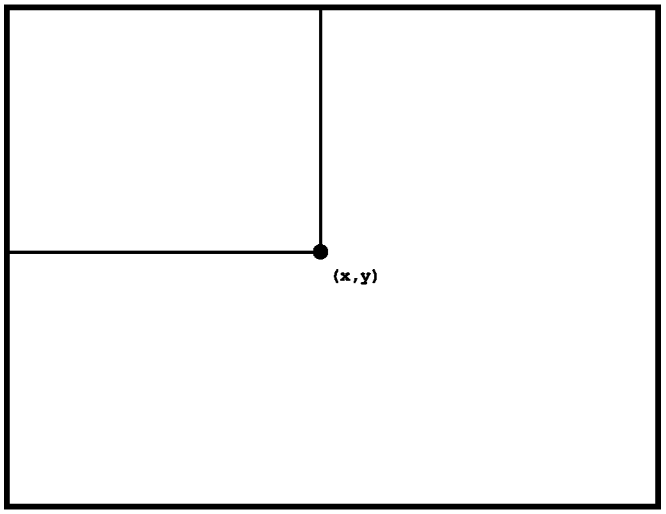

Figure 7: The integral image at (x,y) has the sum over the box with corners $(0,0)$ and (x,y).

S now tells us the sum over any rectangle whose upper left corner is the origin. How can we compute the sum over a different rectangle? We use inclusion-exclusion. Suppose that we are given our rectangle as two points $(x_1,y_1),(x_2,y_2)$, where (x_1,y_1) is the upper left corner of the rectangle, and (x_2,y_2) is the lower right corner of the rectangle. Then the sum over the rectangle is

$$R(x_1,y_1,x_2,y_2) = S[x_2,y_2] - S[x_1-1,y_2] - S[x_2,y_1-1] + S[x_1-1,y_1-1].$$

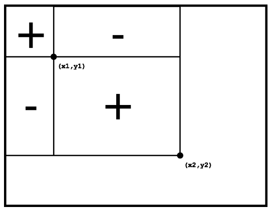

Figure 8: Using Inclusion-Exclusion.

We will want to use this technique later on, but for now it lets us efficiently compute local averages, which is nice.

3 Convolutions

We want to generalize the notion of local averages - this will give us convolutions, which can be thought of as weighted local averages.

Local averaging blurs an image, but not necessarily in the best way. Consider the following example: we have an region of uniform color which should show up in an image as intensity 6. There is some noise, so the values will be randomly perturbed. Suppose that one pixel is perturbed a lot - instead of 6, it hs intensity 72.

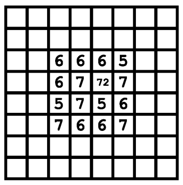

Figure 9: One pixel has a lot of noise.

(Maybe one of the little electronic components of your digital camera is burning out.) Local averages don't deal particularly well with this - in the blurred image, we will have a sudden jump when the box that we are moving around starts including the value 72.

To defeat this, we want a weighted average, where things at the border are counted less than things near the center. Then the pixel with intensity 72 will not have much effect on values that are close to it but not that close. We can think of a weighted average as being given by a mask. We can define an operation called *cross-correlation* this way:

$$(I * H)[x, y] = \sum_u \sum_v I[x + u, y + v]H[u, v]$$

Another operation with almost the same definition is *convolution*, which is sort of the cross-correlation with a flipped version of H:

$$(I \otimes H)[x, y] = \sum_u \sum_v I[x - u, y - v]H[u, v]$$

These are basically the same operation, but convolution has some nice algebraic properties that cross-correlation doesn't. (Also, note that if the filter H is symmetric, with $H[-u, -v] = H[u, v]$, then the two operations are the same.)

Convolution is commutative: $A \otimes B = B \otimes A$. Convolution is associative: $(A \otimes B) \otimes C = A \otimes (B \otimes C)$. Convolution satisfies the distributive law: $A \otimes (B + C) = A \otimes B + A \otimes C$. (Here the addition is pointwise - we add the intensities of B and C at every point, so $(B + C)[x, y] = B[x, y] + C[x, y]$.) So, if we want to convolve an image A with a filter B, and then convolve the result with a filter C, we might be able to save time by precomputing the convolution of B and C. These are the basic tools of image processing. They have two key properties - the

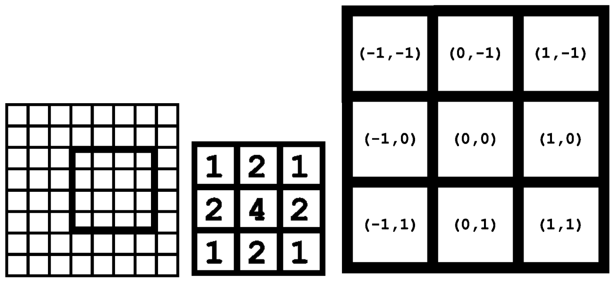

Figure 10: Left: a window in the image. Middle: a filter. Right: the coordinates of the filter

distributive law, which is a sort of linearity property, and translation invariance. If we shift an image I by a vector (u, v), so that $I'[x, y] = I[x - u, y - v]$, and then convolve that with a filter, we get the same result as if we convolve with the filter and then translate.

Why does convolution have these nice algebraic properties? One way to understand it it to think of arrays as polynomials. So, our array F turns into the polynomial $f(x) = F[0] + F[1]x + F[2]x^2 + \cdots + F[n]x^n$. Then you can check that the array $F \otimes G$ is exactly the same as the array corresponding to the polynomial we get by multiplying $f(x)$ and $g(x)$. Similarly, for a 2-D array, we can associate it with a polynomial in two variables, where the coefficient of $x^i y^j$ is $I[i, j]$. Again, the result of convolving two such arrays F and G corresponds to the result of multiplying the corresponding polynomials $f(x, y)$ and $g(x, y)$.

When we translate our algebraic properties into this setting, we see that we are just saying that multiplication of polynomials is commutative and associative, and distributes over addition, which are all familiar facts.

One of the most important filters is the Gaussian filter, $H[u, v] = \frac{1}{2\pi\sigma^2} e^{-\frac{u^2+v^2}{\sigma^2}}$. In this class, we will always smooth noise by convolving with a Gaussian.

There is one strange thing about the Gaussian though - the way we have defined it, it is infinitely large! We need to cut if off somewhere, but if we cut it off too soon, we will suffer from the same problem that the box filter suffered from. The standard method for dealing with this is to cut off the Gaussian at four standard deviations, so that it goes from -4σ to 4σ. This cuts off only a very tiny fraction of the weight.