EE569: Homework #2

Yifan Wang
wang608@usc.edu #3038184983

January 30, 2019

1 Edge Detection

1.1 Sobel

To apply Sobel edge detector on color images, converting to grayscale is required. I use formula from [1]:

$$Gray_{val} = 0.299R_{val} + 0.587G_{val} + 0.114B_{val}$$

to calculate gray value.

Then, using convolution cores from [2] to calculate x-direction gradient and y-direction gradient ($Figure 2, 3$). Gradient magnitude is computed by:

$$Grad = \sqrt{(Grad_x^2 + Grad_y^2)}$$

To show edges as the homework required, result images are opposited.

(a) (b)

Figure 1: Color image to Grayscale

1.2 Canny

Canny function from $OpenCV$ [5] is wrapped in my function, it use $Tiger.jpg$ and $Pig.jpg$ as input. To show the edges more clearly, result images are opposited.

1.2.1 Non-maximum Supression

After gradient is computed, pixels whose gradients are above threshold in a small local neighbourhood would make edges wider, leading to blurred edges. To avoid this from happening, non-maximum suppression severs as a sift to get rid of other small gradients by comparing each gradient numerically in positive and negative gradient directions, and only keep the largest one. Result relatively thin and clear edges.

(a) Result (b) Gradient in x direction (c) Gradient in y direction

Figure 2: Sobel on $Pig.raw$

(a) Result (b) Gradient in x direction (c) Gradient in y direction

Figure 3: Sobel on $Tiger.raw$

This idea can be explained more easily in mask RCNN, which would give a bounding box for targeted object for example a cat. However, network would generate several bounding boxes in different size and location (ex: some contain the whole cat, while others only contains part of a cat like its leg or face), to just output one best bounding box, a score is calculated to judge which best contains cat (smallest bounding box contains whole cat), and choose it as the final output bound box. This is the same in Canny, only difference is that score is represented by gradients rather than calculated on image patches.

(a) Canny on $Pig.jpg$ (b) Canny on $Tiger.jpg$

Figure 4

1.2.2 Low and High Threshold

High threshold serves as a strong qualification, which can separate edges from backgrounds with confidence almost 100%. The higher it is, the more likely it is an edge pixel. If a pixel's gradient is higher than this value, it is almost sure this pixel is on edges. However, if we only have high threshold, edges are less likely be continuous, since some parts of edges may not so clear due to variation in contrast or luminance. To make contours look much better and more continuous, a low threshold is chosen to connect these strong edges (general trend: $3T_L = T_H$).

In Canny, on condition that a pixel belongs to edges, if its neighbors can pass low threshold, they would be set as edge points together. Repeat this process, until a closed contour is found. It would be more impressive to use idea of conditional probability to explain it, let $G(i,j)$ be gradient at location (i,j):

$$P(Pixel(i,j) \in Edges) \approx \begin{cases} 1, & G(i,j) \geq T_H \\ unknown, & T_L \leq G(i,j) < T_H \\ 0, & G(i,j) < T_L \end{cases}$$

