

Week 2: Fourier Transforms, Image Convolution, Edge detection

Welcome to my notes on computer vision! Please see the readme for details about the class and my reasons for taking it. All credit for this material's organization goes to the creators of the course! I'm just taking notes :)

As you may have noticed, I missed taking online notes on week 1 (color geometry and camera sensing) because I hadn't decided to post my notes at the time. To be completely honest, it was an easier section that (while foundational and interesting) will be less directly relevant to the rest of the course, so notes on it here aren't worth going back and creating.

With that out of the way, here I go into the first of our big topics!

Fourier transforms

Wow, this course didn't waste any time getting to difficult material! I'll do my best to review it here

- Why:
 - we will be using this on images later. Used in image compression, image analysis, and other related fields
 - This stuff is often useful for sound processing (not what we will be doing here though)
- What:
 - Discrete fourier transforms:
 - It decomposes an image into sine and cosine components. This creates a new image in the frequency domain (with the og image being in the spatial domain)
 - Continuous fourier transforms: similar to the dft, but with an integral to do the summation (not used on images because pixels are discrete values, so we won't be using this as much)
- How:
 - There is a formula for the DFT:

$$F(k, l) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} f(i, j) e^{-i2\pi(\frac{ki}{N} + \frac{lj}{N})}$$

- What took me awhile to understand was the reason that imaginary numbers are coming into play here. I had to go to outside resources, but the video on youtube

by 3blue1brown on fourier transforms was incredibly helpful for developing intuition for the topic. The video deals with continuous fourier transforms, but the similarities to the DFT are obvious. My summary of the video is below

- First, wrap the signal (imagine some kinda sine wave) around a circle (this is done mathematically by the e^{-i})
 - The winding frequency impacts how the new graph looks
 - When winding frequency matches og frequency, we get all of the high points on the right and the low points on the left (his visualization helps here, I recommend watching that if you are reading these notes)
- Calculate the “center of mass” of the new graph (it'll go around the center except when the og frequencies are present. When they are present, the center of mass is MUCH further right)
- We try all the different winding frequencies, calculating the center of masses. Plot that (2 dimensions: center of mass and winding frq).
- The spots where the center of mass was higher are the frequencies of the original signal(s)! Now we have a way to take combined multiple signals and find the original frequencies of EACH signal. Wow.

The transform of a pure frq peaks at one point (where frq matches the og). The transform of a combined frequency peaks at the frequencies of the original signals.

- But this isn't the entire mathematical buildup. So here are notes on that, step by step.
- $e^{(\text{somenumber} \cdot i)}$ is the same as walking that many units around a circle in the complex plane. (euler's formula)
- $e^{(2\pi \cdot i)}$ is the walking around the circle (see og formula for context)
- $e^{(2\pi \cdot i \cdot f \cdot t)}$ is the winding up the graph accounting for frequency and time
- We use $e^{(-2\pi \cdot i \cdot f \cdot t)}$ to go clockwise. (at this stage, we r walking around the circle)
- We multiply the function by this circle at each point to get the winded up graph!
- To approximate the center of mass: sample a bunch of points, add em all up, divide by number of points. (this is what the mathematics is doing- for cft using integrals, for dft)
 - For the real ft, don't divide by the time interval. That way, we can easily see the spots where the center of mass is much higher
 - Graph this as $g(f)$. This takes frequency as an input! With the center of mass on the y axis.

To be completely honest, as of now I feel I understand what it is but I haven't seen the applications of this to computer vision yet (although I see how this has powerful implications for signal processing). Hopefully the course will go into that. I got a 94% on the quiz for 2.1 (missed 1 out of 17).

