Final Exam Preparation

Andrés Ponce

December 21, 2020

Chapter 10: Image Segmentation

Image segmentation involves finding the area of interest in an image, by identifying the different objects in it and trying to identify the objects inside it.

There are two basic principles to apply, **discontinuity** and **similarity**. The first refers to local discontinuity in the intensity of the images, while the second implies that pixels of the same region will not vary too much from each other. We can do detection where we only detect the edges between the different edges in an object or where we fill in the pixels of each region.

Point, Line, and Edge Detection

These types of features are areas of the image where there is an abrupt change in the local values of the intensities in the pixels. These areas are called *edges*. Since we detect the change in intensity from one pixel to another, this is equivalent to calculating the **derivative** of the pixels.

At a value x, its derivative is

$$\frac{\partial f}{\partial x} = f'(x) = f(x+1) - f(x)$$

while the second derivative is the derivative of the first

$$\frac{\partial^2 f}{\partial x^2} = f''(x) = f(x+1) + f(x-1) - 2f(x)$$

First order derivatives usually result in *thicker* images while second order derivatives result in much finer detail overall. Point detection should use the second derivative, since they produce finer detail. We do this by using the *Laplacian*:

$$\nabla^2 f(x,y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

or basically adding the second derivatives both in the x and y axis. This basically reduces to:

$$f(x+1,y) + f(x,y+1) + f(x-1,y) + f(x,y-1) - 4f(x,y)$$

This means that we take the difference of the point with all its neighbors to see if the single point is a point of interest or not. This will measure if a point is really different than the neighboring points.

Line Detection

When detecting lines, we also can use the Laplacian mask. However, we have to be careful of the *double lining*, which occurs when the inside of the line is the same intensity and thus the second derivative produces zero while the outside of the line will be marked because the two sides are different in intensity.

Edge Models

We have a couple different types of edges. Perfect or ideal edges occur over one pixel and are very stark. Usually, edges are smoother and cover several pixels.

How do we detect edges in an image? We use the *gradient vector*, which can be defined as:

$$\nabla f \equiv [g_x, g_y] = [\partial x, \partial y]$$

which is going to represent the rate of change at a specific point. The *direction* in which the values change can be found by taking the magnitude of the gradient vector, by the Pythagorean theorem:

$$M(x, y) = \sqrt{g_x^2 + g_y^2}$$

Finding diagonal lines is also a challenge. For example, there are several possible masks that we can use, like the **Roberts**, **Prewitt**, and the **Sobel**. These masks help us calculate the gradient at a point. There are also versions of these operators for detecting diagonal edges.

There are some more complicated methods for edge detection, but they use complicated formulas such as

$$\nabla G(x, y) = [\frac{x^2 + y^2 - 2\phi^2}{\phi^4}]e^{\frac{x^2+y^2}{2\phi^2}}$$

How do we connect the points in the detected line once we find them? We can check the vicinity for other edge points, then compare their gradient and apply a threshold. Points that are isolated can be discarded and considered noise.

Canny Edge Detector

There are three objectives which makes this algorithm superior:

1. *Low error rate*: All edges should be found and there should be no missed edges or false positives.
2. *Edge points should be well localized*: The edges marked should be as close to the actual edges.

3. *Single edge point response*: There should only be one point per each true value in the edge.

The Canny edge detector will apply some threshold to the gradient magnitude[1] T and consider as valid edge pixels those that fall within that threshold and *are connected to other edge points*.

[1] We smooth the image first with a Gaussian filter for the same reason as LoG.

To calculate all the straight lines in an image, we could use the brute-force approach and compare the line passing through each candidate edge pixel. The complexity of the candidates would be $O(n^2)$, whereas the checking process would increase it to $O(n^3)$.

Hough Transform for Line Detection

If w erecall, the Hough Transform is useful for detecting lines and other shapes. We do this by using a form of polar cordinates

$$x\cos(\theta) + y\sin(\theta) = \rho$$

ρ measures the perpendicular distance between the line and the horizontal axis. Then θ is the angle measured from the vertical axis to the line denoted by ρ.

The idea behind the Hough transform is to use an accumulator structure that will keep track of which candidate edge points are the best estimation fo the line. We loop through all the edge points to find the one that best fits through the edge. The complexity is $O(nN_\theta)$.

