EE569 – HW2

Mozhdeh Rouhsedaghat

2726554211 rouhseda@usc.edu

Problem 1

Abstract and Motivation:

Edge detection is one of the low-level image processing techniques which helps in understanding the image. It can be used for high-level vision tasks such as semantic segmentation, object detection, etc. There are different methods to do edge detection including Sobel and Canny edge detection algorithm which are differentiation based and Sketch token and Structured edge which are machine learning based approaches. The output of an edge detection algorithm is an image with 0 or 1 at each pixel location which 1 means the corresponding pixel is an edge and 0 means otherwise.

Approach and Procedures:

Sobel Edge detector

Sobel uses these two masks to approximate gradient value at each pixel in x and y direction.

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

Gx

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

Gy

After the convolution with these masks we can compute the gradient value at each pixel by computing sqrt(dx^2+dy^2) at that pixel.

In the next step, we normalize the gradient values, compute the Gradient Magnitude Cumulative Distribution, set a threshold and find the corresponding gradient value and in the output edge map, we set a pixel to one if the corresponding gradient value is greater than the threshold gradient value.

Canny Edge detector

In this approach, after applying gaussian filter to the image (to remove noise), we find the magnitude and orientation of gradient at each pixel to find the edges. This approach has two improvements over Sobel which are explained below:

Non-maximum Suppression (NMS): This method is used to remove extra parts of the edges and make them thinner. In this method we only consider the local maximum of the gradient as edge and set others to non-edge.

Hysteresis Thresholding: In this method we consider two thresholds: low and high.

If the gradient value at any pixel is greater than the high threshold, we consider that pixel as edge and if it is less than the low threshold we consider it as non-edge. If the gradient value is between the low and high thresholds we must check whether this pixel is connected to an edge pixel. If yes, we consider it as edge pixel and if no, we count it as non-edge.

Structured Edge [1]

This method is a machine learning base method for edge detection and can detect edges if it has seen them in the training data.

The output edge map of this method is 16*16 patches in which every pixel can be zero or one (2^256), but actually the degree of freedom is not that high as the output is determined using the correlation between pixels and has C(256, 2) = 32640 dimensions. By using some dimension reduction techniques including random sampling and PCA, this dimension also reduces to 5.

It uses Random Forest learning method to classify input image into different labels and some feature including color space and gradient magnitude are used to train this learning model.

For prediction we can use 4 random trees and average their probability distribution for each label to reach a probability edge map at the output. If we want a binary edge map we can easily set a threshold and binarize the output by comparing probabilities with the threshold (This method is completely explained in part c)

Results:

a)

