Lecture 13: Edge Detection

©Bryan S. Morse, Brigham Young University, 1998–2000
Last modified on February 12, 2000 at 10:00 AM

Contents

Reading

SH&B, 4.3.2–4.3.3
Castleman 18.4–18.5.1

13.1 Introduction

Remember back in Lecture 2 that we defined an *edge* as a place of local transition from one object to another. They aren't complete borders, just locally-identifiable *probable* transitions.

In this lecture and the next, we'll discuss ways for detecting edges locally. To do this, we will exploit local neighborhood operations as covered in CS 450. If you need to brush up on the basics of convolution, now would be a good time to do so.

13.2 First-Derivative Methods

Most edge detectors are based in some way on measuring the intensity gradient at a point in the image.

Recall from our discussion of vector calculus and differential geometry that the gradient operator ∇ is

$$\nabla = \begin{bmatrix} \frac{\partial}{\partial x} \\ \\ \frac{\partial}{\partial y} \end{bmatrix} \tag{13.1}$$

When we apply this vector operator to a function, we get

$$\nabla f = \begin{bmatrix} \frac{\partial}{\partial x} f \\ \\ \frac{\partial}{\partial y} f \end{bmatrix} \tag{13.2}$$

As with any vector, we can compute its magnitude $\|\nabla f\|$ and orientation $\phi\left(\nabla f\right)$.

- The gradient magnitude gives the *amount* of the difference between pixels in the neighborhood (the *strength* of the edge).
- The gradient orientation gives the *direction* of the greatest change, which presumably is the direction across the edge (the edge normal).

Many algorithms use only the gradient magnitude, but keep in mind that the gradient orientation often carries just as much information.

Most edge-detecting operators can be thought of as gradient-calculators. Because the gradient is a continuous-function concept and we have discrete functions (images), we have to approximate it. Since derivatives are linear and shift-invariant, gradient calculation is most often done using convolution. Numerous kernels have been proposed for finding edges, and we'll cover some of those here.

13.2.1 Roberts Kernels

Since we're looking for differences between adjacent pixels, one way to find edges is to explictly use a $\{+1, -1\}$ operator that calculates $I(\overline{x}_i) - I(\overline{x}_j)$ for two pixels i and j in a neighborhood. Mathematically, these are called *forward differences*:

$$\frac{\partial I}{\partial x} \approx I(x+1, y) - I(x, y)$$

The Roberts kernels attempt to implement this using the following kernels:

+1	
	-1

g_1

	+1
-1	

g_2

While these aren't specifically derivatives with respect to x and y, they *are* derivatives with respect to the two diagonal directions. These can be thought of as components of the gradient in such a coordinate system. So, we can calculate the gradient magnitude by calculating the length of the gradient vector:

$$g = \sqrt{(g_1 * f)^2 + (g_2 * f)^2}$$

The Roberts kernels are in practice too small to reliably find edges in the presence of noise.

13.2.2 Kirsch Compass Kernels

Another approach is to apply rotated versions of these around a neighborhood and "search" for edges of different orientations:

3	3	3
3	0	3
-5	-5	-5

3	3	3
-5	0	3
-5	-5	3

-5	3	3
-5	0	3
-5	3	3

$\ldots$

Notice that these do not explicitly calculate the gradient—instead, they calculate first-derivatives in specific directions. By taking the result that produces the maximum first derivative, we can approximate the gradient magnitude. However, the orientation is limited to these specific directions. These kernels make up the *Kirsch compass kernels* (so called because they cover a number of explicit directions).

13.2.3 Prewitt Kernels

The Prewitt kernels (named after Judy Prewitt) are based on the idea of the *central difference*:

$$\frac{df}{dx} \approx [f(x+1) - f(x-1)]/2$$

or for two-dimensional images:

$$\frac{\partial I}{\partial x} \approx [I(x+1, y) - I(x-1, y)]/2$$

This corresponds to the following convolution kernel:

-1	0	1

(To get the precise gradient magnitude, we must then divide by two.) By rotating this 90 degrees, we get $\partial I/\partial y$.

These kernels are, however, sensitive to noise. Remember from CS 450 (or from another signal- or image-processing course) that we can reduce some of the effects of noise by *averaging*. This is done in the Prewitt kernels by averaging in y when calculating $\partial I/\partial x$ and by averaging in x when calculating $\partial I/\partial y$. These produce the following kernels:

-1	0	1
-1	0	1
-1	0	1

$\partial/\partial x$

-1	-1	-1
0	0	0
1	1	1

$\partial/\partial y$

(Of course, now we have to divide by six instead of two.) Together, these give us the components of the gradient vector.