This method is usually used for line detection since it provides a general way of detecting many different types of shapes. However, it can be a little rough around the edges, but this is nothing that some pre-processing cannot fix.

Segmentation by Thresholding

This method will just split an image based on the values in the histogram. We can base it on the peaks in the histogram and separate the pixels based on it. To do it, suppose the threshold is the halfway intensity value of the image. Then to choose the new threshold, we take the average value of the values with a greater intensity and lower intensity, respectively. $T = (m_1 + m_2)/2$.

Otsu's Method

This method selects the best threshold based on the *separability* between two groups of pixels. We need the *between-class variance* and the *global variance*. We try to select the value that maximizes the between-class variance and the global variance.

We can also select a threshold by selecting a threshold on a local area of an image.

The **Watershed algorithm** will "flood" an image by starting from the lower intensity values of an image to the greater values. Then, a group of connected components will be together in the same area. However, this algorithm might be bad due to over-segentation in an image.

Image Morphology

Morphology in image processing refers to extracting meaning from an image using some methods involving shapes. Operations on these images might involve reflections, rotations, and translations. The areas of interest inside an iamge are called **structuring elements**, and these are the building blocks that we use to deal with the shapes in these images.

Erosion and Dilation

Erosion can be expressed as

$$A \ominus B = \{z | (B)_z \subseteq A\}$$

which means that B can be **translated** by z and still be contained in A. Is this like choosing a subset of the image elements of A?

Dilation can be defined as

$$A \oplus B = \{z | (\hat{B})_z \cap A \neq \emptyset\}$$

which means that is the set of translations such that B, rotated around its axis and translated by z is still contained in A. With dilation we place the center of the structuring element, then we extend the image by whatever values in the structuring element are not yet placed in the final image. The result is a bigger image than at first, since we are including the new elements that were not there originally.

Opening and Closing

Erosion and dilation are the fundamental operation in opening and closing an image.**Opening** an image generally closes small gaps and breaks down isthmuses in an image. It is defined as

$$A \circ B = (A \ominus B) \oplus B$$

The first operation is to remove some of the outermost elements of the image with erosion. If we choose our structuring element B sensibly, we could then remove some of the rough edges will providing some smoothing. This operation essentially preserves the part of A where B can "roll around"

The **closing** of a set is represented as

$$A \bullet B = (A \oplus B) \ominus B$$

which means that the image is first dilated and then eroded. This operation can then be thought of as B rolling on the outside of the structuring element and rolling it flat from the outside.

These operations can be used to construct filters.

The **Hit-or-Miss** transformation chooses a seed point and then tries to expand the image by doing some dilation using the mask. This is reapeated until we find the figure in this subset of pixels.

By taking the difference of the original image with some structuring element, we can perform some basic boundary detection. For example,

$$\beta(A) = A - (A \ominus B)$$

serves as a boundary detector because the erosion will get rid of the outer elements, and subtracting the eroded image from the big image will result in the pixels on the border.

Data Compression

The goal to of data compression is to reduce the amount of information that an image takes up. *Information* is not the same as the *data* that is sent. Data is the means by which the information is sent across the network. Some data might be redundant, meaning that it contributes nothign useful to the thing we're doing

We can measure the amount of compression on the image by:

$$R = 1 - \frac{1}{C}$$

Where C is defined as $C = \frac{b}{b'}$.

There are three types of data redundancy:

- **Coding Redundancy**: *Codes* are the set of symbols used to convey information.
- **Spatial Redundancy**: Because in an image, nearby pixels are related, sometimes the same information is repeated.
- **Irrelevant Information**: Human visual systems sometimes can't tell the difference between some small details stored in images, but computers will store these values.

Once we select a code, or a way of representing the data, we can get the average code length by:

$$L_{avg} = l(r_k)p_r(k)$$

where r_k is the code length of the symbol and $p_r(k)$ is the probability of getting a certain symbol. The quickest way fo reducing the information required for encoding an image is reducing the irrelevant information, but how do we do that? We can get rid of the bit planes of the image, among others. There are also some ways of quantifying how much information we lost, we have the **mean square error** and **signal to noise ratio**

$$e_{rms} = [\frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}[\hat{f}(x,y) - f(x,y)]^2]$$

This basically measures the distance we were "off" by for each of the data points.

$$SNR_{ms} = \frac{\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}[\hat{f}(x,y)]^2}{\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}[\hat{f}(x,y) - f(x,y)]^2}$$