Now that we have a background on that, lets move to the next big topic for this week:

Image Convolution and Edge Detection

- Why: edge detection can detect boundaries between things in an image.
- What and how:

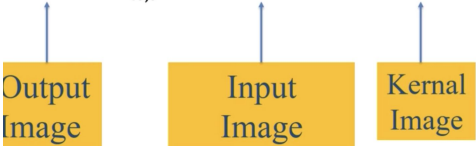
Video 2.9 (there are a lot of videos, so i'm breaking it up like this)

- Image filtering is like sliding a "mask" (a matrix, called the kernel) around the original image, taking the weighted sum (multiplying the kernel value at a spot by the image value at a spot and summing that up) and replacing the og image value with the weighted sum. (later, this is the convolution kernel! Filtering and convolution can essentially do the same thing)
- This is similar to a neural network (some signals get amplified or dampened. Depends on kernel!)
- $I(x)$ =input image
- $f(x)$ is the kernel
- $g(x)$ is the output image after filtering
- Convolution is based on the eye; it is based on how we see (retina, rods, cones, etc. See a basic psych course if you don't know this)

Video 2.10

- Using a 3x3 kernel as shown below will blur the image horizontally:
 - 0 0 0
 - .3 .3 .3
 - 0 0 0
- Using a 3x3 kernel as shown below will blur the image vertically:
 - 0 .3 0
 - 0 .3 0
 - 0 .3 0
- We can used kernals where we subtract. For example if we use the kernel below, we can double the brightness of the middle pixel while removing horizontal blur: (this pulls out a lot more sharpness and details from the image! Wow. They showed an example and it was crazy)
 - 0 0 0 0 0 0
 - 0 2 0 - .3 .3 .3
 - 0 0 0 0 0 0

Image filtering

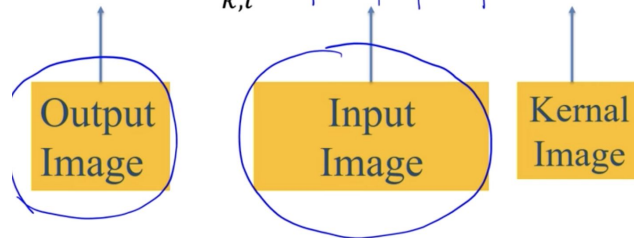
$$g[m, n] = \sum_{k, l} I(m + k, n + l) * f(k, l)$$


- Mathematically, image filtering is as shown above. We need this in order to make a computer do it! But conceptually this is the same as sliding the kernel around and taking the weighted averages.

Video 2.11

- Convolution IS NOT filtering. With filtering, the output kernel is flipped left-right up-down. Convolution is filtering with kernel flipped.
 - We see this through an impulse kernel which is just 0 with 1 in the center. That flips everything! (kinda like retina flipping everything. Weird)
 - Convolution formula shown below (basically we look at it backwards)

Convolution $I \otimes f$

$$g[m, n] = \sum_{k, l} I(m - k, n - l) * f(k, l)$$


-
- Convolution is commutative (i convolved with f = f convolved with i) (showed proof in video)
- Convolution has linear independence. So we can take the image apart and then reassemble it! We don't need to do it all in one step (we could do two convolutions with the decomposed mask then add it together. This helps with huge data because we can go step by step.
- Convolution is reversible

Video 2.12 (mostly technical details related to matlab implementation)

- When looking at the edges, we do not want to assume pictures just go blank outside the frame! There are several other options that will improve our convolution (we put fake pixels around the image so that we use a less unrealistic image- real pictures don't have black around the outside of the pictures!!!)
 - We could just copy the edges and use it again
 - We could copy the edges, flip it, and use it again
 - We could do several other things: warp, wrap, clamp, mirror, blurred zero, etc. There's a bunch of ways but the key idea is DO NOT leave it blank. At least try to make it realistic
 - We do these operations using for loops (not the most efficient, but it works. Matlab gives us the image as an array) (gotta be careful using the loops though! Think about boundaries etc. Not too hard though)

Video 2.13 Common filters for convolution

- The gaussian (essentially a bump. It's the normal distribution from stats)
 - This does image smoothing and can remove noise. It can also reveal underlying function (even tho it makes it blurry, as we saw in the example)
- The derivative of gaussian
 - Two bumps; one up one down
 - Used to extract edges!!!! (in canny edge detection specifically)
- Laplacian of gaussian (the mexican hat operator)
 - Looks like an upside down sombrero
 - Used for texture properties of the image

