CS 3430: Scientific Computing
Assignment 06
Differentiation and Edge Pixel Detection

Vladimir Kulyukin
Department of Computer Science
Utah State University

February 24, 2024

Learning Objectives

1. Edges as Derivatives of Luminosity
2. Edge Pixel Detection
3. Image Processing

Introduction

Edge detection is one of the domains where differentiation ideas are used all the time, because detecting edges can be thought of as taking derivatives of luminosity or, which is equivalent, detecting changes in luminosity. In this assignment, we will implement the simple edge pixel detection algorithm we learned in Lectures 12 (cf. `cs3430_s24_lec12_edge_detection.pdf` in the Lecture 12 zip in Canvas).

You will save your solutions in `cs3430_s24_hw06.py` and submit this file in Canvas. My unit tests are in `cs3430_s24_hw06_uts.py`. The unit tests for Problem 2 do not contain assertions, because edge detection is not exactl science: a useful edge in one domain (e.g., road lane detection) is useless in a different one (e.g., plant classification). It is all about context.

You will need to install the Python Image Library (PIL) for this assignment, which you can do by following instructions at python-pillow.org or pillow.readthedocs.io. I ran the unit tests for this assignment in Python 3.6.7 and PIL 6.0.0, an older version of PIL, and then in Python 3.11.2 and PIL 10.2.0. All passed without a problem in both combinations. PIL is another useful scientific computing tool that you should add to your stack. It is not nearly as involved and sophisticated as OpenCV, another open source image processing and computer vision library. But, PIL is very handy for rapid prototyping, visualization, and image generation.

Coding Lab

The zip archive for this assignment contains the folder `imgs` where I put some jpg and png images. Some of these images are from my past and current research project, some from free duckduckgo image repos. We will use them in this assignment. After we install PIL, we can read in images as follows.

```
>>> import PIL
>>> from PIL import Image
>>> img1 = Image.open('imgs/EdgeImage_01.jpg')
```

```
>>> type(img1)
<class 'PIL.JpegImagePlugin.JpegImageFile'>
>>> img1.size
(272, 328)
```

The image `EdgeImage_01.jpg` is the left image in Figure 1. I synthetically generated it to have the white upper part and the black lower part. If we want to test an edge detection technique, it had better work on images like this one. Once read, the image is an matrix of pixels. How many values each pixel contains depends on the format of an image. In a JPG image, each pixel is typicaly a 3-tuple (r, g, b), where $0 \leq r, g, b \leq 255$ with 0 denoting the lowest end of the color spectrum and The last expression the the code segment above, i.e., `img1.size`, returns the size of the pixel matrix. In PIL, the first number is the width and the second number is the height. So the PIL Image object stored in `img1` has 272 columns and 328 rows. The count of columns and rows in PIL starts at 0 from the top left corner.

Figure 1: The original image EdgeImage_01.jpg (left); the image with the edge pixels detected in Edge_Image_01.jpg by the method of Problem 2.

Here is how to reference individual pixels.

```
>>> img1.getpixel((0,0))
(255, 255, 255)
>>> img1.getpixel((1,0))
(255, 255, 255)
>>> img1.getpixel((2,0))
(255, 255, 255)
>>> img1.getpixel((3,0))
(255, 255, 255)
>>> img1.size
(272, 328)
>>> img1.getpixel((271,0))
(255, 255, 255)
### this call results in an exception
>>> img1.getpixel((272,0))
IndexError: image index out of range
```

There are a few debugging/exploration utility functions in `cs3430_s24_hw06.py`. Among those are `display_pil_img_row` and `display_pil_img_col`.

```
def display_pil_img_row(pil_img, r):
    ncols, _ = pil_img.size
    for c in range(ncols):
        print(pil_img.getpixel((c, r)))

def display_pil_img_col(pil_img, c):
    _, nrows = pil_img.size
    for r in range(nrows):
        print(pil_img.getpixel((c, r)))
```

Let us use it on `img1`.

```
>>> from cs3430_s24_hw06 import *
>>> display_pil_img_row(img1, 0)
(255, 255, 255)
(255, 255, 255)
...
(255, 255, 255)
```

