CS 3430: SciComp with Py
Assignment 08
Edge and Line Detection

Vladimir Kulyukin
Department of Computer Science
Utah State University

March 14, 2020

Learning Objectives

1. Edge Detection
2. Line Detection
3. Hough Transform

Introduction

In this assignment, we'll implement the simple edge detection algorithm we learned in lectures 14 and 15 and then use it to implement a simple version of the Hough Transform (HT) algorithm from lectures 15 and 16. You'll need to install the Python Image Library (PIL) for this assignment. Go to `https://python-pillow.org/` to install PIL on your OS. You may also want to install OpenCV (`www.opencv.org`) to run the OpenCV code samples from lectures 15 and 16 to see how OpenCV handles edge detection and HT.

You'll save your solutions in `cs3430_s20_hw08.py` and submit it in Canvas. I've included in the homework zip my unit tests `cs3430_s20_hw08_uts.py` for this assignment. Unlike the unit tests for the previous assignments, these unit tests don't contain assertions. Edge and line detection are as much of an art as a science. A useful edge or line in one domain (e.g., road lane detection) is useless in a different one (e.g., plant classification).

Problem 01: (2 points)

Implement the function `depil(pil_img, default_delta=1.0, magn_thresh=20)` in `cs3430_s20_hw08.py`. This function takes a PIL image `pil_img` and applies the edge detection algorithm from Lecture 14. It returns a new one channel PIL image where the edge pixels are labeled as white (i.e., their pixel value is 255) and non-edge pixels are labeled as black (i.e., their pixel value is 0). The keyword parameter `default_delta` is used in computing the vertical and horizontal luminosity changes at pixels. The `magn_thresh` specifies the value of the gradient's magnitude at a pixel for the pixel to qualify as an edge pixel. When you work on this function, remember that in PIL pixels are referenced in a column-first manner. In other words, a pixel at (x, y) is at column x and row y.

Let's implement this function step by step. Start by implementing the function `pil_pix_dxdy(pil_img, cr, default_delta)` in `cs3430_s20_hw08.py` that returns a 2-tuple (dx, dy) of the horizontal and vertical changes in luminosity at a pixel in a PIL image `pil_img` whose

position is specified by the 2-tuple `cr` where `cr[0]` is the pixel's column and `cr[1]` is the pixel's row. The luminosity values are computed with the relative luminosity formula we discussed in Lecture 14. The formula is implemented in the `lumin` function in `cs3430_s20_hw08.py`.

Recall from Lecture 14 that the vertical change (i.e., dy) is computed as the difference between the luminosities of the pixel's upper and lower neighbors (i.e., the two pixels at positions `(cr[0],cr[1]-1)` and `cr[0],cr[1]+1)` and the horizontal change (i.e., dx) is computed as the difference between the luminosities of the pixel's right and left neighbors (i.e., the two pixels at positions `(cr[0]+1,cr[1])` and `(cr[0]-1,cr[1])`.

Implement `grd_magn(dx, dy)` that takes the dx and dy values computed by `pil_pix_dxdy()` and computes the gradient's magnitude from them. Proceed with the implementation of the function `grd_deg_theta(dx, dy)` that computes the theta (i.e., the orientation of the gradient) from dx and dy. This function should return the degree value. Why degrees? What's wrong with radians? Absolutely nothing! However, in my opinion, it's easier to read and debug degrees than radians.

Everything is in place now for `depil()`. As you iterate through the image's pixel and compute magnitude and orientation of the gradient at each pixel, you may want to covert the magnitude and orientation values to integers. Also, throw in an assertion `assert abs(th) <= 180` to make sure that your orientations make sense. The function `depil()` thresholds only on magnitudes but, when you're debugging, orientations are a great help. You can add orientation thresholding to your implementation in the future.

The file `cs3430_s20_hw08_uts.py` has 20 unit tests for Problem 01. Each test corresponds to a separate image. All the test images are in the directory `imgs`. The directory `my_output_imgs` contains the images that my code generates for each unit test. The unit tests, when run, save your output images in the directory `output_imgs`. Your output images should look similar to mine. When we run the unit tests, we won't be comparing your output images to mine pixel by pixel for exact matches. But, your images should look sufficiently similar, especially the edge pixels detected in simple images such as `EdgeImage_01.jpg`.