Fig1: Gray-level versions of Pig ang Tiger images

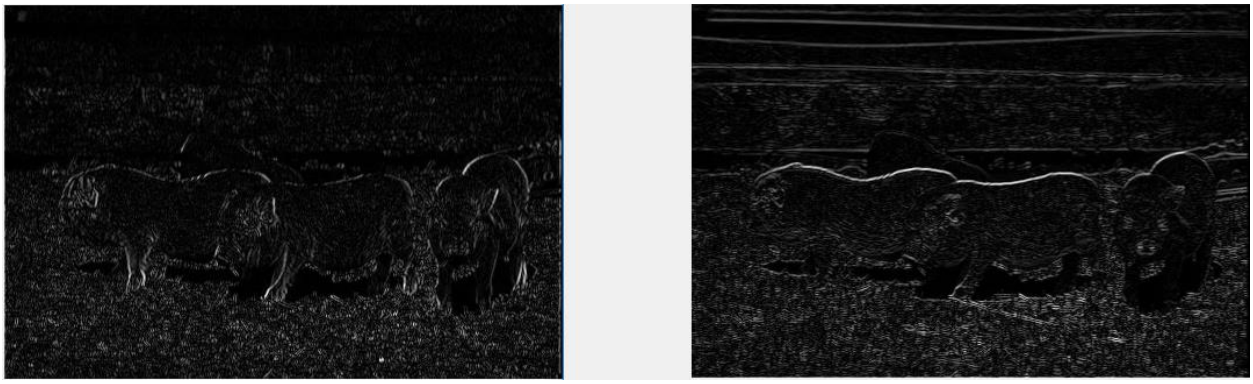


Fig2: Normalized x-gradient (left) and y-gradient (right) of Pig image

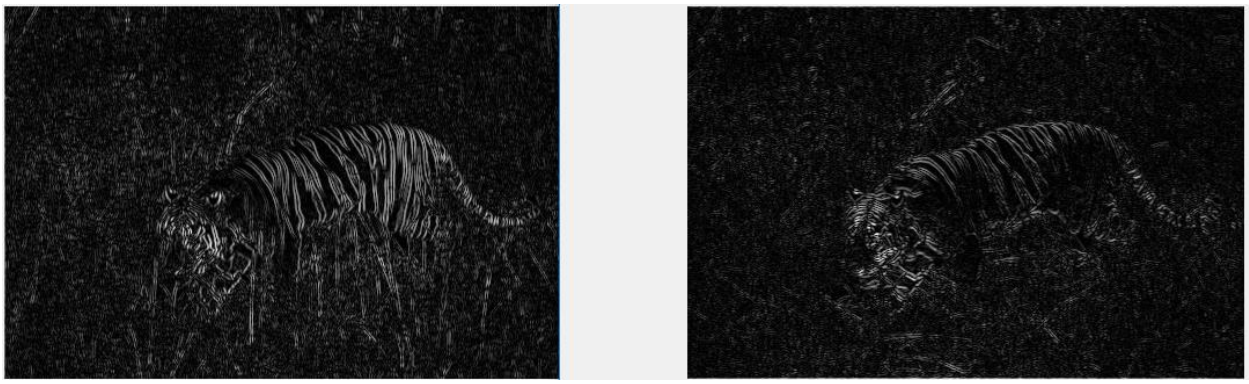


Fig3: Normalized x-gradient (left) and y-gradient (right) of Tiger image

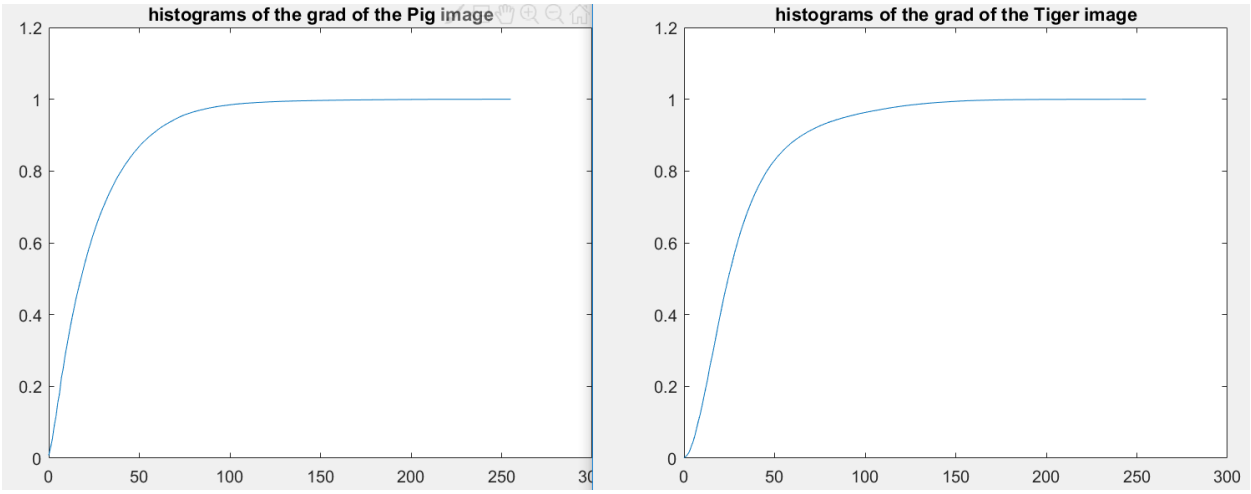


Fig4: Normalized gradient magnitude cumulative histogram of Pig and Tiger images

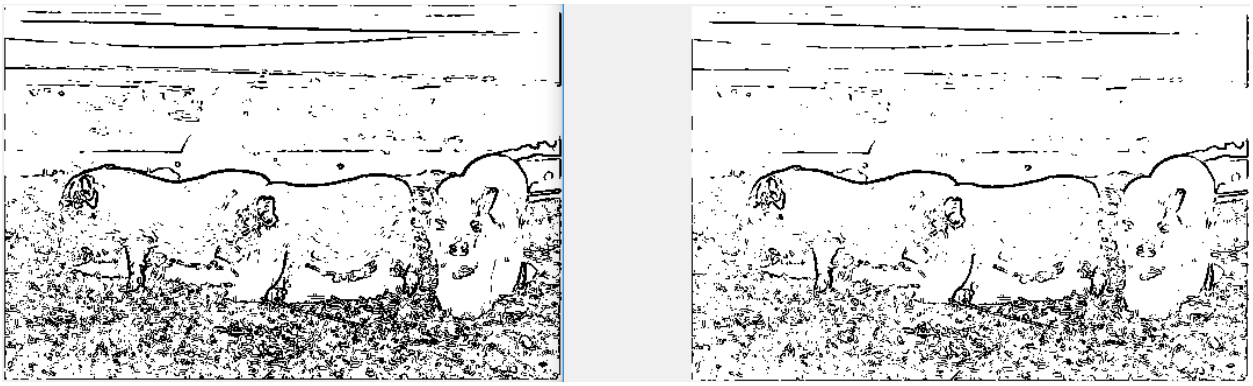


Fig5: Edge map of the Pig image with threshold 85%(left) and 90%(right)

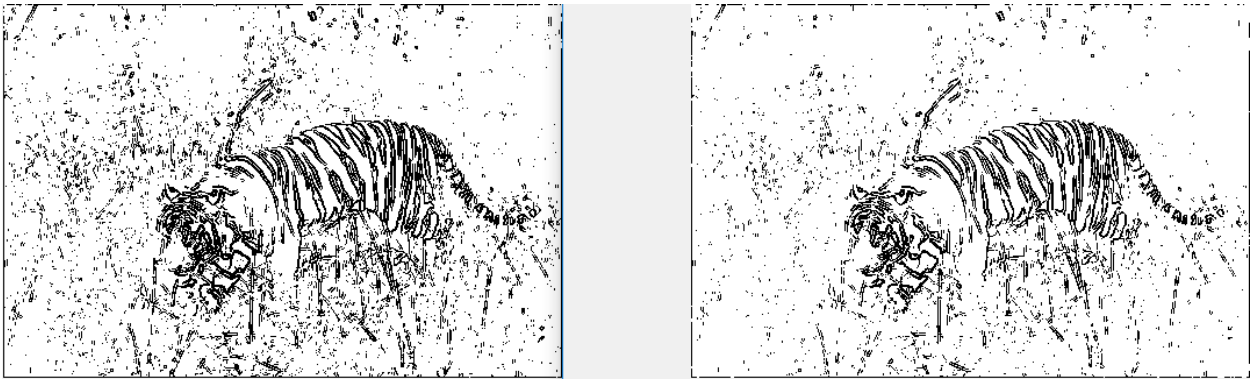


Fig 6: Edge map of the Tiger image with threshold 85%(left) and 90%(right)

To obtain the gray-level images I calculated the average of r, g and b values of each pixel as the gray scale value (Fig1). Then applied the Sobel masks to find x-gradient and the y-gradient values. Then calculated the overall gradient value at each pixel by using sqrt(dx^2+dy^2) and normalized it. Then set 85% and 90% thresholds to remove weaker gradients and set stronger ones as edge in the output edge map.

As you see in the edge map of the Pig image, by changing the threshold from 85% to 90% the ground scene become less visible which is pleasant but if we try to increase the threshold further, the main part of the body of pigs vanishes and they can't be detected easily, so 90% is a good threshold for Pig image.

