Edge Detection in Python

Ritvik Kharkar Jan 23, 2020 · 5 min read ★

Last quarter, I was helping teach a Python course at my university, and learned a lot about image processing as a result. I wanted to continue sharing that knowledge in this article as we talk about the **theory and execution** of edge detection using Python!

something cool, why is it a useful technique? To motivate that point, consider the following image of a pinwheel and its "edges-only" counterpart:

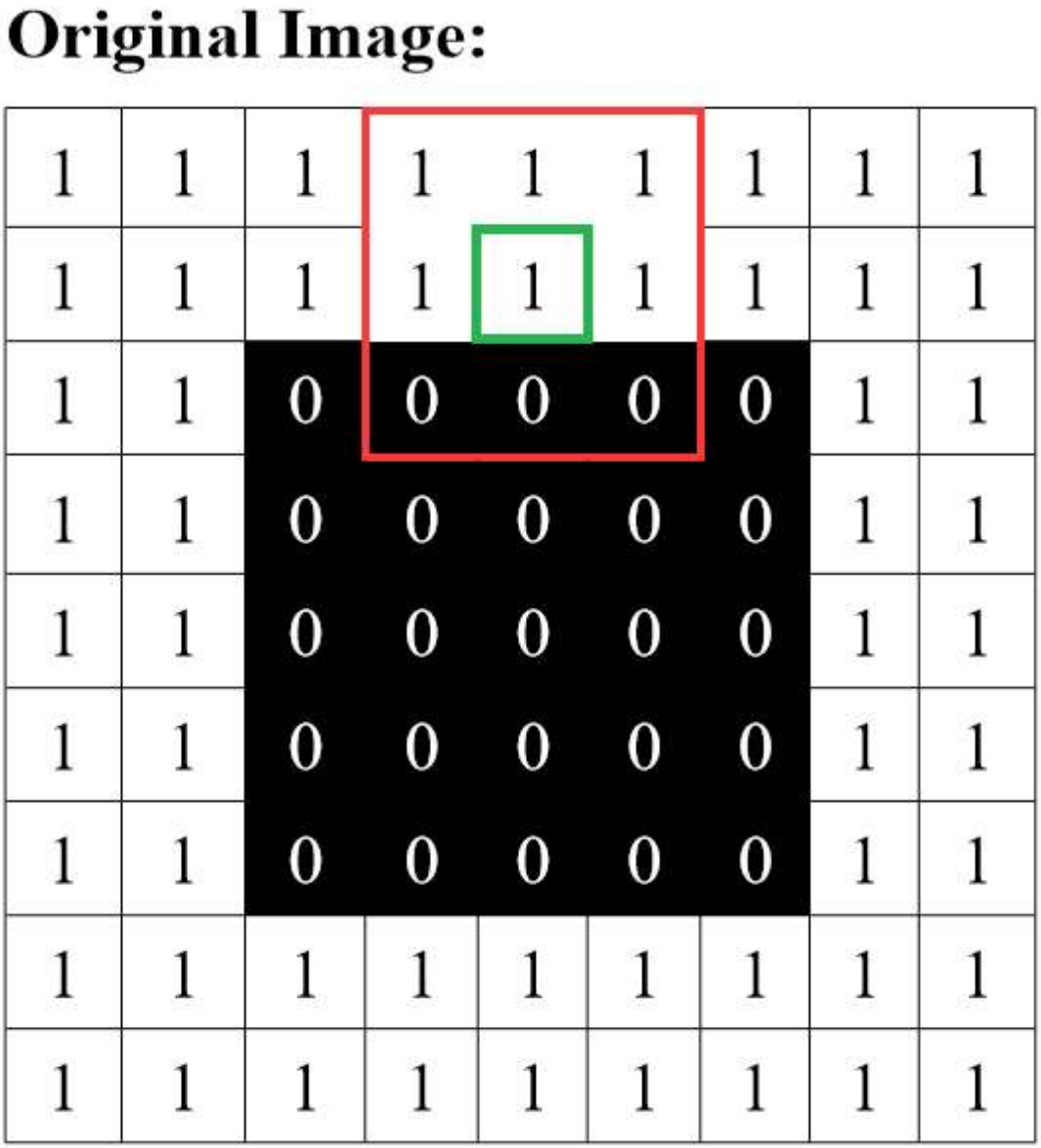
Image of pinwheel (left) and its edges (right)

We can see that the original image on the left has various colors and shades, while the "edges-only" representation on the right is black and white. If asked which image requires more data storage, I bet you would say the original image. And this makes sense; by detecting the edges of an image, we are doing away with much of the detail, thereby making the image "more lightweight".

