Photogrammetric Computer Vision Lab2 Report

Professor Sohn

By SeyedMostafa Ahmadi

October
2021

Canny Edge Detector Implementation

I will describe each step with supporting results.

Smoothing and Derivatives

In the course presentations, we learned that the matrix of derivatives should be convolved with the main matrix. In the previous lectures, we had that convolution has associative property. So, we have:

M2 = D * (G * M1), where

D: DERIVATIVES MATRIX
G: GAUSSIAN FILTER
M1: THE MAIN IMAGE
*** : CONVOLUTION SIGN**
M2: TH RESULT IMAGE

Then, we use associative property:

M2 = (D * G) * M1

Using this property does not change the results but it can help the program to run so much faster. Because to matrices of D and G are very smaller than M1, and convolution between them is so much faster than the one where one side is the big matrix of M1. In other words, if we use the first equation, we must operate two convolutions in which both of them we should involve the big matrix. But, if we use the first equation, the big matrix is only convolved once!

So, we do these two steps (smoothing and derivatives) together.

The result is pretty simple as you can see for the train image we have:

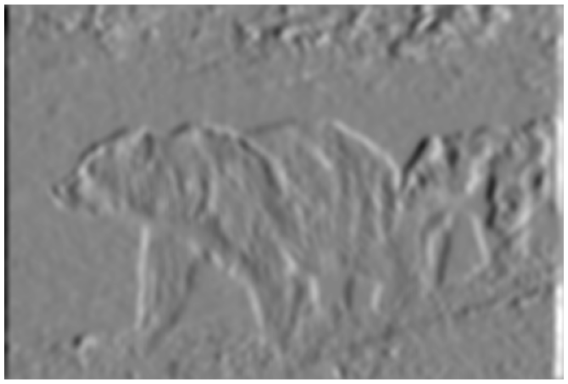

Figure 1. Derivation of training image on y axis

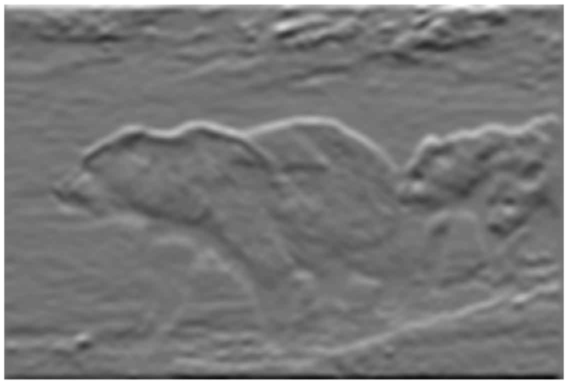

Figure 2. Derivation of training image on x axis

Edge Strength and Direction

For this step I had a pretty simple work to do.

As described in the "*Lab 2 Instructions Overview.pdf*" file, I just followed the instructions.

$$E_s = \sqrt{M2_x^2 + M2_y^2}\text{; and}$$

$$E_d = \arctan(\frac{M2_x}{M2_y})\text{; Where}$$

E_s: Edge strength

E_d: Edge direction

$M2_x$: Derivative of Image M1 in x axis

$M2_y$: Derivative of Image M1 in y axis

The result of this step:

Non-Maximum Suppression

The theory behind this part is pretty simple. Based on the direction in each pixel, we should compare the strength pixel with its neighbourhoods. Defining a pixel's neighbourhoods is as follows:

We define a for loop to go through all pixels of resulted E_s with considering E_d. According to Figure 5, we are checking the pixel with the red circle in it. The grey arrow