About the tiger image, by changing the threshold from 85% to 90%, more grass vanishes, and the tiger becomes clearer, but if we increase the threshold more, we will lose some main part of the body of the tiger, so 90% is a good threshold for Tiger image.

b)

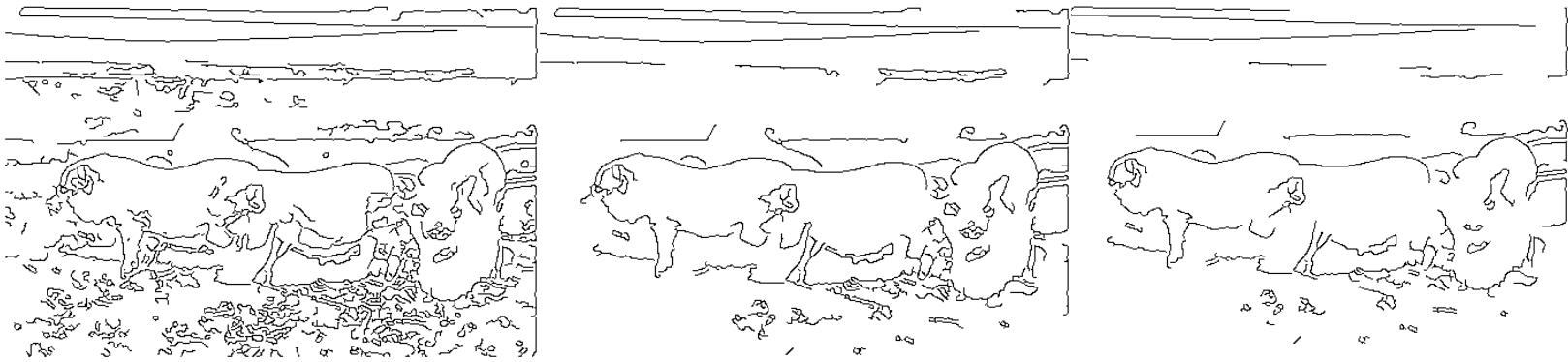


Fig7: Output of Canny edge detector for Pig image (left: high_th=.2 low_thr=.1 middle: high_th=.34 low_thr=.11 right: high_th=.35 low_thr=.15)

Fig8: Output of Canny edge detector for Tiger image (left: high_th=.3 low_thr=.1 middle: high_th=.31 low_thr=.175 right: high_th=.4 low_thr=.25)

By increasing the thresholds weaker gradients will be considered as non-edge and will be removed from the edge map. But if we increase it more than needed, main parts of the edge map will also be removed.

Non-maximum suppression is an edge thinning technique in which we compare the grad of each pixel with the grad of the neighbor pixels which are in the positive and negative grad direction. If it is greater than them, we keep it and otherwise remove it to make the edge thinner. In fact, we are choosing the local maxima of the gradient as the only edge pixel.

If the gradient value at any pixel is greater than the high threshold, we consider that pixel as edge and if it is less than the low threshold we consider it as non-edge. If the gradient value is between the low and high thresholds we must check whether this pixel is connected to an edge pixel. If yes, we consider it as edge pixel and if no, we count it as non-edge.

c)

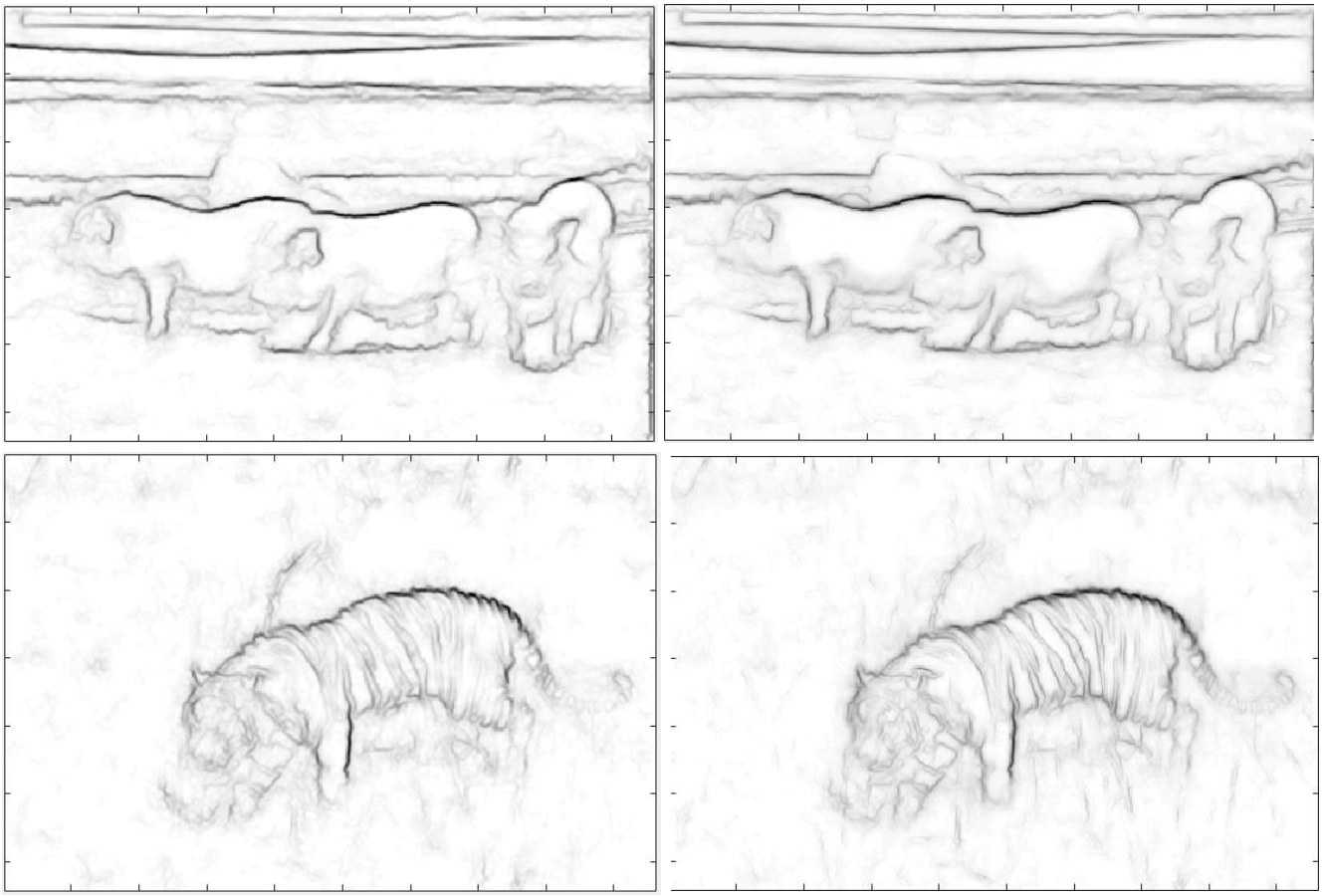


Fig9: output of the SE edge detector without multiscale(left) and with multiscale(right)

