Image processing miniproject

Horizontal and Vertical Edge Detection

Daniel Kartin

Project repository

Teacher
Tsampikos Kounalakis
tkoun@create.aau.dk
AAU CPH - MED3
Image processing

December 4, 2017

Contents

1
Algorithm

Starting out, I did some research into horizontal and vertical edge detection by going back and looking through the old slides on the subject, reading in the book, reading the opencv documentation, to understand how their version worked, and doing general knowledge search on the Internet.

An edge is defined as a sudden jump in intensity in an image. Imagine the simple graph in Figure 1.1, it illustrates the hypothetical image with an intensity jump, in the algorithm this should be seen as an edge.

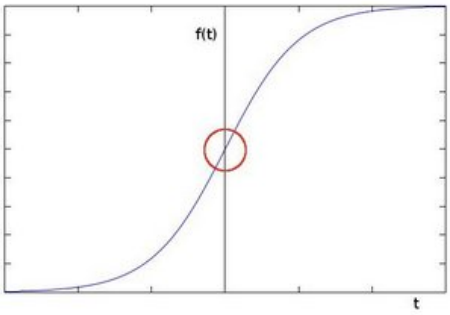

Figure 1.1: Simple representation of a hypothetical intensity jump in an image. Taken from the opencv documentation on the sobel function.

Doing simple edge detection involves making two 3x3 kernels; one for horizontal and one for the vertical axis. As the assignment called for a gray scale image, the image will have to be converted to gray scale, or a gray image have to be used. The kernels that worked the best for me, looked like this:

$$xKernel = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \qquad yKernel = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

In theory a larger kernel could be used, as long as it is an odd size. If a larger kernel is used though, the resulting edges will be thicker. These two kernels are then individually convolved with the image, this calculates the approximate derivative. The resulting two images, will be called xGradient and yGradient. These images are now already showing respectively horizontal and vertical edges, as this takes the gradient of the images on each axis.

At each point of the image, we then have to calculate the approximate of the gradient at that point. This is done by essentially combining the results of the two earlier convolutions. This is done by applying the following formula:

$$finalGradient = \sqrt{xGradient^2 + yGradient^2}$$

to every pixel in the image. The result is the gradient of the final image, combining both the horizontal and vertical edges from the prior calculations.

So the resulting algorithm in steps will be:

1. Load image
2. Make kernels
3. Convolve image with x kernel
4. Convolve image with y kernel
5. Take the resulting two gradients and take the approximate gradient of them combined into one final image
6. Threshold to make the image binary, and not lose any detail
7. (Optional) Have a Tuborg to celebrate

2
The code

The code in it's entirety can be found in the appendix 5 and 6, or in the Project repo. This chapter will focus on the important bits, and not showcase all the code as that wouldn't be a proper use of anyone's time.

Starting out, we make the two kernels, these kernel values are based on the official numbers from the wikipedia article about edge detection, the official opencv documentation, plus trial and error.

```
1 //X sobel kernel
2 int kernelX[3][3] = {1, 0, -1, 2, 0, -2, 1, 0, -1};
3
4 //Y sobel kernel
5 int kernelY[3][3] = {1, 2, 1, 0, 0, 0, -1, -2, -1};
```

Listing 1: Horizontal and vertical kernels

After that we take the gray scale version of the original image and clone it into all the materials that will be used later on.

```
1 int radius = 1;
2
3 //Saving the initial image, to be overwritten by the for loops
4 gradX = src.clone();
5 gradY = src.clone();
6 gradF = src.clone();
```

Listing 2: Cloning the gray scale image into all the to be used materials

Now we have to convolve the image in the horizontal and vertical direction, we start off with the x kernel. Convolving the image requires a double for loop, for both the x and the y positions. Starting at the very beginning of the image, it will for every pixel, add unto an integer the result from applying the kernel to the pixel and it's neighbors. The exact same thing is done for the y part of the image, except it is saved in gradY instead, and using the yKernel instead of the xKernel.

```
for (int row = radius; row < src.rows - radius; row++) {
        for (int col = radius; col < src.cols - radius; col++) {
                int scale = 0;
                for (int i = -radius; i <= radius; i++) {
                        for (int j = -radius; j <= radius; j++) {
                                scale += src.at<uchar>(row + i, col + j) * kernelX[i + radius][j + radius];
                        }
                }
                gradX.at<uchar>(row - radius, col - radius) = scale / 480s;
        }
}
```

Listing 3: Looping over the the image with the x kernel to get approximate gradient on the x axis of the image.