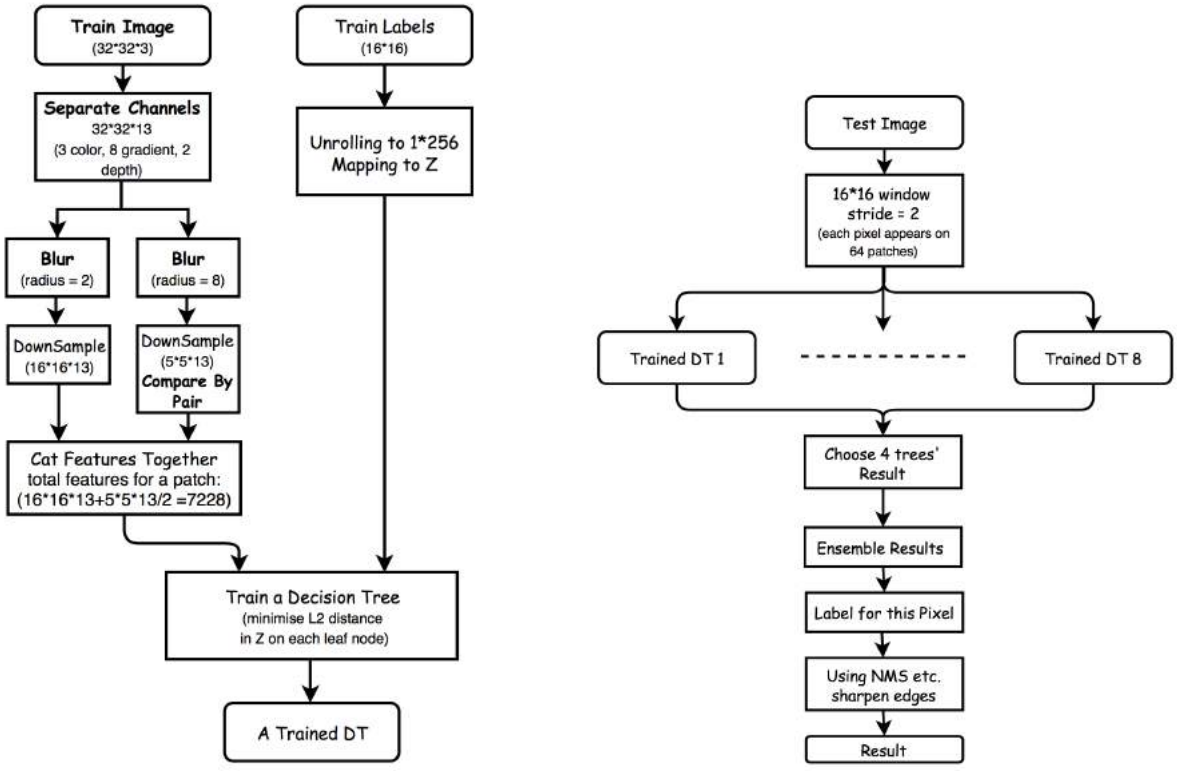
for the case $T_L \leq G(i,j) < T_H$:

$$P(Pixel(i,j) \in Edges | T_L \leq G(i,j) < T_H) \approx \begin{cases} 1, & \text{some of its neighbour} \in Edges \\ 0, & \text{none of its neighbour} \in Edges \end{cases}$$

where T_L refers low threshold, T_H is high threshold.

1.3 Structured Edge

1.3.1 Model Explanation



(a) Training (b) Testing

Figure 5: Structure Egde Detector

Structure edge detector uses several (8 trained, use 4 in paper) [6][7][8][9] decision trees to ensemble a decision forest. This method has a flavor of Bagging, both of them separate training data or features to train several weak learners (each of them does not have the same accuracy as a same tree trained by full dataset), however three cobblers combined makes a genius mind, this is true on weak learners, since one single tree has high variance and tends to be overfitting.

It uses a mapping to map each label mask y to a higher dimensional space Z. This mapping method help to calculate the loss while training decision tree, due to the fact that it would be costly and unstable to calculate loss in Y which is a 16×16 mask. The detail of mapping is : unrolling it to a 1×256 vector, and check similarity between 2 position i, j where $i \neq j$. Resulting a $C_2^{256 \times 256}$ binary sequence. For efficient a subsequence length 256 is used (performs well in test). Then PCA (faster) or K-means is used to get labels from Z space. Calculate distance in such space is super easy. Then, for training a decision tree, it needs to minimize distance in each leaf node, which can be interrupted as maximize similarity among features in leaf nodes.

Training data preprocessing is required. Input is 32×32 image patches. Firstly, for a single patch, it is separated into 13 channels (3 colors, 2 depth, 8 gradient). Then, two blur core is used on each channel. (1) smaller core ($radius = 2$), downsample image to $16 \times 16 \times 13 = 3328$ pixel lookup features; (2) larger core ($radius = 8$), downsample to $5 \times 5 \times 13$ and compare the similarity between each two pixel in each channel, resulting $13C_2^{5 \times 5} = 3900$ pairwise difference features.

With train labels and train data parpared, each decision tree is trained as a normal decision tress to separate different features and minimize information entropy, with techniques like cutting branches if current separate can not improve the tree's performance (which is generally regard as wrong classification).