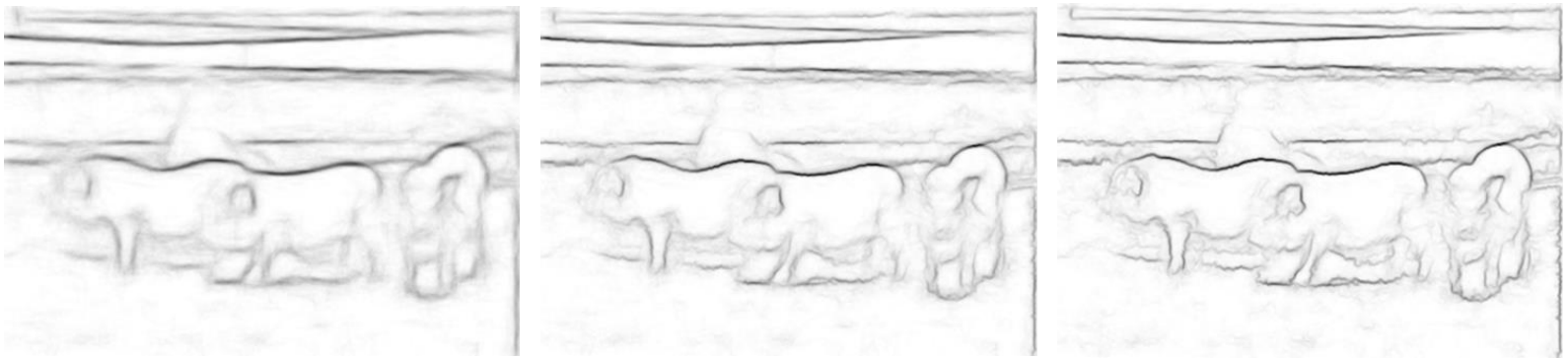


Fig 10: output of the SE edge detector with sharpening factor 0 (left), 1 (middle), and 2 (right)

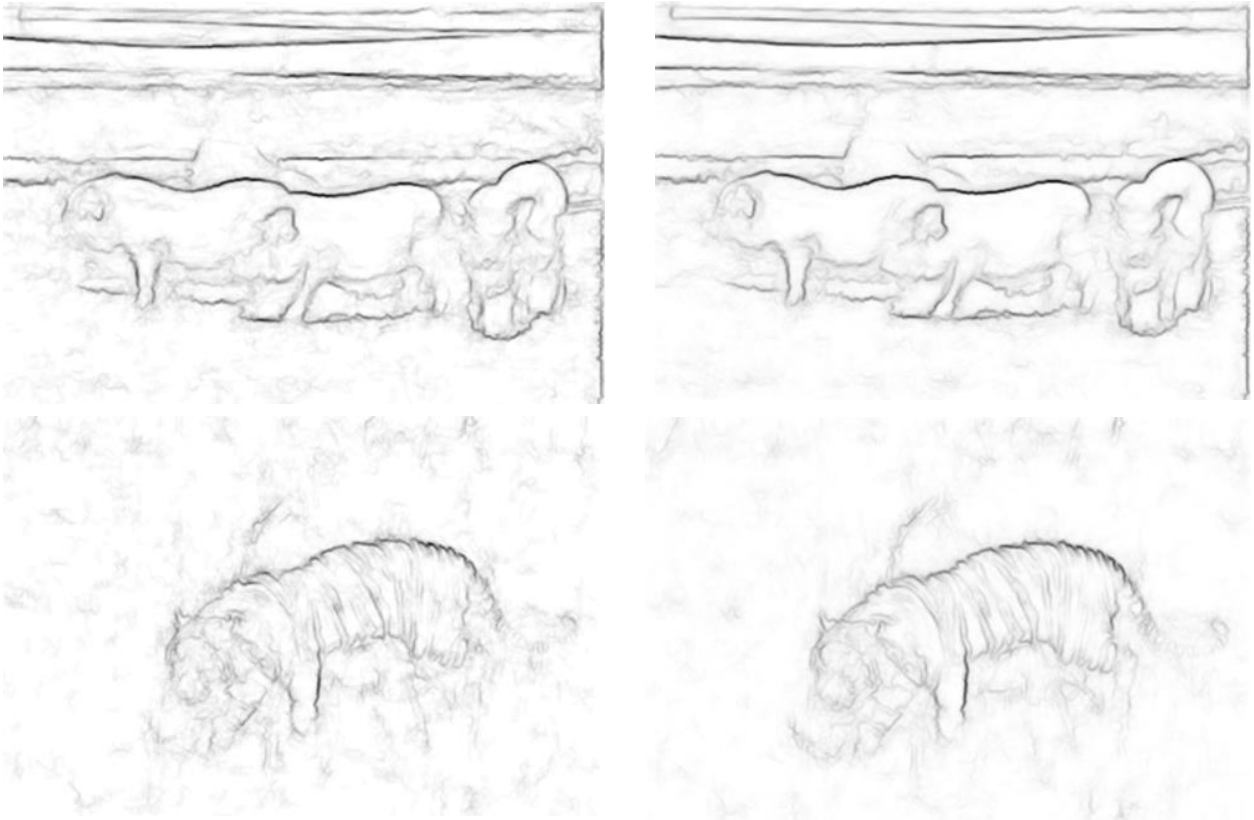


Fig 11: output of the SE edge detector with numer of evaluation trees 1(left) and 4 (right)