Thus, edge detection can be incredibly useful in cases where we don't need to maintain all the intricate details of an image, but rather ***only care about the overall shape***.

How to Perform Edge Detection — The Math

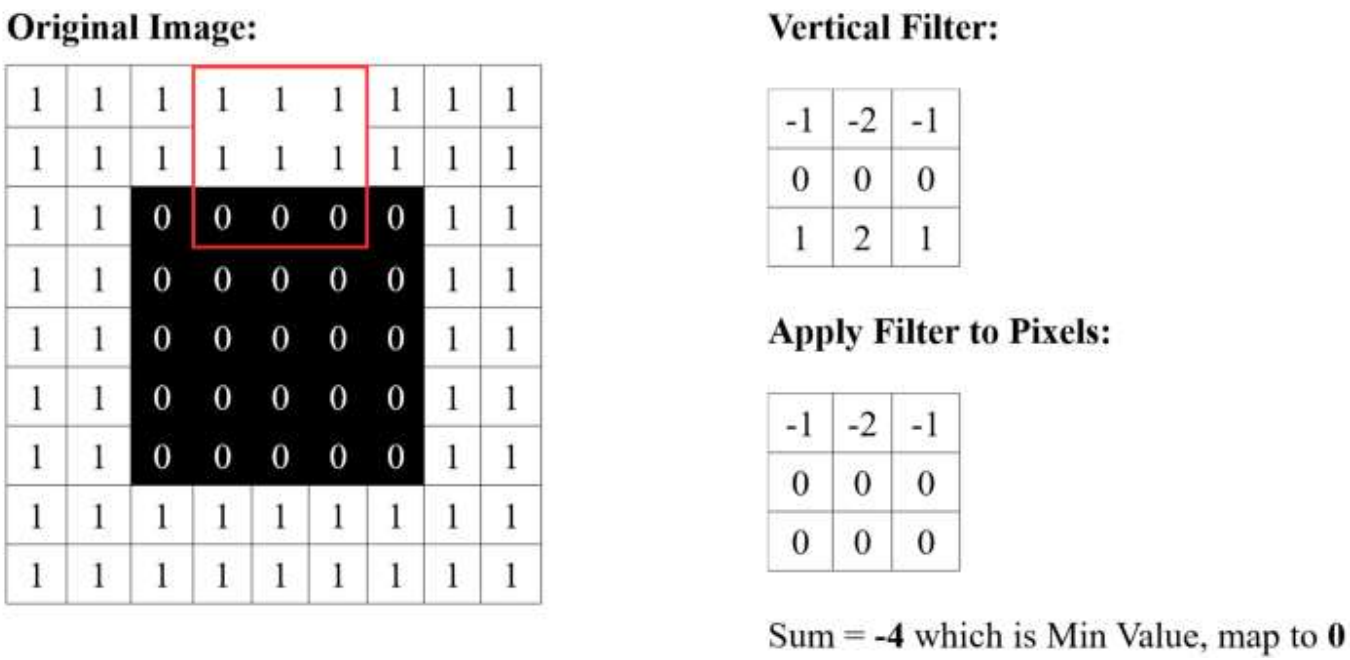
Before talking about the code, let's take a quick look at the math behind edge detection. We, as humans, are pretty good at identifying the "edges" of an image, but how do we teach a computer to do the same thing?



Our working image

In this example, we consider each pixel to have a value between **0 (black)** and **1 (white)**, thus dealing with only black and white images for right now. The exact same theory will apply to color images.

Now, let us say we are trying to determine whether or not the green highlighted pixel is part of the edge of this image. As humans, we would say **yes**, but how can we use neighboring pixels to help the computer reach the same conclusion?

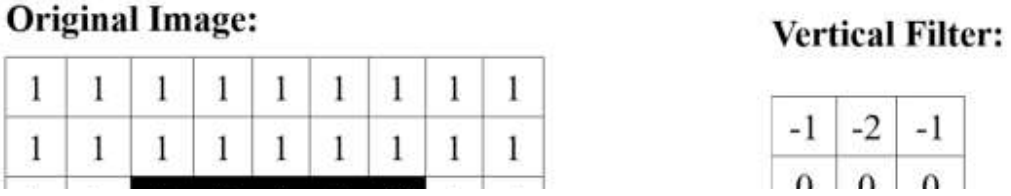


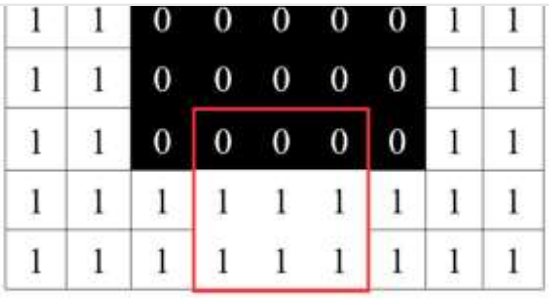
Apply the vertical filter to the local box of pixels

The filter we will "apply" is shown above, and looks rather mysterious at first glance, but let us see how it behaves. Now, when we say ***"apply the filter to the little local box of pixels"*** we mean multiply each pixel in the red local box by each pixel in the filter element-wise. So, the top left pixel in the red box is 1 whereas the top left pixel in the filter is -1, so multiplying these gives -1, which is what we see in top left pixel of the result. Each pixel in the result is achieved in exactly the same way.

