CS 3430: S19: SciComp with Py
Assignment 10
Detecting Edges with Gradients and Computing Image Similarity

Vladimir Kulyukin
Department of Computer Science
Utah State University

March 30, 2019

Learning Objectives

1. Gradients
2. Edge Detection
3. Image Similarity
4. PIL/PILLOW

Introduction

In this assignment, we'll implement the gradient-based edge detection algorithm we discussed in lectures 19 and 20. We could implement this algorithm with OpenCV images (i.e., numpy arrays). But we'll implement it with PIL/PILLOW, another Python image library. Granted that PIL/PILLOW is not nearly as powerful as OpenCV, it is smaller, more straightforward to use, and useful to have in one's repertoire of tools, especially for rapid prototyping.

Start by installing PIL/PILLOW. You should install PIL if you are working in Py2 or PILLOW if you are working in Py3. If you've checked out a rapsberry pi for this semester, you don't need to do anything, because PIL is already installed. I don't have any recommendations for Windows. You may want to try `https://wp.stolaf.edu/it/installing-pil-pillow-cimage-on-windows-and-mac/` or look for another technical blog with better instructions. If you want to install it on Linux, try "sudo apt-get install python-PIL" for Py2 or "sudo pip install pillow" for Py3.

To check if PIL/PILLOW is installed correctly, start your Python and do "import Image" in the Python shell. If there is no error message, your PIL/PILLOW is installed.

Let's do a few things to get started with PIL/PILLOW. There is the `img` folder in the zip of this assignment with a few images. All images I refer to in this text are in this folder. Here's how you can open images in PIL/PILLOW and get values of individual pixels. The first image is a binary image, which is why each pixel value is just an integer (0 or 255). The second image is an RGB image, which is why each pixel a 3-tuple of integers. Both images are shown in Fig. 1.

```
>>> img = Image.open('img/1b_bee_01_ed.png')
>>> img.getpixel((15, 10))
0
>>> img2 = Image.open('img/1b_bee_01.png')
>>> img2.getpixel((15, 10))
(190, 189, 182)
```

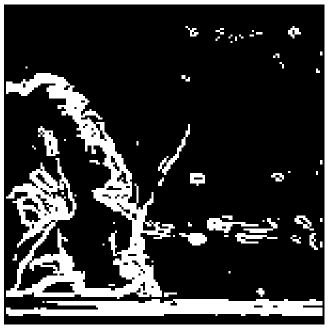
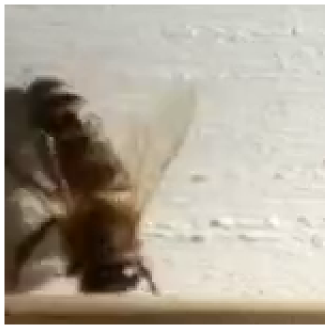

Figure 1: Binary image 1b_bee_01.png (left); RGB image 1b_bee_01.png (right).

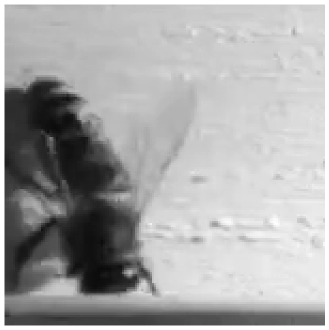

Figure 2: Binary image 1b_bee_01.png (left); RGB image 1b_bee_01_gray.png (right).

If you need to grayscale an image, here is how you can do it. The images are shown in Fig. 2.

```
img = Image.open('img/1b_bee_01.png').convert('LA')
img2 = img.save('img/1b_bee_01_gray.png')
```

When you implement the gradient-based edge detection algorithm, you'll need to crate an empty binary image where you'll set the values of edge pixels to 255 (white) and the values of non-edge pixels to 0 (black). Here's how you can do it.

```
>>> img = Image.new('L', (10, 10))
>>> img.getpixel((0, 0))
0
>>> for r in range(img.size[0]):
        for c in range(img.size[1]):
            assert img.getpixel((r, c)) == 0

>>> img.putpixel((0, 0), 255)
>>> img.getpixel((0, 0))
255
```

Problem 1: Detecting Edges (4 points)

Implement the function `gd_detect_edges(rgb_img, magn_thresh=20)` that takes an RGB PIL/PILLOW image and the keyword `magn_thresh` that specifies the value of the gradient magnitude's threshold and