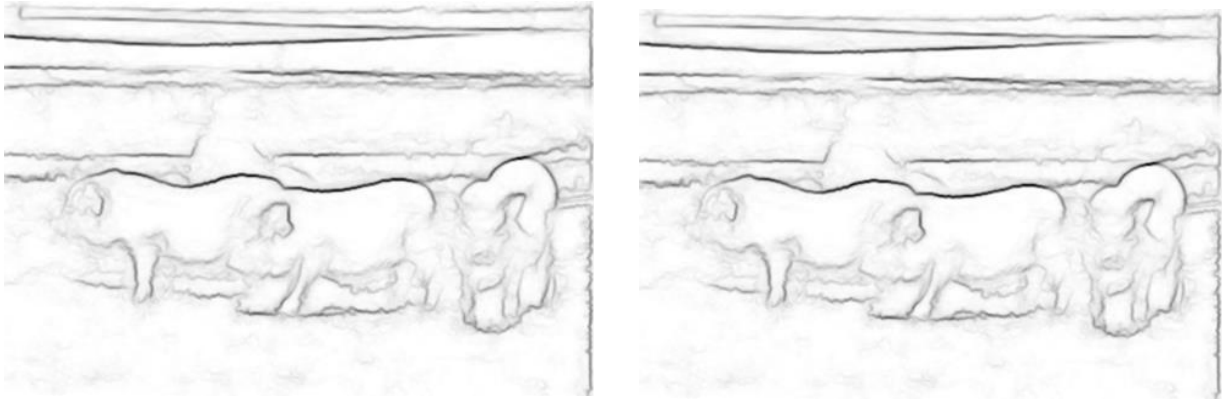


Fig 12: output of the SE edge detector with the max num of threads for evaluation=1 (left) and =4 (right)

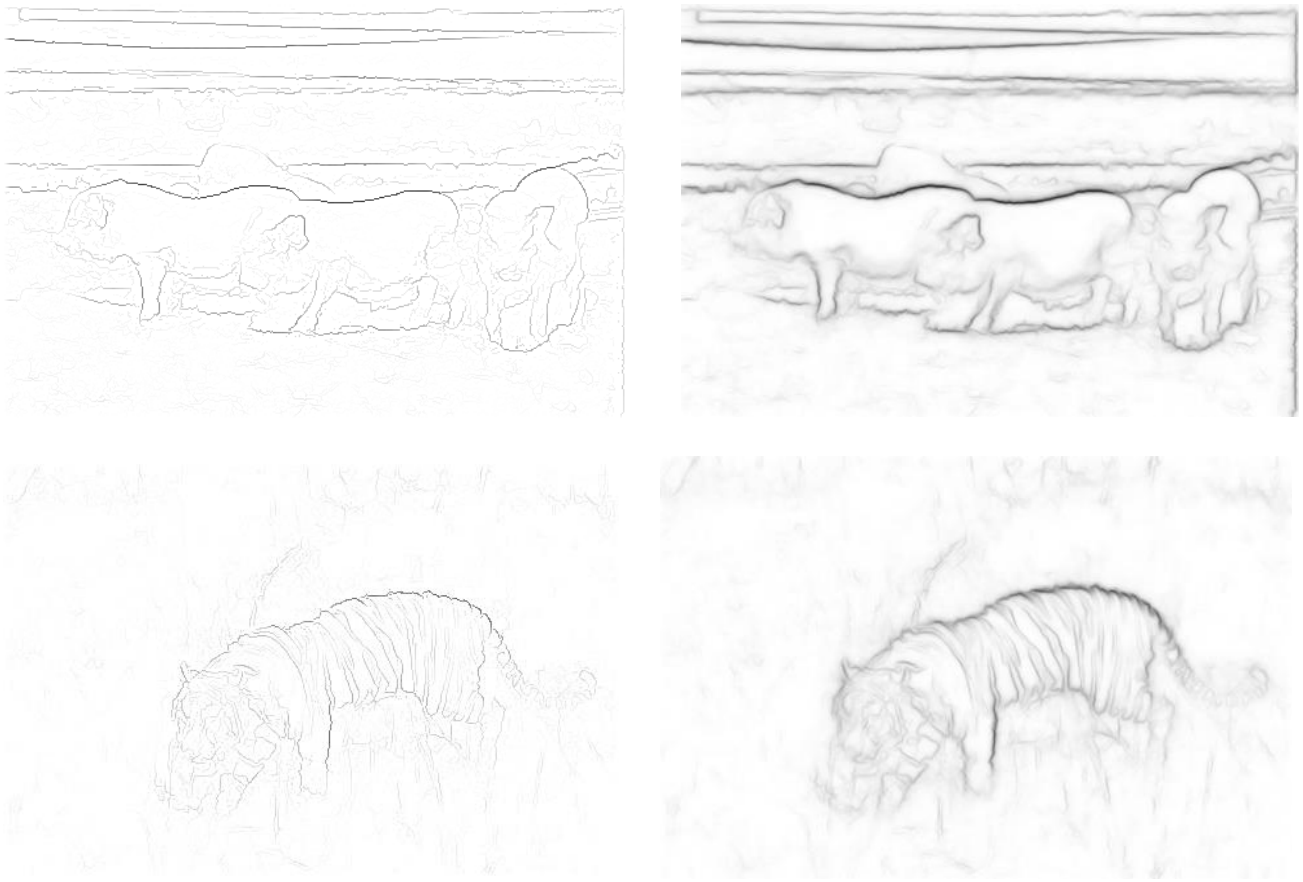


Fig 13: output of the SE edge detector with (left) and without (right) NMS

1) Below you can find the diagram of SE algorithm:

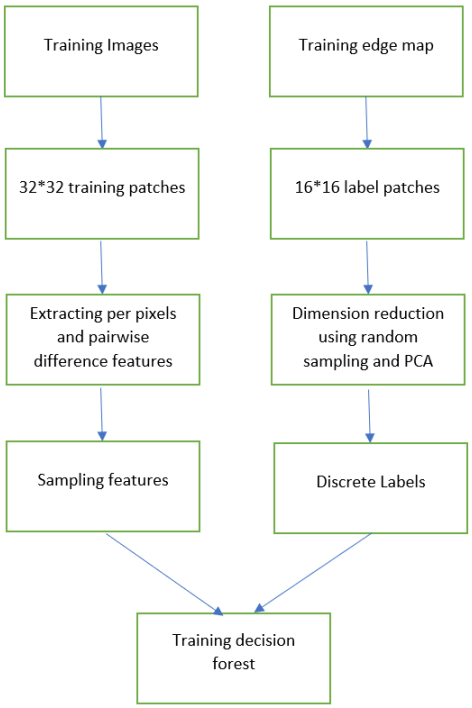


Fig 14: Diagram of the training part of SE

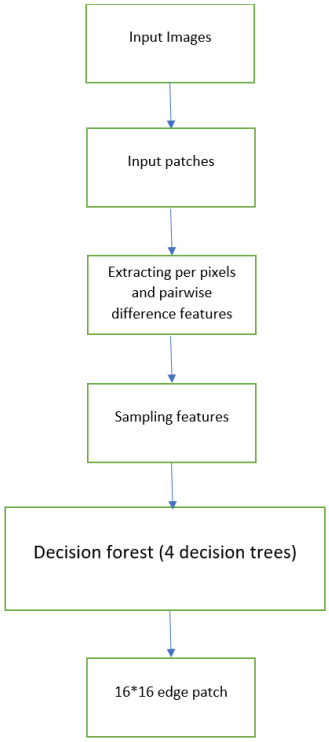


Fig 15: Diagram of the test part of SE

SE EDGE DETECTOR (one-page description): In the SE approach, to obtain the desired label for image patches, we map all the edge structure labels (y which is a 16*16 block) to an intermediate space z, as computing similarities over y is not easy. Z is a long binary vector with size C(16*16,2)=32640 which show whether every pair of pixel in y belong to the same segment or not. To reduce the dimension of z we sample 256 dimensions of it and finally use PCA to reduce its dimension to 5. We can use two approaches to obtain discrete labels C given z. One is to use K-mean clustering, but in SE PCA is used to reach discrete labels as it is faster than K-means. We use the resulting discrete labels for training a random forest model.

As the feature vector we use the following features per pixel: 3 color channels, 2 magnitude of gradient and 8 direction of gradient channel which add up to 16*16*13=3328 features. The pairwise difference is also computed in this way: first we down sample channels to 5*5 blocks and then compute their pairwise difference which add up to C(5*5,2)*13 features. So overall 7228 features can be used. For training each random tree a random sample of these features are used.

To build the decision forest, 4 trees are used and to make trees more diverse (which leads to a higher accuracy) random sampling of data to train each tree and random sampling of the mentioned features to train each node in each tree is done. I have explained about constructing Random forest below (part 2).

For testing, we average the result of each edge map (4 edge maps which are outputs of random trees) which yields a soft edge response. The output is a probability edge map and each pixel will have a value between 0 and 1. We can use non-maximum suppression to make the edges sharper. This method is done per input patch so is highly parallelizable. [1]

2)

Random forest consists of multiple independent decision trees and use their output to produce a single output. Whether by averaging the output of trees or by majority voting.

Decision tree: Assume we have N training data with labels (N is very large) we select a subset of them randomly (Ns). We also have K features for each dataset and randomly pick a subset of them (Ks). Using Ns and Ks we form a decision tree. A decision tree classifies an input by leading it to left or right at each branch to reach a leaf. Each node is associated with a function and a threshold (based of training features) and sends an input to the left or right according to it. In training trees, we design this function and set its threshold so that it maximizes the information gain. Training stops when a certain depth is achieved.

3)

Fig 9-13 are the output results. Considering the images, multi scaling improves the results. As large-scale window is reliable but poor in localization and small-scale window captures details but gives many false positives. Using both helps in reaching a better result. Increasing the sharpening factor makes the output edges sharper and using NMS makes the output edges thinner. Increasing the number of trees from 1 to 4 increases the accuracy and leads to a better edge map. Increasing the max number of threads can speed up the edge detection process in case of large size images, but here it didn't change the time that much. (The process is fast and simple enough to be done with a single thread)

Comparing with the previous approaches for edge detection shows SE can remove complicated background edges better and gives a better edge map. The reason is that in the training data we don't label such complex patterns as edge, so SE won't consider them as edge.

d)1)I have used sobel output with threshold 90% for both Pig and Tiger, Canny with high_th=.34 low_thr=.11for Pig, Canny with high_th=.31 low_thr=.175 for Tiger and SE with multiscale=1, sharpen=2, nTreesEval=4, nms=0.

PIG Image	G1	G2	G3	G4	G5
SE Pm	0.54	0.55	0.66	0.84	0.66
SE Rm	0.33	0.33	0.32	0.34	0.32
SE Fm	0.41	0.41	0.43	0.49	0.43
SE F-max	0.5716				
Sobel Pm	0.13	0.13	0.18	0.22	0.19
Sobel Rm	0.69	0.66	0.56	0.54	0.58
Sobel Fm	0.22	0.22	0.27	0.31	0.28
Sobel F-max	0.2656				
Canny Pm	0.25	0.26	0.37	0.51	0.37
Canny Rm	0.73	0.70	0.63	0.69	0.64
Canny Fm	0.37	0.38	0.47	0.59	0.47
Canny F-max	0.4652				