The above call prints out 272 RGB 3-tuples (255,255,255), because row 0 of this image, as we can see in Figure 1, consists of only white pixels. Let us call it on the last row, i.e., row 327.

```
>>> display_pil_img_row(img1, 327)
(0, 0, 0)
(0, 0, 0)
...
(0, 0, 0)
>>>
```

All 272 RGB 3-tuples are (0,0,0), i.e., black. Images with 3-value pixels are called 3-channel. When we detect edge pixels, we typically work with grayscale images. Grayscale images, again typically, but not always are 1-channel images, i.e., each pixel contains an integer between 0 and 255, which represents the luminosity at that pixel. Let us learn how to create a 1-channel image in PIL.

```
>>> img1_l = Image.new('L', img1.size)
>>> img1_l.size
(272, 328)
>>> img1_l.getpixel((0, 0))
0
>>> img1_l.getpixel((1, 0))
0
```

We created `img1_l` as a 1-channel image of the same size as `img1`. All pixel values are black by default. Here is how we can change the value of a pixel in an image.

```
>>> img1_l.putpixel((1,0), 10)
>>> img1_l.getpixel((1,0))
10
```

Let us attempt to do something crazy with pixel values.

```
>>> img1_l.putpixel((1,0), 300)
>>> img1_l.getpixel((1,0))
255
>>> img1_l.putpixel((1,0), -1000)
>>> img1_l.getpixel((1,0))
0
```

Fortunately for us PIL keeps us sane by keeping the pixel values between 0 and 255. There is another utility boolean function, `is_in_pil_range`, in `cs3430_s24_hw06.py` that returns True if both values in a 2-tuple `(column, row)` passed to the function as the 2nd argument reference a pixel that is not on the border of the image object `img` passed to the function in the first argument. A border pixel is a pixel in a border row or colum, i.e., first or last.

```
>>> is_in_pil_range(img1, (0,0))
False
>>> is_in_pil_range(img1, (1,1))
True
>>> img1.size
(272, 328)
>>> is_in_pil_range(img1, (1,327))
False
>>> is_in_pil_range(img1, (1,326))
True
>>> is_in_pil_range(img1, (271,327))
False
>>> is_in_pil_range(img1, (1,327))
False
```

Problem 1 (2 points)

We will implement the edge pixel detection algorithm step by step.

Implement the function `pil_pix_dxdy(pil_img, cr, default_delta)` in `cs3430_s24_hw06.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 implemented in the `lumin` function in `cs3430_s24_hw06.py`.

Recall from Lecture 12 that the vertical change (i.e., Dy) is computed as the difference between the luminosities of the pixel's upper and lower neighbors and the horizontal change (i.e., Dx) is computed as the difference between the luminosities of the pixel's right and left neighbors. Then the pixel where we are estimating the change in luminosity is placed inside the *change* triangle (cf. Slide 05 in `cs3430_s24_lec12_edge_detection.pdf` in the Lecture 12 zip in Canvas).

When Dx or Dy is 0, it is set to `default_delta`, e.g., 1. For example, when the value of the pixels above and below the pixel which we are testing for the presence of an edge (at that pixel) are the same, then Dx or Dy are set to this value. Thus, if the pixel values left and right of the pixel tested for the presence of an edge are 255 and default_delta is set to 1, then

$$Dx \quad = \quad 255 - 255 = 1.$$

This helps us avoid the division by 0. We can set it to an arbibrarily small values, e.g., 1e-100, but then we may have to face some numerical instability.

Let us see how `pil_pix_dxdy` works at various pixels in different images.

```
>>> img1.getpixel((1,151))
(255, 255, 255)
>>> img1.getpixel((1,152))
(0, 0, 0)
>>> img1.getpixel((1,153))
(0, 0, 0)
>>> dx, dy = pil_pix_dxdy(img1, (1,152), 1)
>>> dx
1
dy = 254.99999999999997)
```

In the above code segment, we see that the three consecutive pixels in column 1 of `img1` are (255,255,255), (0,0,0), and (0,0,0). So, we have a change from white to black, and the horizontal change, dx, is 1 whereas the vertical change is 255. What if the pixels are identical? Then both dx and dy are set to `default_delta` whatever the latter is set to.

```
>>> img1.getpixel((1,150))
(255, 255, 255)
>>> img1.getpixel((1,149))
(255, 255, 255)
>>> pil_pix_dxdy(img1, (1,149), 1)
(1, 1)
>>> pil_pix_dxdy(img1, (1,149), 0.000001)
(1e-06, 1e-06)
>>> pil_pix_dxdy(img1, (1,149), 1.e-06)
(1e-06, 1e-06)
```

Now implement the function `grd_magn(dx, dy)`, where dx and dy refer to Dy and Dx, respectively, in Slide 05 in `cs3430_s24_lec12_edge_detection.pdf` and computed by `pil_pix_dxdy()`. This function computes the gradient's magnitude from them given on the same slide.