The next step is to sum up the pixels in the result, giving us -4. Note that -4 is actually the *smallest* value we can get by applying this filter (since the pixels in the original image can be only be between 0 and 1). Thus, we know the pixel in question is part of a **top vertical edge** because we achieve the minimum value of -4.

To get the hang of this transformation, let's see what happens if we apply the filter on a pixel at the bottom of the square:





Apply Filter to Pixels:

0	0	0
0	0	0
1	2	1

Sum = **4** which is Max Value, map to **1**

We see that we get a similar result, except that the sum of the values in the result is 4, which is the *highest* value we can get by applying this filter. Thus, we know we found a pixel in a **bottom vertical edge** of our image because we got the highest value of 4.

To map these values back to the 0–1 range, we simply add 4 and then divide by 8, mapping the -4 to a 0 (**black**) and mapping the 4 to a 1 (**white**). Thus, using this filter, called the *vertical Sobel filter*, we are able to very simply detect the vertical edges in our image.

What about the horizontal edges? We simply take the **transpose of the vertical filter** (flip it about its diagonal), and apply this new filter to the image to detect the horizontal edges.

Now, if we want to detect horizontal edges, vertical edges, and edges that fall somewhere in between, we can **combine the vertical and horizontal scores**, as shown in the following code.

Hopefully the theory is clear! Now let's finish up by looking at the code.

How to Perform Edge Detection — The Code

First some setup:

```
1  %matplotlib inline
2
```

```
#define the vertical filter
vertical_filter = [[-1,-2,-1], [0,0,0], [1,2,1]]

#define the horizontal filter
horizontal_filter = [[-1,0,1], [-2,0,2], [-1,0,1]]

#read in the pinwheel image
img = plt.imread('pinwheel.jpg')

#get the dimensions of the image
n,m,d = img.shape

#initialize the edges image
edges_img = img.copy()
```

edge_detection_prep.py hosted with ♡ by GitHub view raw

- Replace “pinwheel.jpg” with whatever fun image you want to find the edges of! Make sure it’s in the same working directory.

And the edge detection code itself:

```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt

#define the vertical filter
vertical_filter = [[-1,-2,-1], [0,0,0], [1,2,1]]

#define the horizontal filter
horizontal_filter = [[-1,0,1], [-2,0,2], [-1,0,1]]

#read in the pinwheel image
img = plt.imread('pinwheel.jpg')

#get the dimensions of the image
n,m,d = img.shape

#initialize the edges image
edges_img = img.copy()
```

```
    for row in range(3, n-2):
        for col in range(3, m-2):

            #create little local 3x3 box
            local_pixels = img[row-1:row+2, col-1:col+2, 0]

            #apply the vertical filter
            vertical_transformed_pixels = vertical_filter*local_pixels
            #remap the vertical score
            vertical_score = vertical_transformed_pixels.sum()/4

            #apply the horizontal filter
            horizontal_transformed_pixels = horizontal_filter*local_pixels
            #remap the horizontal score
            horizontal_score = horizontal_transformed_pixels.sum()/4

            #combine the horizontal and vertical scores into a total edge score
            edge_score = (vertical_score**2 + horizontal_score**2)**.5

            #insert this edge score into the edges image
            edges_img[row, col] = [edge_score]*3

    #remap the values in the 0-1 range in case they went out of bounds
    edges_img = edges_img/edges_img.max()
```

edge_detection.py hosted with ♡ by GitHub view raw

A few things to note:

- There will be a small border around the image since we are unable to fully create the local 3 x 3 box on the border pixels.
- Since we are doing detection on both horizontal and vertical edges, we just divide the raw scores by 4 (rather than adding 4 and then dividing by 8). It is not a major change but one which will better highlight the edges of our image.
- Combining the horizontal and vertical scores might cause the final edge score to go out of the 0–1 range, so we finish by re-normalizing the scores.

Running the above code on a more complicated image:

results in the edge detection:

And that's all! Hope you learned something and stay tuned for more data science articles ~