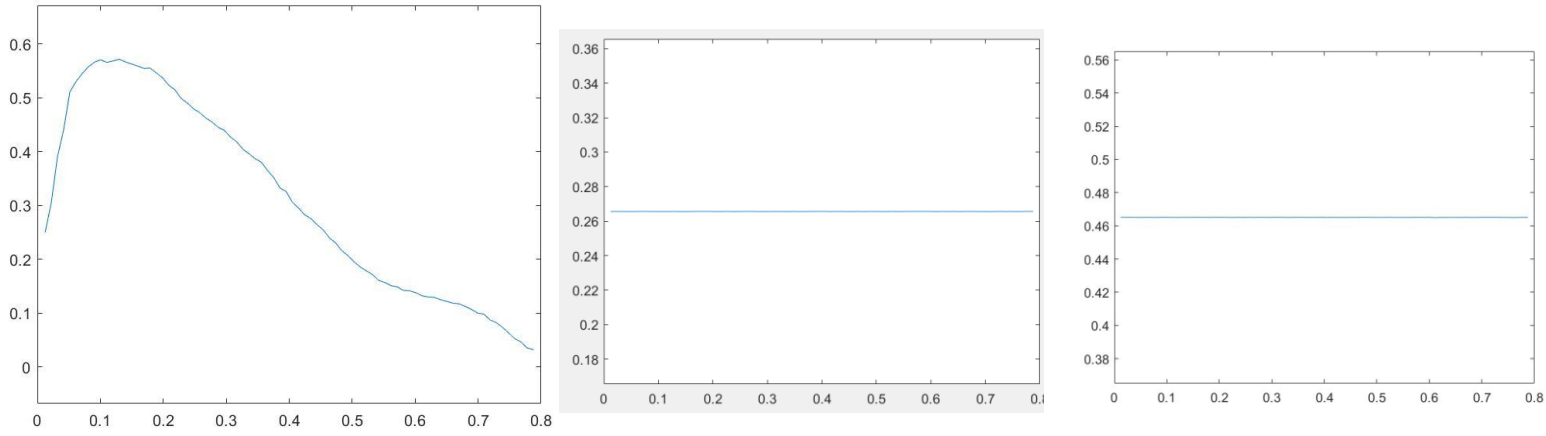


Fig 15: Diagram of the mean of F measure of SE(left), Sobel (middle), and Canny (right) over models based on different thresholds (x axis)-Pig

Tiger Image	G1	G2	G3	G4	G5
SE Pm	0.72	0.73	0.74	0.93	0.72
SE Rm	0.31	0.32	0.30	0.17	0.25
SE Fm	0.43	0.44	0.43	0.28	0.37
SE F-max	0.5245				
Sobel Pm	0.09	0.09	0.11	0.43	0.10
Sobel Rm	0.86	0.91	0.91	0.93	0.76
Sobel Fm	0.16	0.17	0.19	0.57	0.18
Sobel F-max	0.2656				
Canny Pm	0.15	0.17	0.19	0.69	0.17
Canny Rm	0.83	0.88	0.88	0.83	0.69
Canny Fm	0.25	0.28	0.31	0.75	0.27
Canny F-max	0.41				

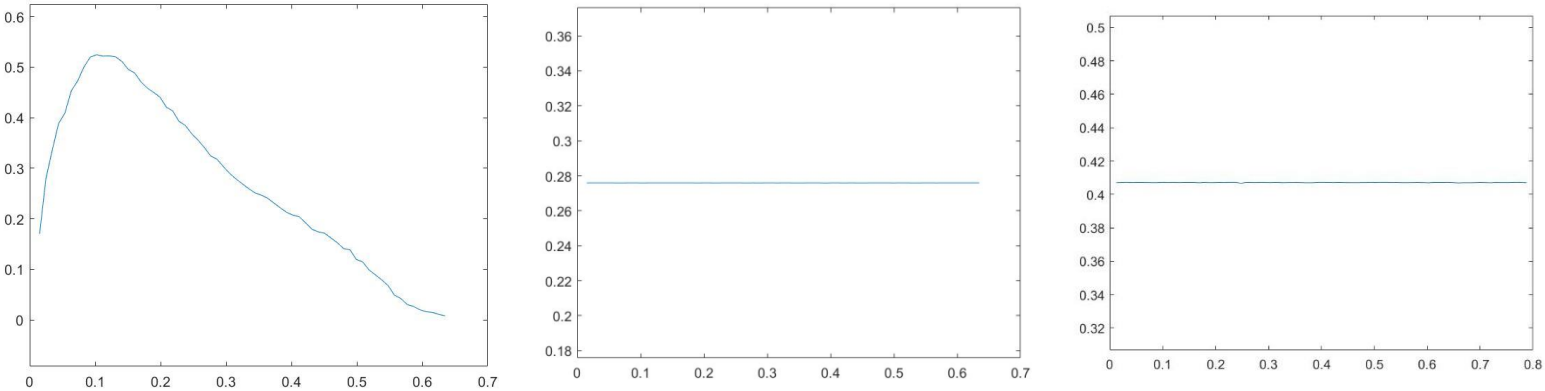


Fig 16: Diagram of the mean of F measure of SE(left), Sobel (middle), and Canny (right) over models based on different thresholds (x axis)-Tiger

2)Pig is easier to get a high F measure, As in Tiger image the background has a lot of complexity (grass) while they shouldn't be detected as edge by the edge detector.
3) it is one over the average of 1/p and 1/r: we want both parameters (p and r) to be considered in it. So, we first average 1/p and 1/r. But we want F to be the bigger the better the accuracy of edge detection, so we compute 1 over that value.
No, if one of them is much smaller it affects F and reduces it significantly.
P+R=C -> F=2*(C-R)*R/C -> F is max when d((C-R)*R)/dR=0 -> 2R=C -> R=P=C/2

Discussion

The Sobel edge detector is easy to implement and fast, but is doesn't have a satisfactory accuracy, it is highly dependent on the threshold and produces disconnected and thick edges which is not pleasant.
Canny have some improvements over Sobel and produces thin and connected edges, but it still can't avoid considering complex background as edge.
SE is better than both in terms of performance (F measure), as it doesn't consider complex patterns as edge, the reason is we don't train it to consider very complex patterns as edge, so it doesn't. Its test time is also fast, and it produces the result of each block independent of other blocks, so is highly parallelizable which is great.

Problem 2

Abstract and Motivation:

For printing machines, it is hard to produce 255 level of intensity for gray scale images. Many printing devices can only produce black and white levels. On the other hand, human perception averages neighboring pixels (just like a low pass filter), so we can use higher density of black dots to show darker parts and lower density of dots to show brighter parts. This is called Halftoning. There are several methods for halftoning including Random Thresholding, Dithering, and error diffusion.