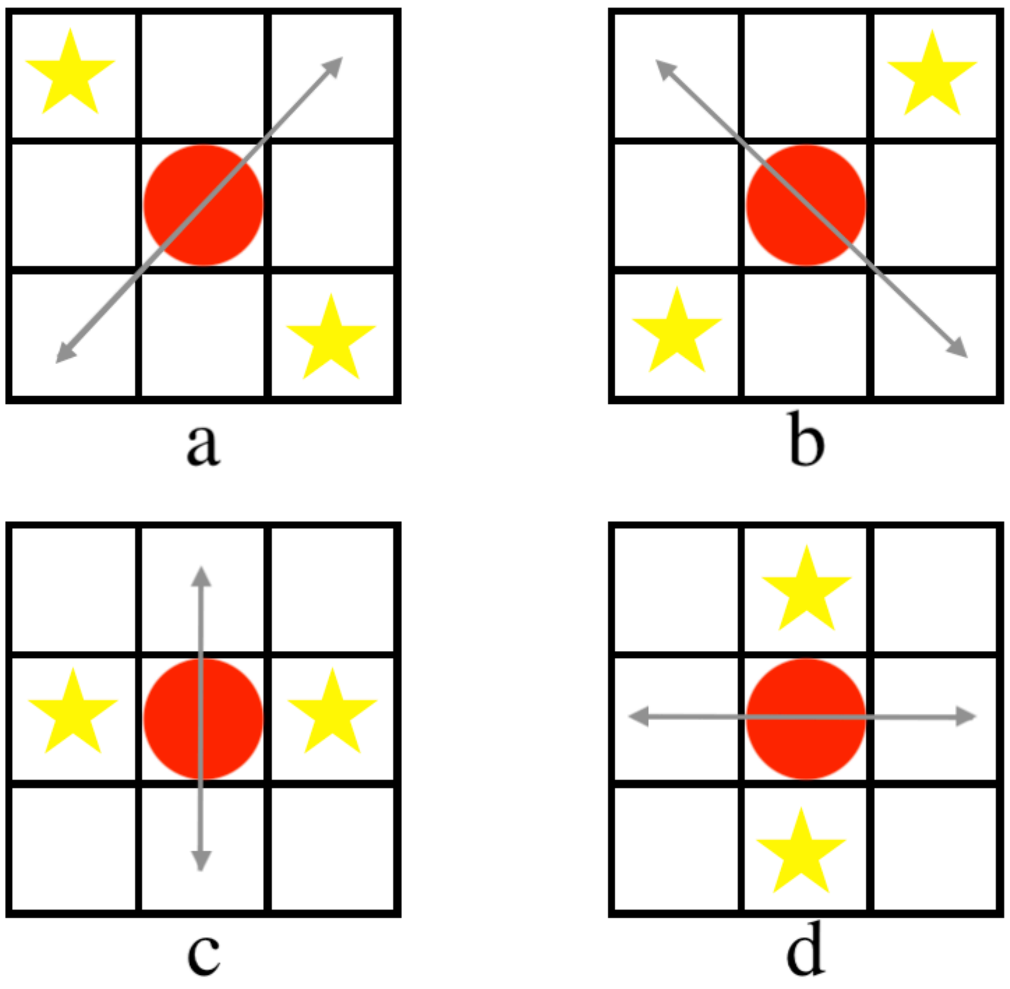

Figure 5. Definition of neighbourhood based on edge direction

shows E_d[red pixel] and the yellow stars represent the neighbours of the red pixel. In each condition (based on E_d[red pixel]) we should compare E_s[red pixel] with its neighbours.

The pseudo-code below shows the operation after we determine which pixels are neighbours of the red pixel:

```
Result[red pixel] = 0
If $E_s$[red pixel] > $E_s$[Yellow Stars]: {
        Result[red pixel] = $E_s$[red pixel]
}
```

In practice, it rarely happens that E_d[red pixel] exactly matches the grey arrows. So, we should define a confident interval that we can map all 360° as a direction in those four patterns of Figure 5.

One simple way is to define the interval equally as 45° for each direction. Also, in the "*Lab 2 Instructions Overview.pdf*" they say we can take this interval 45°. But, I don't think this would be a good idea and we can reach better results with other settings.

For example, take a look at Figure 6. I think this would work better.

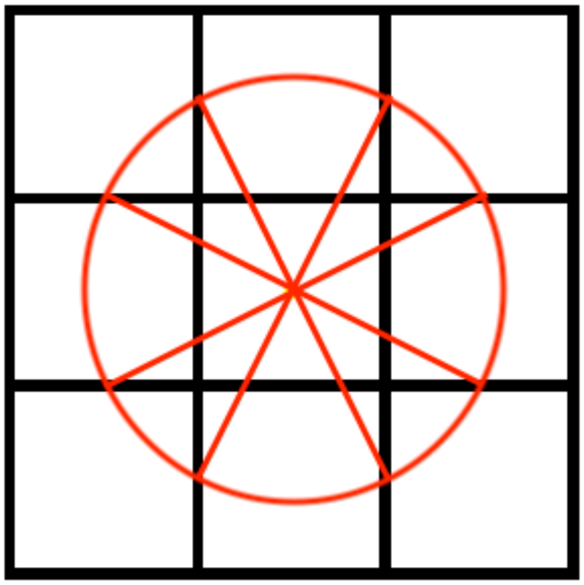

Figure 6. Maybe a better idea for mapping of angle to direction

We should use some high school geometry to know the angles.