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? Nothing whatsoever! However, in my opinion, it is easier to debug degrees than radians. We can convert back to radians after the code is working. Here's a test of `grd_magn` and `grd_deg_theta`, where the change from white to black has the orientation of positive 90. This is arbitrary, of course, but as long as we are consistent, we are fine.

```
>>> dx, dy = pil_pix_dxdy(img1, (1,152), 1)
>>> dx
1
>>> dy
254.99999999999997
>>> grd_magn(dx,dy)
255.00196077677518
>>> grd_deg_theta(dx,dy)
90
```

Figure 2: The original image EdgeImage_06.jpg (left); the image with the edge pixels detected in Edge_Image_01.jpg by the method of Problem 2.

Let us take a look at EdgeImage_06.jpg in the left of Figure 2 where each row switches first from black to white and then from white back to black.

We can explore row 1 to see the pixels where these changes happen as follows.

```
>>> for c in range(0, img6.size[0]):
...     print((c, img6.getpixel((c,1))))
...
(0, (0, 0, 0))
(1, (0, 0, 0))
...
(98, (0, 0, 0))
(99, (0, 0, 0))
(100, (4, 4, 4))
(101, (252, 252, 252))
(102, (255, 255, 255))
(103, (254, 254, 254))
...
(199, (255, 255, 255))
(200, (254, 254, 254))
(201, (3, 3, 3))
(202, (0, 0, 0))
(203, (2, 2, 2))
(204, (0, 0, 0))
(205, (0, 0, 0))
(206, (1, 1, 1))
...
(299, (0, 0, 0))
```

So, the change from white to black happens at the pixels (199,1), (200,1), and (201,1), which we can

verify as follows.

```
>>> img6.getpixel((199,1))
(255, 255, 255)
>>> img6.getpixel((200,1))
(254, 254, 254)
>>> img6.getpixel((201,1))
(3, 3, 3)
```

Let us compute dx, dy at the pixel (200,1) and then compute its `grd_magn` and `grd_deg_theta`, i.e., the gradient magnitude and orientation.

```
>>> dx,dy = pil_pix_dxdy(img6, (200,1), 1)
>>> dx
-251.99999999999997
>>> dy
1
>>> grd_magn(dx,dy)
252.00198411917313
>>> grd_deg_theta(dx,dy)
180
```

So, the horizontal chage dx is large compared to the vertical change dy and the orientation of the gradient is 180, which makes sense.

Now let us explore what happens when we switch from black back to white and compute the magnitude and orientation at a pixel where that change happens.

```
>>> img6.getpixel((100,1))
(4, 4, 4)
>>> img6.getpixel((101,1))
(252, 252, 252)
>>> img6.getpixel((102,1))
(255, 255, 255)
>>> dx,dy = pil_pix_dxdy(img6, (101,1), 1)
>>> dx
250.99999999999997
>>> dy
1
>>> grd_magn(dx,dy)
251.00199202396777
>>> grd_deg_theta(dx,dy)
1
```

So, the change from black to white at the pixel (101,1) results in the gradient magnitude of ≈ 255 and the gradient orientation of $1 \approx 0$.

Problem 2 (3 points)

Everything is in place now for us to implement the function `dex_pil(pil_img, default_delta=1, magn_thresh=20)` in `cs3430_s24_hw06.py`. The function name abbreviates the phrase "detect edge pixels with PIL." This function takes a PIL image `pil_img` and applies the edge detection algorithm

from Lecture 12 (cf. Slide 07 in `cs3430_s24_lec12_edge_detection.pdf` in the zip of Lecture 12 in Canvas).

This function 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 the threshold value for computing the vertical and horizontal luminosity changes at pixels. The keyword parameter `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 individual pixels are referenced in a column-first manner, i.e., a pixel at (x, y) is at column x and row y.