Approach and Procedures:

Random thresholding

In this approach we choose a random number between 0 and 255 as the threshold at each pixel and if the value of pixel is greater than the threshold we set it to 1 and otherwise to 0.

$$op(i,j) = \begin{cases} 0 & 0 \le ip(i,j) < rand(i,j) \\ 255 & rand(i,j) \le ip(i,j) < 256 \end{cases}$$

Dithering

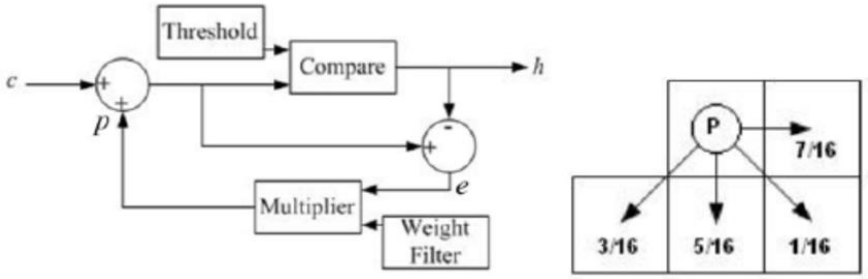
In this approach we define a threshold matrix with size of a power of 2. (2, 4,8 or…). In this matrix we have a distribution of different thresholds so that low or high threshold don't form a specific concentration. Then we divide the input image into blocks with the same size and in each block compare the grayscale value of the pixel with the corresponding threshold in the threshold matrix. If it was greater than the threshold we set it to 1 and otherwise to 0.
Threshold matrices are computed as follows:

I2= $\begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix}$ $I_{2n}(i,j) = \begin{bmatrix} 4\times I_n(i,j)+1 & 4\times I_n(i,j)+2 \\ 4\times I_n(i,j)+3 & 4\times I_n(i,j) \end{bmatrix}$

Threshold matrix: $T(x,y) = \frac{I(x,y)+0.5}{N^2} \times 255$

Error Diffusion

In this method we set a fixed threshold (127) but diffuse the error of half toning of each pixel to the neighboring pixels. There are several methods for distributing the error among **future** neighboring pixels, for example Floyd-Steinberg:
(Future means the process has not done on that pixels yet, so order of processing pixels matters)



Color half toning with error diffusion

In this method we first turn the RGB values into CMY values and then do any of the error diffusion methods per channel. (for each of the CMY channels)

MBVQ-based Error diffusion

Human perception is sensitive to brightness changes. This method tries to minimize brightness changes during half toning. It defines 6 regions with minimum brightness variation and if a pixel is within one of these regions, it maps the image to its nearest vertex inside that region and then

does the error diffusion per channel. In this way the brightness of the pixel will have the minimum variation.

Results

1)

Fig 17: original gray-scale image (left) and output of halftoning using Random thresholding method

As you see this is not a good approach, as it the output has a granular appearance because of the low frequency noise. (It seems like there is dirt on the output image)

2)

Fig 18: output of halftoning using Dithering method with I2(left), I8(middle), and I32(right)

As you see using bigger thresholding matrices provides lots of different levels of threshold in each small region and so is more capable of showing density changes and details.

b)

Fig 19: output of halftoning using error diffusion method with Floyd-Steinberg's (left), JJN (middle), and Stucki (right) error diffusion matrix

The result of using the last two error diffusion matrices (bigger matrices) is clearer. For example, the vertical bars of Golden gate are not clear in the left image, so diffusing error to a larger area works better. From the last two matrices, Stucki's output seems more distinct.

In comparison with dithering method, error diffusion has a better performance. In the output of the dithering method small blocks of change within the output are visible while the latter approach leads to a smoother output in which gradually changes in density is more acceptable.

One idea to improve the current result is to try larger matrices as the error diffusion matrix. Another one is to change the scanning order to be able to spread the error more uniformly. We can first divide the image into 5*5 blocks and do the quantization for the central pixel in each block and diffuse its error to all the neighboring pixels. (within a 5*5 window). Next, we do it for the pixel in the position of (5,5) within each block and diffuse its error to its neighboring pixels within a 5*5 window (except those ones which are already quantized) and continue this approach to finally quantize all the pixel values. This approach leads to a more uniform error diffusion and leads to a smoother image.

c)

Fig 20: output of color halftoning using Separable Error Diffusion (middle), and MBVQ-based Error diffusion (right)

The result of separable error diffusion method is acceptable (like the result of error diffusion method for gray-scale image), the only shortcoming is that it may have some color distortion.

1) Human perception is very sensitive to brightness changes. MBVQ method tries to do the halftoning in a way that minimizes brightness variations of the output. It determines 6 regions inside the rgb color space with minimum brightness variation and the first step is to find the region that the pixel initially is. Then we find the nearest vertex to it inside that region and change its value to rgb values of that vertex and then do the error diffusion per channel. In this way, we make sure the brightness of the output has minimum variation.

2) Both outputs are satisfactory, but the output of MBVQ method preserves the brightness of the image a bit better.

Discussion

Random thresholding method was an improvement over fixed threshold halftoning. As it couldn't show local variation in the image. Random thresholding gives the pixels with a gray-scale value less than 127 a chance to turn to white dots after halftoning. The closed they are to 127, the more likely they are to exceed the threshold as it is a random number between 0 and 255.

But in general, this is not considered a good approach and doesn't produce satisfactory results as the random noise has both low and high frequency components while low frequency acts like dirt on the image. (We can also look at the random threshold as a random noise which is added to 127.) We prefer only high frequency noise to be added to threshold called blue noise.

Dithering works like a blue noise and its principal is to not to let low or high threshold values concentrate on some part of the threshold matrix.

Error diffusion tries to compensate the difference between the initial and quantized value of each pixel by diffusing it to the next neighboring pixels and is a smart and effective way for halftoning.

References

[1] P. Dollár and C. L. Zitnick, "Structured forests for fast edge detection," in Proceedings of the IEEE International Conference on Computer Vision, 2013, pp. 1841–1848

[2] Digital Image Processing: PIKS Inside, Third Edition. William K. Pratt