The last part is taking the two resulting gradients from each axis, and applying the formula $gradF = \sqrt{xGrad^2 + yGrad^2}$ on every pixel from both images, essentially combining them. Then we do an extremely simple threshold to emphasize any remaining edges, but unfortunately also making the border of the image, completely white.

```
for (int row = 0; row < gradF.rows; row++) {
        for (int col = 0; col < gradF.cols; col++) {

                gradF.at<uchar>(row, col) = static_cast<uchar>(sqrt(pow(gradX.at<uchar>(row, col), 2) +
                pow(gradY.at<uchar>(row, col), 2)));
                //Simple threshold
                if (gradF.at<uchar>(row, col) > 1) {
                        gradF.at<uchar>(row, col) = 255;
                } else {
                        gradF.at<uchar>(row, col) = 0;
                }
        }
}
```

Listing 4: Calculating an approximation of the gradient at every point, using both the x and y resulting images

3 The program

When the program runs, it takes the image, and performs the image processing on it. Along the way it makes windows showing the steps visually. First step is just the input image

Figure 3.1: The input image

We then use the built-in opencv function to convert the image to gray scale, as this isn't strictly relevant to horizontal and vertical edge detection.

Figure 3.2: Gray scale image

This is then the image after the vertical edges have been detected.

Figure 3.3: Vertical edges detected

Then there's the horizontal edges.

Figure 3.4: Horizontal edges detected

And then there's the complete final result, with the combined images.

Figure 3.5: The resulting image, combining both from both the other images.

Now the two very important parts of the program is the kernels, and the amount you divide the scale by, when taking the gradients of the image. Starting off with the scale number, I am fairly certain this is a form of simple normalization. Doubling the number from 480 to 960, makes the function much more aggressive, as seen in Figure 3.6.

Figure 3.6: The final image, with the doubled scale divider number

Now when changing the kernel, the results change drastically. The kernel used in Figure 3.7 looks like this

$$xKernel = \begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix}, \qquad yKernel = \begin{bmatrix} -3 & -10 & -3 \\ 0 & 0 & 0 \\ 3 & 10 & 3 \end{bmatrix}$$

With the new kernel, the image is much more "detailed", and the whole mountain range line is kept intact, but much "noise" inside the mountain is also introduced.

Figure 3.7: The final image, with another kernel

Appendices

```
#include <iostream>
#include <cmath>
#include <opencv2/opencv.hpp>

using namespace cv;

Mat src, gradF, gradX, gradY;

void doImageProcessing() {

        imshow("original image", src);


        //Standard Sobel kernel
        //int kernelX[3][3] = {1, 0, -1, 2, 0, -2, 1, 0, -1};

        //Sobel-Feldman kernel
        int kernelX[3][3] = {-3, 0, 3, -10, 0, 10, -3, 0, 3};

        //Standard Sobel kernel
        //int kernelY[3][3] = {1, 2, 1, 0, 0, 0, -1, -2, -1};

        //Sobel-Feldman kernel
        int kernelY[3][3] = {-3, -10, -3, 0, 0, 0, 3, 10, 3};


        int radius = 1;

        //Saving the initial image, to be overwritten by the for loops
        gradX = src.clone();
        gradY = src.clone();
        gradF = src.clone();

        //Looping over the the image with the x kernel
        //From this we get the gradient image
        for (int row = radius; row < src.rows - radius; row++) {
                for (int col = radius; col < src.cols - radius; col++) {
                        int scale = 0;
                        for (int i = -radius; i <= radius; i++) {
                                for (int j = -radius; j <= radius; j++) {
                                        scale += src.at<uchar>(row + i, col + j) * kernelX[i + radius]
                                        [j + radius];
                                }
                        }
                        gradX.at<uchar>(row - radius, col - radius) = scale / 480;
                }
        }
```

Listing 5: Horizontal and vertical kernels

```
    imshow("X edge detection", gradX);

//Looping over the image with the y kernel
//From this we get the gradient image
for (int row = radius; row < src.rows - radius; row++) {
        for (int col = radius; col < src.cols - radius; col++) {
                int scale = 0;

                for (int i = -radius; i <= radius; i++) {
                        for (int j = -radius; j <= radius; j++) {
                                scale += src.at<uchar>(row + i, col + j)* kernelY[i + radius]
                                [j + radius];
                        }
                }
                gradY.at<uchar>(row - radius, col - radius) = scale / 480;
        }
}

imshow("Y edge detection", gradY);

//Here we calculate an approximation of the gradient at every point, using both the x and y images
for (int row = 0; row < gradF.rows; row++) {
        for (int col = 0; col < gradF.cols; col++) {

                gradF.at<uchar>(row, col) = static_cast<uchar>(sqrt(pow(gradX.at<uchar>(row, col), 2) +
                pow(gradY.at<uchar>(row, col), 2)));
                //Simple threshold
                if (gradF.at<uchar>(row, col) > 1) {
                        gradF.at<uchar>(row, col) = 255;
                } else {
                        gradF.at<uchar>(row, col) = 0;
                }
        }
}

imshow("Edges", gradF);

waitKey(0);
}

int main() {

src= imread("/home/daniel/Documents/opencvFilters/horizont.jpg", CV_LOAD_IMAGE_GRAYSCALE);

if (src.empty()) return -1;

doImageProcessing();
return 0;
}
```

Listing 6: Horizontal and vertical kernels