While testing, a window of size 16×16 slides on test image with stride 2, so that a single tree would evaluate a pixel 8×8 times. Four out of eight trained tree is used, for a single pixel, $8 \times 8 \times 4$ evaluate results are achieved. Then averaging predicted results to ensemble these labels together (sum up and normalize in this case), a final result for this pixel is generated. After that other techniques like NMS is used to get a better and sharper edge result.

The main flavor of Structure Edge detector is using label patches as training data to derive features when there is likely to be an edges in a pixel which is related to its neighbors, contrast, luminance, etc. like dictionaries (one for each decision trees), when a test image is input to it, we just need to derive features from it and look it up in dictionaries and combine their result. It is no longer simply gradient based, more features are included resulting more accurate edges.

1.3.2 Parameter Test

To make the difference between different parameters more obvious, I do not binary edge image. Whether using NMS makes huge difference ($Figure6(f, g)$, $Figure7(f, g)$). Using NMS results a sharper edge which is just what NMS supposed to do as a sift. Pixels with low confidence as edge points are discarded.

Using $sharpen$ ($Figure6(b, c)$, $Figure7(b, c)$) can provide sharp edge as well. But the way is different from NMS. In $sharpen$, edges remains the same size, but looks like adding contrast to image by removing low frequency part in result (like using second order derivative ex Laplacian), which means if a pixel's gray value is closer to 0 than 255 (or some other thresholds), then make it more close to 0, vice versa.

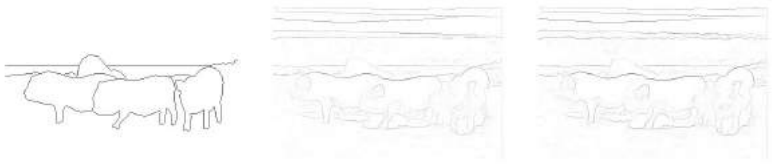
For $mutiscale$, using it would blur and widen edges ($Figure6(d, e)$, $Figure7(d, e)$), which can be explained that more information is included in deciding edges, resulting averaging. Finding on pyramid size of images can provide more accurate edges. (Some edges in large scale image may not be regarded as edges, but if downsample image, they can be detected. Then upsample edge map to original scale and combine with others (causing blur in this step).) In that case, it would be helpful when edges are not clear or an accurate result is needed. Through combining with NMS and $sharpen$, a nice, accurate and thin edge can be achieved.

While number of decision trees used in voting affect less than previous techniques. A fair number of trees can improve accuracy on details, result a more accurate edge map. ($Figure6(h - k)$, $Figure7(h - k)$)

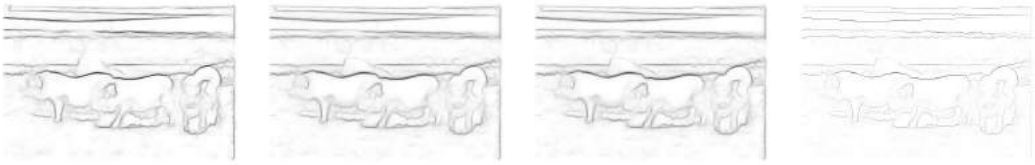
To have a sharper edge before binarize, NMS and $sharpen$ should be used. And to make it more accurate, $multiscale$ and fair number of trees should be used. Binarized result are shown in $Figure8$ both use $multiscale$, NMS, $sharpen = 2$, and 4 decision trees.

1.4 Evaluation

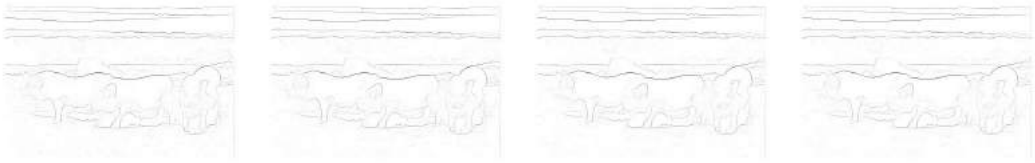
Code for evaluation is in Appendix.