13.2.4 Sobel Kernels

The Sobel kernels (named after Irwin Sobel, now currently working for HP Labs) also rely on central differences, but give greater weight to the central pixels when averaging:

-1	0	1
-2	0	2
-1	0	1

$\partial/\partial x$

-1	-2	-1
0	0	0
1	2	1

$\partial/\partial y$

(Of course, now we have to divide by eight instead of six.) Together, these also give us the components of the gradient vector.

The Sobel kernels can also be thought of as 3×3 approximations to first-derivative-of-Gaussian kernels. That is, it is equivalent to first blurring the image using a 3×3 approximation to the Gaussian and then calculating first derivatives. This is because convolution (and derivatives) are commutative and associative:

$$\frac{\partial}{\partial x}(I * G) = I * \frac{\partial}{\partial x} G$$

This is a very useful principle to remember.

13.2.5 Edge Extraction

Most edges are not, however, sharp dropoffs. They're often gradual transitions from one intensity to another. What usually happens in this case is that you get a rising gradient magnitude, a peak of the gradient magnitude, and then a falling gradient magnitude. Extracting the ideal edge is thus a matter of finding this curve with optimal gradient magnitude. Later, we'll discuss ways of finding these optimal curves of gradient magnitude.

13.3 Second-Derivative Methods

There are other operators that try to find these peaks in gradient magnitude directly.

Let's first consider the one-dimensional case. Finding the ideal edge is equivalent to finding the point where the derivative is a maximum (for a rising edge with positive slope) or a minimum (for a falling edge with negative slope).

How do we find maxima or minima of one-dimensional functions? We differentiate them and find places where the derivative is zero. Differentiating the first derivative (gradient magnitude) gives us the second derivative. Finding optimal edges (maxima of gradient magnitude) is thus equivalent to finding *places where the second derivative is zero*.

When we apply differential operators to images, the zeroes rarely fall exactly on a pixel. Typically, they fall between pixels. We can isolate these zeroes by finding *zero crossings*: places where one pixel is positive and a neighbor is negative (or vice versa).

One nice property of zero crossings is that they provide closed paths (except, of course, where the path extends outside the image border). There are, however, two typical problems with zero-crossing methods:

1. They produce two-pixel thick edges (the positive pixel on one side and the negative one on the other, and
2. They can be extremely sensitive to noise.

For two dimensions, there is a single measure, similar to the gradient magnitude that measures second derivatives. Consider the dot product of ∇ with itself:

$$\nabla \cdot \nabla = \begin{bmatrix} \frac{\partial}{\partial x} \\ \\ \frac{\partial}{\partial y} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial}{\partial x} \\ \\ \frac{\partial}{\partial y} \end{bmatrix} \tag{13.3}$$

$$\tag{13.4}$$

$$= \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \tag{13.5}$$

We usually write $\nabla \cdot \nabla$ as ∇^2. It has special name and is called the *Laplacian* operator.

When we apply it to a function, we get

$$\nabla^2 f = \left(\begin{bmatrix} \frac{\partial}{\partial x} \\ \\ \frac{\partial}{\partial y} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial}{\partial x} \\ \\ \frac{\partial}{\partial y} \end{bmatrix} \right) f \tag{13.6}$$

$$= \frac{\partial^2}{\partial x^2} f + \frac{\partial^2}{\partial y^2} f \tag{13.7}$$

One interesting property of the Laplacian is that it is rotationally invariant—it doesn't depend on what directions you choose, so long as they are orthogonal. The sum of the second derivatives in *any* two orthogonal directions is the same.

We can find edges by looking for zero-crossings in the Laplacian of the image. This is called the *Marr-Hildreth* operator.

Let's now turn our attention then to neighborhood operators for measuring the Laplacian.

13.3.1 Laplacian Operators

One way to measure the second drivative in one dimension is to extend the forward-differencing first-derivative operator to the forward difference of two forward differences:

$$\begin{array}{rcl} \frac{d^2 f}{dx^2} & \approx & [f(x+1) - f(x)] - [f(x) - f(x-1)] \\ & = & f(x+1) - 2f(x) + f(x-1) \end{array} \tag{13.8}$$

We can thus use a simple $[1, -2, 1]$ neighborhood operator to approximate the second derivative in one dimension. The Laplacian is one of these in the x direction added to one of these in the y direction:

0	0	0
1	-2	1
0	0	0

+

0	1	0
0	-2	0
0	1	0

=

0	1	0
1	-4	1
0	1	0

Other variations include the following:

$$\begin{array}{|c|c|c|}\hline 0.5 & 0.0 & 0.5 \\ \hline 1.0 & -4.0 & 1.0 \\ \hline 0.5 & 0.0 & 0.5 \\ \hline \end{array} + \begin{array}{|c|c|c|}\hline 0.5 & 1.0 & 0.5 \\ \hline 0.0 & -4.0 & 0.0 \\ \hline 0.5 & 1.0 & 0.5 \\ \hline \end{array} = \begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & -8 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$$

or

$$\begin{array}{|c|c|c|}\hline 1 & -2 & 1 \\ \hline 1 & -2 & 1 \\ \hline 1 & -2 & 1 \\ \hline \end{array} + \begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline -2 & -2 & -2 \\ \hline 1 & 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|c|}\hline 2 & -1 & 2 \\ \hline -1 & -4 & -1 \\ \hline 2 & -1 & 2 \\ \hline \end{array}$$

As mentioned earlier, zero crossings of this operator can be used to find edges, but it is susceptible to noise. This sensitivity comes not just from the sensitivity of the zero-crossings, but also of the differentiation. In general, the higher the derivative, the more sensitive the operator.

13.3.2 Laplacian of Gaussian Operators

One way to deal with the sensitivity of the Laplacian operator is to do what you'd do with any noise: blur it. This is the same philosophy we saw earlier with the Sobel kernels for first derivatives.

If we first blur an image with a Gaussian smoothing operator and then apply the Laplacian operator, we can get a more robust edge-finder.

But why go to all the trouble? Convolution is associative and commutative, so we can combine the Gaussian smoothing operator with the Laplacian operator (by convolving them one with another) to form a single edge-finding operator.

But this is still too much work. Instead of approximating the Laplacian operator with forward differencing and then applying it to a Gaussian, we can simply differentiate the Gaussian

$$G(x, y) = e^{-r^2/2\sigma^2} \tag{13.9}$$

(where $r^2 = x^2 + y^2$) to get a closed-form solution for the Laplacian of Gaussian:

$$\nabla^2 G(x, y) = \left(\frac{r^2 - \sigma^2}{\sigma^4}\right) e^{-r^2/2\sigma^2} \tag{13.10}$$

We can then approximate this over a discrete spatial neighborhood to give us a convolution kernel.

But wait! There's an even better way. We can not only calculate the closed-form solution for a Laplacian of Gaussian, we can compute its Fourier Transform. We know that the Fourier Transform of a Gaussian is another Gaussian, and we also know that we can differentiate using a ramp function ($2\pi i\nu_x$ or $2\pi i\nu_y$) in the frequency domain. Multiply together the spectrum of the image, the Fourier Transform of a Gaussian, and two differentiating ramps in the one direction and you have a second-derivative of Gaussian in one direction. Do the same thing in the other direction, add them together, and you have the Laplacian of Gaussian of the image. (See, that Fourier stuff from CS 450 comes in handy after all.)

13.3.3 Difference of Gaussians Operator

It can be shown that the Laplacian of a Gaussian is the derivative with respect to $2\sigma^2$ of a Gaussian. That is, it is the limit of one Gaussian minus a just smaller Gaussian. For this reason, we can approximate it as the difference of two Gaussians. This is called the *Difference-of-Gaussians* or DoG operator.

Also appearing in the literature is the *Difference-of-LowPass filters* or DoLP operator.

13.4 Laplacians and Gradients

Zero-crossings of Laplacian operators only tell you where the edge is (actually where the edge lies *between*), not how strong the edge is. To get the strength of the edge, and to pick which of the two candidate pixels is the better approximation, we can look at the gradient magnitude at the position of the Laplacian zero crossings.

13.5 Combined Detection

If one operator works well, why not use several? This seems like a reasonable thing to do, but the way to do it is nontrivial. We won't go into it in class, but understand that it is trying to combine information from multiple orthogonal operators in a vector space and then project the results to the ”edge subspace”.

13.6 The Effect of Window Size

The results you get with a 3×3, 5×5, or 7×7 operator will often be different. What is the correct window size? Well, there isn't one standard one that will work under every circumstance. Sometimes you have to play with it for your application. Smart vision programs will often use multiple windows and combine the results in some way. We'll talk about this notion more in the next lecture.

Vocabulary

- Gradient
- Gradient magnitude
- Gradient orientation
- Roberts edge detector
- Prewitt edge detector
- Sobel edge detector
- Kirsch compass gradient kernels
- Laplacian
- Laplacian-of-Gaussian
- Difference-of-Gaussians
- Marr-Hilbert edge detector