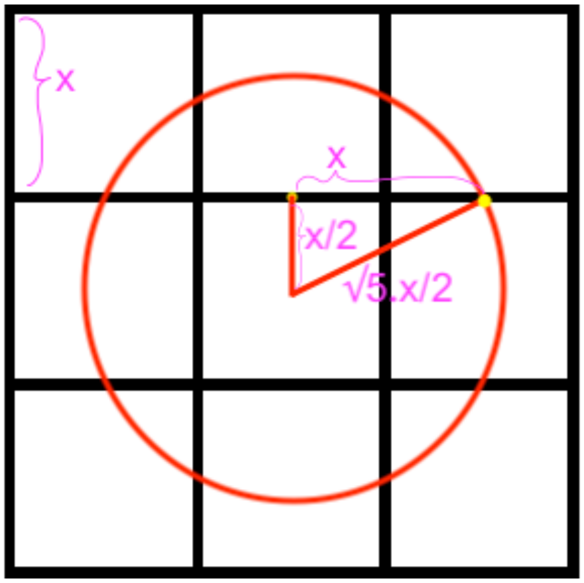

Figure 7. Calculations for finding the angle!

After lots of calculations and using geometry from high school, I finally found that in this setting, horizontal and vertical directions have 53° and diagonal directions have 37°.

After all these calculations, I realized that there maybe even a better setting for these angles. So, what can I do? A good idea would be that we take this as a variable that we are trying to optimize!

We name the angle corresponding to vertical directions as ***alpha*** and the other one ***beta***. It is obvious that: ***alpha*** + ***beta*** = 90°

Then, when we want to find the best solution, we are going to find the best angle for an image.

Here is the difference between alpha = 53° and alpha = 45°:

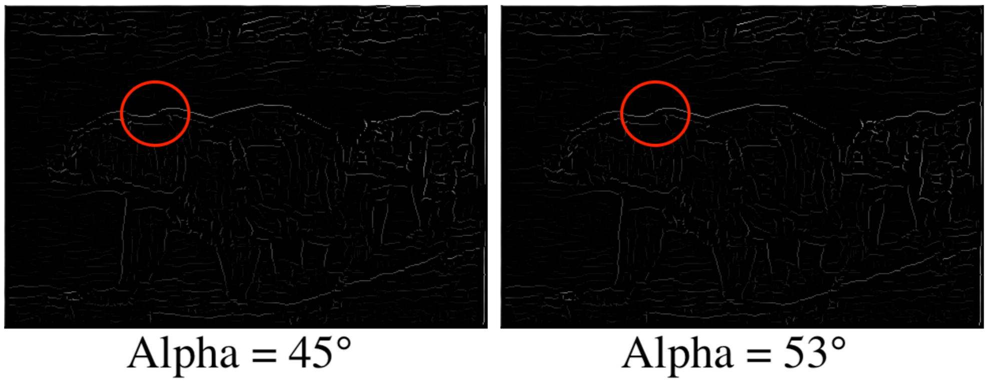

Figure 8. Difference because of alpha value. Alpha = 53° has less discontinuities rather than Alpha = 45°

Thresholding

This step is closely related to the next step which is hysteresis. Here we simply set two thresholds, one for determining strong edges and the other one for determining weak edges. Figure 9 shows what I mean. Then we assign all the strong pixels the value of 1 and all the weak pixels value of 0.5.