Problem 02: (3 points)

Imlement the function `ht(pil_img, angle_step=1, pix_val_thresh=5)` that takes a PIL image `pil_img` returned by `depil()`, runs the HT algorithm discussed in Lectures 15 and 16, and returns the HT table. The keyword parameter `angle_step` specifies the theta degree increment of the theta dimension of the table. In your implementation, the theta values should go from 0 to 359 in increments of 1. The second keyword parameter, `pix_val_thresh`, specifies the value at which the pixel is considered an edge. There's no need to change the default value for this assignment. It'll work. Actually, it's not that important in the context of this assignment, because `depil()` makes each pixel in the image to be 0 or 255. However, if in the future you want to add more sophisticated marking of edge pixels (i.e., the more edgy a pixel is, the closer its value approaches 255), you can use this keyword to play with different thresholding schemes.

The method `__test_ht()` in `cs3430_s20_hw08_uts.py` shows a standard way to combine edge detection with the HT. We open an image, return the image with edge and non-edge pixels marked, apply the HT to the image with the marked pixels, and delete the image after the HT table is computed. The last step is optional, of course, in case you want to hold on to your edge image for later use.

```
def __test_ht(self, inpath, default_delta=1.0, magn_thresh=20, spl=200):
    img = Image.open(inpath)
    img = depil(img, default_delta=default_delta, magn_thresh=magn_thresh)
    ht_table = ht(img, angle_step=1)
    del img
```

...

As was discussed in Lecture 16, the HT table can be implemented as a 2-dimensional array or a dictionary. In my implementation, it's a dictionary. But you can use either data structure. The unit tests don't check for the data type.

On to the final cut! Implement the function `ht_find_lines(htb, spl=1)` that takes the HT table `htb` computed by `ht()` and returns an array of all lines detected in `htb` whose support level is at least at the value specified by the `spl` parameter. Each line in the returned list is represented by a 3-tuple `(rho, angle, spl)`, where `rho` is the length of the normal from the origin to the line, `angle` is the angle (in degrees) between the x-axis and the normal, and `spl` is the line support level. The values `rho` and `angle` are computed with respect to the centered coordinate system (i.e., (0, 0) is in the middle of the image at `(int(w/2), int(h/2))`, where `w` and `h` are the width and height of the image, respectively. The lines with negative rho values should be ignored. The returned list of lines is sorted by the angle from smallest to largest for easier interpretation.

The file `cs3430_s20_hw08_uts.py` contains 10 unit tests for Problem 02. Before each unit test I pasted in a multi-line comment what my implementation of `ht()` outputs for the test. Let's take a look at a couple of examples. Running `test_hw08_prob02_ut01()` produces the following output.

```
Lines found in imgs/EdgeImage_01.jpg:
[(13, 90, 270), (12, 90, 270)]
```

The above output means that in `EdgeImage_01.jpg` two lines are detected with respect to the origin in the center of the image. The first line has a rho of 13, a theta of 90 degrees, and a support level of 270. The second line has a rho of 12, a theta of 90, and a support level of 270. These two lines are basically the same. Running `test_hw08_prob02_ut06()` produces the following output.

```
Lines found in imgs/EdgeImage_06.jpg:
[(50, 0, 298), (51, 0, 298), (50, 180, 298), (49, 180, 298)]
```

The above output means that in `EdgeImage_06.jpg` two lines are detected with respect to the origin in the center of the image. The first line has a rho of 50, a theta of 0 degrees, and a support level of 298. The second line has a rho of 51, a theta of 0, and a support level of 298. The two lines are basically the same. The third line has a rho of 50, a theta of 180, and a support level of 298. The fourth line has a rho of 49, a theta of 180, and a support level of 298. The third and fourth lines are basically the same.

Your lines will, most likely, be slightly different, but they should be in the same ballpark. In addition to the lecture pdfs, code samples, and your class notes, I included in the references for this assignment (see below) a link to an OpenCV tutorial on the HT and a pdf with the original and very accessible article on the HT by Duda and Hart.

If you decide to do additional reading/research on the HT, always remember the **know-thy-origin principle**. When you read a blog post, a documentation page, or a paper about the HT, figure out first and foremost the origin of the coordinate system with respect to which the rho and theta values of the lines are computed. Some authors never state it explicitly, which is bad. I've seen three different origins in HT implementations: 1) the top left corner of the image; 2) the middle of the image (this is what we do in this assignment); and 3) the bottom left corner of the image. I personally have never seen the origin placed in the top right or bottom right corner. But, both origins are theoretically possible.

What To Submit

Submit your code in `cs3430_s20_hw08.py`. Don't submit your edge images. We'll run the unit tests on your code to see what images it generates.

Happy Hacking!

References

1. `https://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/hough_lines/hough_lines.html`
2. R. Duda, P. Hart. "Use of Hough Transformation to Detect Lines and Curves in Pictures." (pdf included)