You may want to throw in an assertion `assert abs(th) <= 180` to make sure that your orientations make sense. The function `dex_pil()` thresholds only on gradient magnitudes, but, when debugging, orientations can be helpful. Some edge detection algorithms add orientation thresholding, e.g., road lane detection.

The file `cs3430_s24_hw06_uts.py` has a few unit tests for this problem. Each test corresponds to a separate image. All the test images should be in the directory `imgs`. The directory `my_output_imgs` contains the images that my code generated for each unit test. The unit tests, when you run them, save your output images in the directory `output_imgs`. Your output images should look similar to mine. When I run the unit tests to grade your submission, I will not 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, e.g., the right images in Figures 1 and 2. The folder `out_imgs` contains my images with edge pixels marked white. Figures 3, 4, and 5 are some of them.

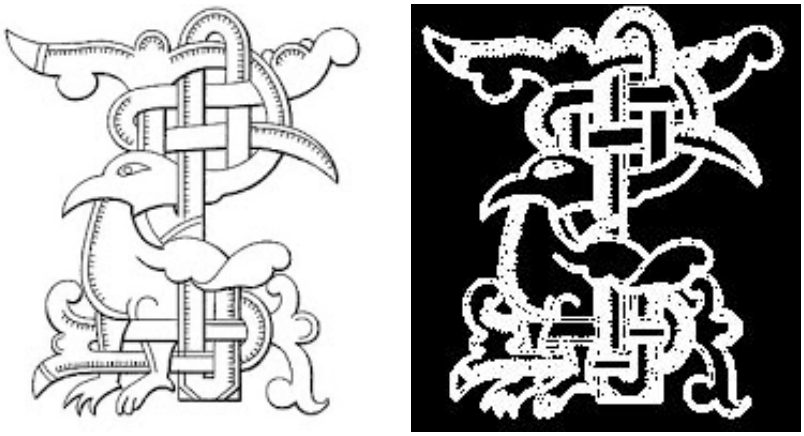

Figure 3: The original BirdOrnament.jpg image (left) and the output image with the edge pixels marked white (right)

What To Submit

Submit your code in `cs3430_s24_hw06.py` in Canvas.

Happy Hacking and Image Processing!

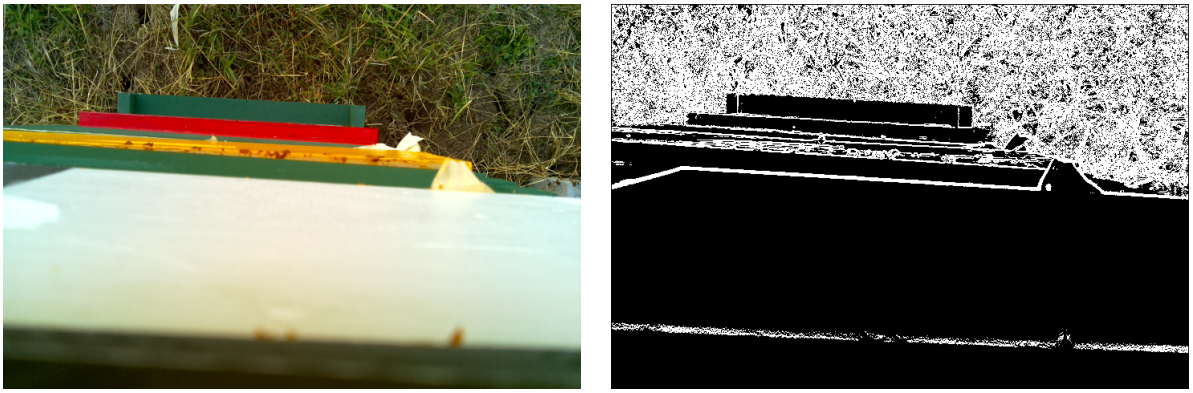

Figure 4: The original hive01.png image taken from the roof of a bee hive (left) and the output image with the detected edge pixels marked white. A quick look is sufficient to conclude that the grass texture has a lot of edges, which helps in eliminating it from subsequent processing when necessary.

Figure 5: The original elephant.jpg image (left) and the image with the detected edge pixels marked as white. I love that bird above the elephant's head!