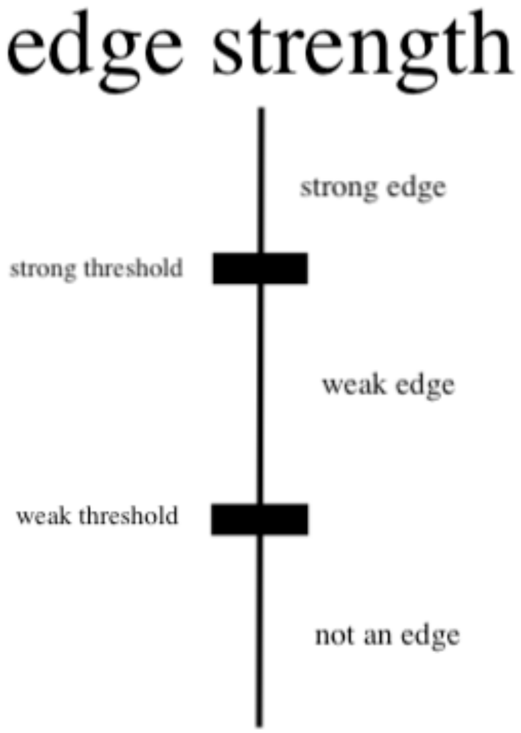

Figure 9. Thresholding

Here is the result of the train image after thresholding where the strong threshold equals 30 and the weak threshold equals 15.

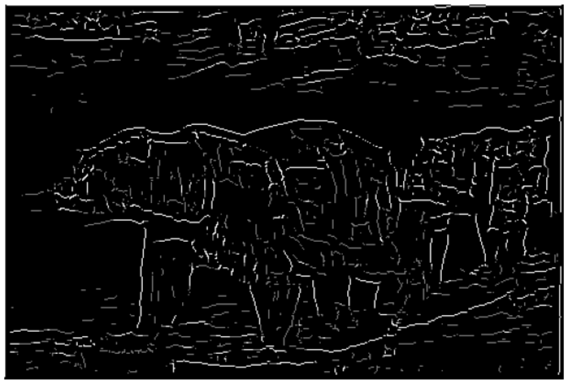

Figure 10. Train image after thresholding

Hysteresis

In this step, we simply remove all the weak edges that are not connected to a strong edge.

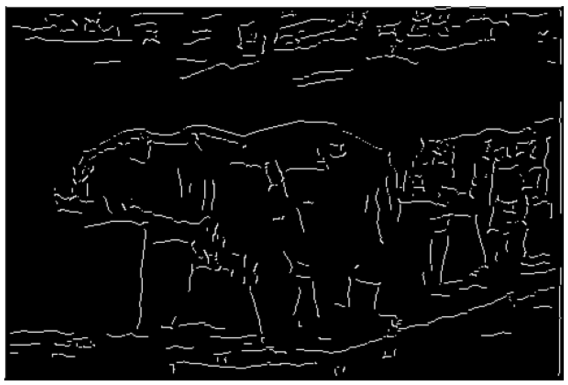

Figure 11. Train image after hysteresis

Now, we technically have finished the canny edge detector algorithm. But, we have some parameters that we should tune.

Parameter Optimization

For this part, at first, I followed the instructions of the "*Lab 2 Instructions Overview.pdf*" file. I mean I draw a plot for the ROC diagram of every reasonable combinations in parameter space. I tried these combination:

```
cut-off = [1,3,5,7,9,11,13,15,17]
Alpha = [45, 53, 60]
```

And in the ROC diagram I was changing the weak threshold in the range of [2 to 40] where the step was 2 (strong was weak*2). You can view all the ROC diagrams for train and test images here: "./data/result/plots/train" and "./data/result/plots/train".

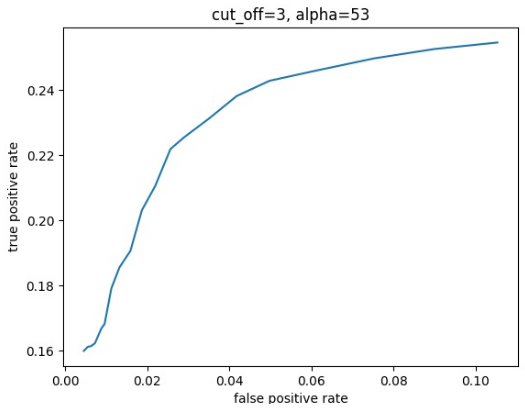

Figure 12. A good sample of ROC diagrams where cutoff=3 and alpha=53

In these diagrams, points near to the top-left of the diagram are the best options for choosing the parameters because accuracy is higher in that points. Look at this equation of accuracy:

$$Accuracy = \frac{(TP + TN)}{(TP + TN + FP + FN)}$$

The denominator is constant and in the top-left of diagram FP is low it means that TN is higher because:
FP+TN = Actual Number of Negatives (constant)
Also in top-left TP is high, Then we can say that (TP+TN) is relatively higher than the other points. So, the accuracy is simply higher (because the denominator in the accuracy equation is constant)!