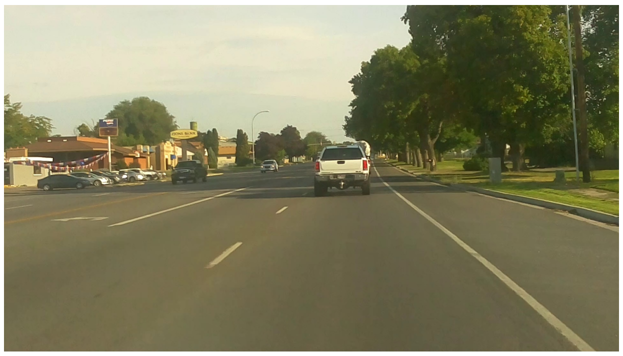
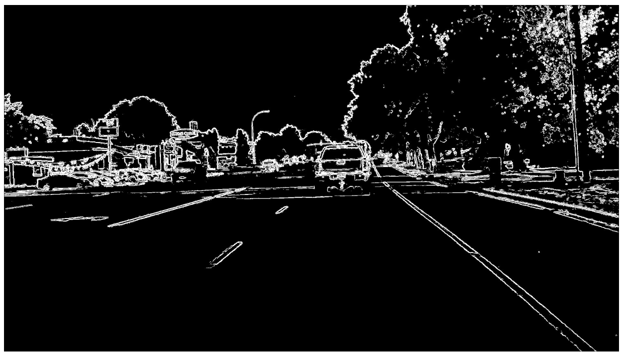

Figure 3: Original image output11885.jpg (left); image with detected edges saved as output11885_ed.jpg (right).

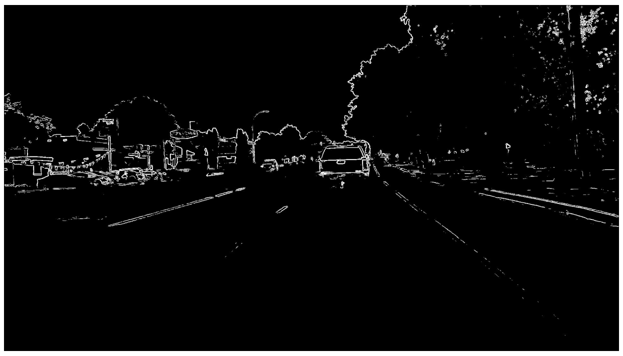

Figure 4: Original image output11885.jpg (left); image with detected edges saved as output11885_ed2.jpg (right).

returns a binary PIL/PILLOW image of the same size where the non-edge pixels are set to 0 and the edge pixels are set to 255. As we discussed in class, the magnitude threshold value specifies that only the pixels whose gradient magnitude is at least the value specified by the treshold (e.g., 20) qualify as edge pixels. The file `cs3430_s19_hw10.py` has the function `save_gd_edges` that you can use to test this function.

```
def save_gd_edges(input_fp, output_fp, magn_thresh=20):
    input_image  = Image.open(input_fp)
    output_image = gd_detect_edges(input_image, magn_thresh=magn_thresh)
    output_image.save(output_fp)
    del input_image
    del output_image
```

Let's detect edges in `output11885.jpg` and save the image with the detected edges in `output11885_ed.jpg`. Both images are shown in Fig. 3.

```
>>> save_gd_edges('img/output11885.jpg', 'img/output11885_ed.jpg', magn_thresh=20)
```

Let's increase the magnitude threshold to 50 when detecting edges in `output11885.jpg` and save the image with the detected edges in `output11885_ed2.jpg`. Both images are shown in Fig. 4. As you can see, there are fewer edges detected. I encourage you to experiment with different magnitude threshold values to see the effects of gradient magnitude thresholding.

Save your implementation of `gd_detect_edges(rgb_img, magn_thresh=20)` in `cs3430_s19_hw10.py`.

Problem 2: Image Similarity (1 point)

Let's implement three popular metrics that measure similarity between two binary images. These metrics are popular in various areas of information retrieval. The first measure is cosine similarity. The cosine similarity between two vectors, V_1 and V_2, is computed using the following formula, where $V_1[i]$ and $V_2[i]$ refer to the i-th values in V_1 and V_2, respectively.

$$\texttt{cosine\_sim}(V_1, V_2) = \frac{\sum_{i=1}^{n} V_1[i] \cdot V_2[i]}{\sqrt{\sum_{i=1}^{n} V_1[i]^2} \sqrt{\sum_{i=1}^{n} V_2[i]^2}}.$$

Implement the function `cosine_sim(img1, img2)` that takes two binary images and computes the cosine similarity between them. Note that the cosine similarity ranges from -1 (the two vectors are opposite), 1 (the two vectors are the same), and 0 (the two vectors are orthogonal). You can read more on cosine similarity at `https://en.wikipedia.org/wiki/Cosine_similarity`. The file `cs3430_s19_hw10.py` has the function `test_cosine_sim` that you can use to test your implementation.

```
def test_cosine_sim(img_fp1, img_fp2):
    img1 = Image.open(img_fp1)
    img2 = Image.open(img_fp2)
    sim = cosine_sim(img1, img2)
    del img1
    del img2
    print(img_fp1, img_fp2)
    print(sim)
```