There was a quiz 2.2 at this point, and I got a 100% (6/6) :)

NOW ON TO EDGE DETECTION :)

Video 2.14-2.16

- We are trying to take the partial derivative with respect to x and the partial derivative with respect to y and then (after taking the magnitude of this gradient) add them up to get the edges all outlined on an image.
- For canny edge detector function in matlab, it returns a b/w thing where 1=edge 0=not edge
 - For his ex of the trees, we extracted textures inside the tree. These are texture edges, and later we will learn to bring them out or conceal them!
 - Canny edge detection is 5 steps
 - 1- filter images by derivatives of gaussian (the two bump)
 - We filter out noise and take the derivative
 - I_x and I_y . Approximate these by using convolution with the following kernels (the kernel looks like two bumps!). In matlab we use $I_x = \text{conv2}(I, dx, 'same')$ $I_y = \text{conv2}(I, dy, 'same')$
 - I_x detects the vertical edges

- I_y detects the horizontal edges
- I_x and I_y are grayscaled
- 2- compute magnitude of gradient
 - Standard formula as shown below
 - $\text{sqrt}(I_x^2 + I_y^2)$
 - Now we will make it b/w. At this stage, the image looks like it detects edges. But we (the computer) don't know which are technically considered edges or where they are.
- 3- compute edge orientation (which way is the edge?)
 - For the orientation of the edge, we need math:

The image gradient direction is given by:

$$\theta = \tan^{-1} \left(\frac{\partial f}{\partial y} / \frac{\partial f}{\partial x} \right)$$

– how does this relate to the direction of the edge?

$$\theta_{edge} = \tan^{-1} \left(-\frac{\delta f}{\delta x} / \frac{\delta f}{\delta y} \right)$$

-
- Display the image gradient flow:
 - `figure(2); clf; imagesc(I); colormap(gray); axis image; hold on;` (this sets everything up and lets us draw on top of it)
 - `quiver(Ix, Iy);` (this draws a vector for every pixel. The gradient!)
 - Reminder from calc 3: the gradient is the direction to walk to increase brightness the fastest (in this case)
 - `Quiver(-Iy, Ix, 'r')`
 - `Quiver(Iy, -Ix, 'r')`
 - The last two quivers had the goal of finding edge orientation. Edge detection is perpendicular to gradient vector!
- 4- detect local maximums
 - We use a `colormap(hot)` to display this! The hotter it is, the more likely to be an edge
 - We can display a vector field on the image using
 - Note: the scale of the image comes into play here. If we shrink an image and do the same process, the edges will look different even if we do the same stuff! With bigger scale images we can see bigger neighborhoods in the physical domain (as compared to pixel domain) (if we keep the filter the same size each time, then as the image shrinks it gets blurrier. This shows the rough idea but is much less precise)

- We want to do enough maximum suppression so that only a few pixels are labelled as edges (essentially finding local maximums, same way as we would in calc 3. Review that for anyone confused)
- 5- edge linking
- Video 2.16 went over a real world example of everything we talked about. Not any relevant new content in the big picture but it was good review
 - We did learn that instead of using loops, we can use meshgrid! Parallel processing on all the pixels at once. `meshgrid(1:5, 1:3)` for example. The first parameter specifies x rows and the second specifies how many columns.
 - When we do convolution, do not use the looping stuff! Use meshgrid (he has a fancy code to do that. We can do the filtering without using a loop. This does the same thing as the loop in around half the running time)

Wow that was longer and took more time to do than I expected. But this is the first real application of the theory that we're getting to, so I was excited to learn about it! Matlab has many edge detectors already implemented, so this won't be directly useful in code past what's already been done (no sense re-implementing it when I can just call a function). It is, however, a good preview of the types of things computer vision can do! See you in week three (hopefully someone is reading haha).