caMicroscope
Google Summer Of Code 2020

Author: Alina Boshchenko

1 Code Challenge

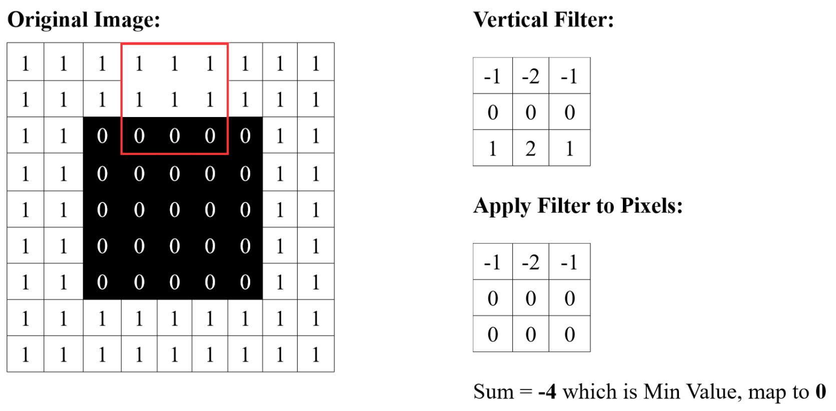
1.1 Task statement and introduction:

Create a page or tool which performs edge detection on a given image and, given a point, returns the distance from that point to the closest edge.

Source Code: https://github.com/caMicroscope/caMicroscope
Solution Code: https://github.com/caMicroscope/caMicroscope

1.2 Classical Edge Detection

The first approach appears to be a mathematically intuitive solution.
For the base part, we consider each pixel on image to have a value between 0 (black) and 1 (white). The same theory will be applied to color images. To determine if the highlighted pixel is part of the edge we take a small 3 x 3 box of local pixels centered at this pixel. After that we apply vertical filters to this box by multiplying each pixel in the red local box by each pixel in the filter element-wise. By calculating minimum



Original Image:

1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	0	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1

Vertical Filter:

-1	-2	-1
0	0	0
1	2	1

Apply Filter to Pixels:

-1	-2	-1
0	0	0
0	0	0

Sum = **-4** which is Min Value, map to **0**

Figure 1:

and maximum possible values of the sum we can determine if the pixel in question is part of a top vertical or bottom vertical edge respectively. However we still need to map values back to the $0 - 1$ range, so we add 4 and then divide by 8, normalizing -4 to 0 (black) and 4 to a 1 (white).
To determine horizontal edges we need to use a horizontal filters which are received from the transposed vertical ones. We apply and normalize them in the same way and find top or bottom horizontal edges by comparing received sums with minimum and maximum possible values .

To detect edges that fall somewhere in between as well, the vertical and horizontal scores are combined.

Results: Described algorithm produced decent results, edges were detected and there was almost no noise presenting on the output image.

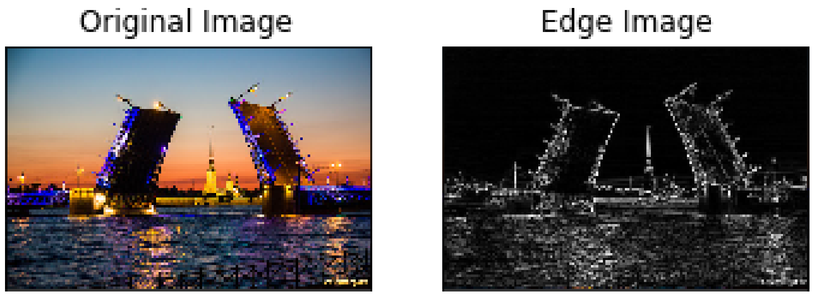


Figure 2: Edge detection using classical approach

1.3 Canny Edge Detection

After analysing the first approach results I decided to move forward and find a way to improve results. For this goal I decided to experiment with Canny Edge Detection in OpenCV. In this approach mage is filtered with a Sobel kernel (like described above) in both horizontal and vertical direction to get first derivatives. From these two images, we can find edge gradient and direction for each pixel. Gradient direction is always perpendicular to edges. It is rounded to one of four angles representing vertical, horizontal and two diagonal directions. After that it is vital to remove any unwanted pixels which may not constitute the edge. For this, every pixel is checked if it is a local maximum in its neighborhood in the direction of gradient. The most important step of the algorithm is Hysteresis Thresholding. Any edges with intensity gradient more than maxVal are sure to be edges and those below minVal are sure to be non-edges, so discarded. Those who lie between these two thresholds are classified based on their connectivity. If they are connected to "sure-edge" pixels, they are considered to be part of edges. Otherwise, they are also discarded.

Results: After performing the algorithm it appears that there is a lot of noise present. To deal with it I decided to apply Gaussian Blur to the image before the main algorithm stage. It enhanced the result significantly and after blur parameters adjustment produced the most accurate result.

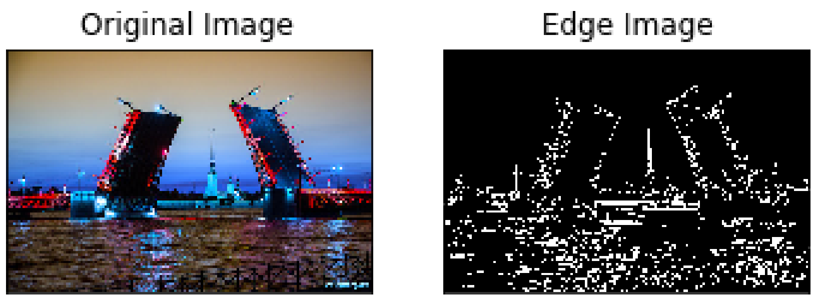


Figure 3: Canny Edge detection without blur

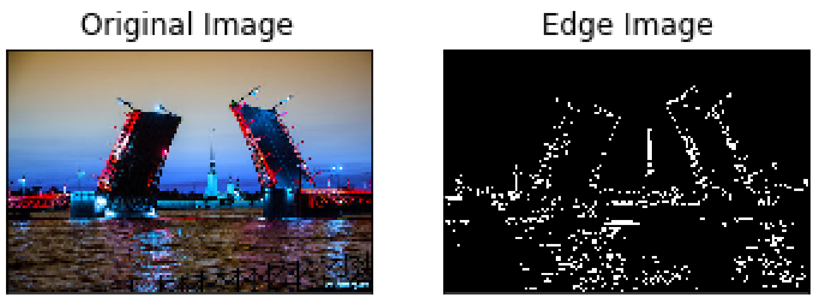


Figure 4: Canny Edge detection with blur

1.4 Next step

For the next stage it is required to create an approach to determine the distance to the closest edge.

Credits for image (Figure 1): https://towardsdatascience.com/