Let's run a few tests.

```
>>> test_cosine_sim('img/2b_nb_09_ed.png', 'img/2b_nb_09_ed.png')
('img/2b_nb_09_ed.png', 'img/2b_nb_09_ed.png')
1.0
>>> test_cosine_sim('img/2b_nb_09_ed.png', 'img/2b_nb_10_ed.png')
('img/2b_nb_09_ed.png', 'img/2b_nb_10_ed.png')
0.512202985103
>>> test_cosine_sim('img/output11884_ed.jpg', 'img/output11885_ed.jpg')
('img/output11884_ed.jpg', 'img/output11885_ed.jpg')
0.352152693884
>>> test_cosine_sim('img/output11885_ed.jpg', 'img/output11884_ed.jpg')
('img/output11885_ed.jpg', 'img/output11884_ed.jpg')
0.352152693884
```

Another commonly used similarity metric is the euclidean distance between two n-dimensional vectors V and U, which is computed as follows.

$$\texttt{euclid\_dist}(U, V) = \sqrt{(u_1 - v_1)^2 + (u_2 - v_2)^3 + ... + (u_n - v_n)^2}.$$

Implement the function `euclid_sim(img1, img2)` that takes two binary images and computes the euclidean distance between them. The file `cs3430_s19_hw10.py` has the function `test_euclid_sim` that you can use to test your implementation.

```
def test_euclid_sim(img_fp1, img_fp2):
    img1 = Image.open(img_fp1)
    img2 = Image.open(img_fp2)
    sim = euclid_sim(img1, img2)
    del img1
    del img2
    print(img_fp1, img_fp2)
    print(sim)
```

Let's run a few tests.

```
>>> test_euclid_sim('img/2b_nb_10_ed.png', 'img/2b_nb_10_ed.png')
('img/2b_nb_10_ed.png', 'img/2b_nb_10_ed.png')
0.0
>>> test_euclid_sim('img/2b_nb_09_ed.png', 'img/2b_nb_10_ed.png')
('img/2b_nb_09_ed.png', 'img/2b_nb_10_ed.png')
16981.9278941
>>> test_euclid_sim('img/2b_nb_10_ed.png', 'img/2b_nb_09_ed.png')
('img/2b_nb_10_ed.png', 'img/2b_nb_09_ed.png')
16981.9278941
```

Finally, the Jaccard similarity is yet another popular similarity measure. Let S_1 and S_2 be two sets. Then the Jaccard similarity between two sets is computed as the ratio of the number of the elements in the intersection of S_1 and S_2 and the number of the elements in the union of S_1 and S_2.

$$\texttt{jaccard\_sim}(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}.$$

Implement the function `jaccard_sim(img1, img2)` that takes two binary images and computes the Jaccard similarity between them. The file `cs3430_s19_hw10.py` has the function `test_jaccard_sim` that you can use to test your implementation. You can use the Python `set()` data structure to convert a binary image into a set of pixel values. The Python class `set` has the member functions `set.union()` and `set.intersection()` that you can also use in implementing this function. Read more on the Jaccard similarity at `https://en.wikipedia.org/wiki/Jaccard_index`.

Let's run a few tests.

```
>>> test_jaccard_sim('img/2b_nb_10_ed.png', 'img/2b_nb_10_ed.png')
('img/2b_nb_10_ed.png', 'img/2b_nb_10_ed.png')
1.0
>>> test_jaccard_sim('img/2b_nb_09_ed.png', 'img/2b_nb_10_ed.png')
('img/2b_nb_09_ed.png', 'img/2b_nb_10_ed.png')
1.0
>>> test_jaccard_sim('img/2b_nb_10_ed.png', 'img/2b_nb_09_ed.png')
('img/2b_nb_10_ed.png', 'img/2b_nb_09_ed.png')
1.0
>>> test_jaccard_sim('img/output11885_ed.jpg', 'img/output11884_ed.jpg')
('img/output11885_ed.jpg', 'img/output11884_ed.jpg')
0.934065934066
>>> test_jaccard_sim('img/output11884_ed.jpg', 'img/output11885_ed.jpg')
('img/output11884_ed.jpg', 'img/output11885_ed.jpg')
0.934065934066
```

Save your code in `cs3430_s19_hw10.py`.

What to Submit

You have to submit `cs3430_s19_hw10.py`. You should also submit all the files needed to run your code. Zip your entire working directory in `hw10.zip` and submit it via Canvas.

And here comes my perpetual refrain! Do not change the names of the files that were given to you or the names of the functions you are asked to implement. The unit tests that I write for the grader every week depend on these names remaining the same.

Happy Hacking!