Genetic Algorithm!

To solve the problem of finding the best tuning for parameters, I used a genetic algorithm. A genetic algorithm is a heuristic search algorithm that I've learned in my AI course during my bachelor's degree. I've always loved the algorithm so this is why I used it to solve this problem. So, the "*Lab 2 Instructions Overview.pdf*" says that we should solve the problem only for two parameters; cut-off frequency and low threshold (weak threshold). But, I want to add 2 more parameters: **alpha** and **high threshold** (strong threshold). In this way, we have an array of parameters: [cutoff, alpha, low, high] that we are searching for the best one in the parameter space.

Algorithm description

To summarize the algorithm in one sentence, suppose that we have a fitness function that the algorithm tries to find the solution that has the best fitness. Now, how it is done? The algorithm has many different implementations and I am going to explain mine.

First off, I have set this function as my fitness function:

Fitness = Accuracy + True Positive Rate - False Positive Rate

Note: During this algorithm, we always use train image (bears).

First, we have a random population with (20 members). It means that we have 20 arrays with different values that the first member of each member represents the value for cutoff, the second one represents alpha, the third represents the low threshold, and the fourth represents the high threshold. Then we run the canny edge detector with each random value and for the resulting image, we calculate fitness for each member. Then we select a mating pool with a size of 10. This selection is random but for each member we have: The better the fitness, the more you are likely to be selected.

Then in this mating pool, members are randomly selected to crossover. Offspring genes (results of crossover, an array of parameter values) are generated from their parents. 10 couples are randomly selected (with replacement) and 10 offspring genes are generated. The first half from the first parent and the other half from the second parent. After each crossover, we should perform mutation. Mutation happens on the offspring arrays and changes a parameter's value in a small amount (between -5 and 5). Then we have the offspring arrays! We take parents and the offsprings as a new population and repeat this algorithm till it converges!

There are some minor points in initializing the first population.
AAfter trying a huge number of parameter arrays, I had an intuition that cutoff values more than 11 are not good. So, the cutoff value should be between [1, 11] and must be an odd number. Also, Alpha cannot be less than 15 and more than 75. Also, the Low parameter is between 1 and 40 and the High parameter is between low and 80.

Long story short! The result image after running so many times is:

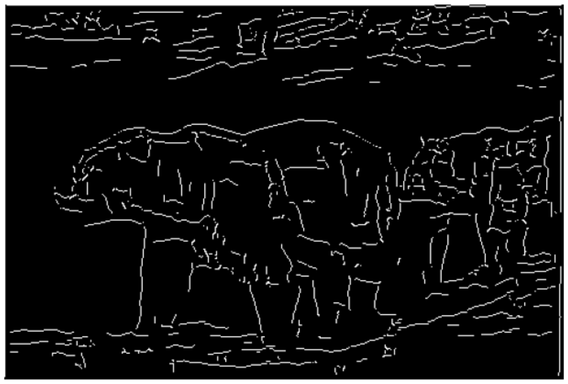

Figure 13. Result of train image

True positive rate = 0.2329
False positive rate = 0.0377
Accuracy = 0.9360

This is resulted with:

cutoff_frequency=3, alpha=53, low=15, high=26

And the test image is: (Accuracy for the test image is even better!)

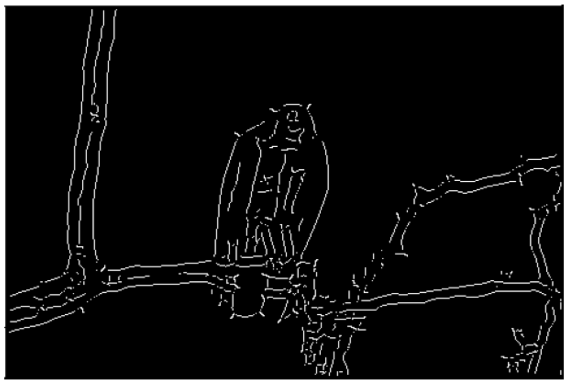

Figure 14. Result of test image

True positive rate = 0.2770
False positive rate = 0.0206
Accuracy = 0.9521

Small comment on the codes:
For viewing the results run “code/canny.py”.
For getting the best parameters from the genetic algorithm (that may differ from the result I got!) run “code/genetic_algorithm.py” and then stop the program after you think it is enough! It usually finds the best after a few loops.

Thank you for your attention!