(a) GroundTruth (b) 1_0_4_1 (c) 1_2_4_1



(d) 0_2_4_0 (e) 1_2_4_0 (f) 1_2_4_0 (g) 1_2_4_1



(h) 1_2_1_1 (i) 1_2_2_1 (j) 1_2_4_1 (k) 1_2_8_1

Figure 6: Structure Egde Detector on $Pig.jpg$
parameters: $mutiscale\_sharpen\_nTreesEval\_nms$

(a) GroundTruth (b) 1_0_4_1 (c) 1_2_4_1

(d) 0_2_4_0 (e) 1_2_4_0 (f) 1_2_4_0 (g) 1_2_4_1

(h) 1_2_1_1 (i) 1_2_2_1 (j) 1_2_4_1 (k) 1_2_8_1

Figure 7: Structure Egde Detector on $Tiger.jpg$
parameters: $mutiscale\_sharpen\_nTreesEval\_nms$

GroundTruth	Sobel			Canny			StructEdge		
	P	R	F	P	R	F	P	R	F
$\#1$	0.09	0.84	0.16	0.13	0.70	0.22	0.27	0.85	0.40
$\#2$	0.09	0.83	0.16	0.14	0.69	0.23	0.29	0.84	0.43
$\#3$	0.13	0.72	0.21	0.17	0.53	0.26	0.53	0.72	0.61
$\#4$	0.16	0.71	0.26	0.23	0.57	0.33	0.71	0.78	0.74
$\#5$	0.12	0.71	0.21	0.17	0.55	0.26	0.52	0.73	0.60
$Overall$	0.22	0.75	**0.34**	0.31	0.59	**0.40**	0.77	0.79	**0.78**

Table 1: Edge Detection score on $Pig.raw$
using function in $edgesEvalDir.m$ to calculate

GroundTruth	Sobel			Canny			StructEdge		
	P	R	F	P	R	F	P	R	F
$\#1$	0.07	0.92	0.13	0.41	0.96	0.14	0.79	0.41	0.45
$\#2$	0.07	0.91	0.14	0.08	0.93	0.14	0.51	0.55	0.48
$\#3$	0.08	0.92	0.15	0.08	0.93	0.16	0.39	0.61	0.47
$\#4$	0.34	0.97	0.51	0.36	0.98	0.52	0.74	0.73	0.74
$\#5$	0.07	0.73	0.14	0.08	0.79	0.15	0.41	0.52	0.46
$Overall$	0.38	0.91	**0.53**	0.41	0.93	**0.57**	0.81	0.79	**0.80**

Table 2: Edge Detection score on $Tiger.raw$
using function in $edgesEvalDir.m$ to calculate

1.4.1 Different Detectors

For Sobel detector, only one threshold controls percentage of edge pixels which could meet situation like: when percentage to keep edges is high, lots of pixels belongs to none edges would be kept produce a mess edge map, turning threshold down eliminating these pixels while also rejecting some edge pixels which have similar gradients as noise pixels. Resulting a less continuous edge image. So that its precision, recall and F measure compare with ground truth is low. And it tends to have wider edges as well.

For Canny detector, two thresholds gives more room to adjust result. To some extent, it helps to keep more edge pixels while suppress noise. But this also release a problem when a region pixels all have gradients higher than low threshold, through Canny detector's way to judge edges, if one of these pixel have gradient higher than high threshold, all the rest pixels would be regard as edge pixels. Leading to wide edges. That is one possible reason its precision, recall and F measure are higher than Sobel but still not pleasing. What's more the use of edge tracking by hysteresis allows to get rid of weak edges and strong but not connected edges.

Comparing with Canny or Sobel edge detector, structure edge detector is less like to be affected by slightly change in part where contains no edge, it would keep pure edge since besides considering gradients, more features are included in deciding whether it is a edge point or not. With the help of techniques like non-maximum suppression and sharpen, it can get a thiner edge. Giving a high F measure score.

Structure Edge detector requires more time to compute edges maps. Time complexity for sobel edge detector on $N \times M$ image is $O(N \times M)$, using $N \times M \times (2 \times 8 + 1)$ additions, $N \times M \times (2 \times 9 + 2)$ multiplications and $N \times M$ square roots If coding separately rather than using general convolution, it can be reduced to $N \times M \times (2 \times 5 + 1)$ additions and $N \times M \times (2 \times 6 + 2)$ multiplications. For canny edge detector, slightly more addition and multiplication is needed, but time complexity remains in $O(N \times M)$. While for structure edge detector, get image patches, transfer to features, put into several trees and get final result need more time (matrix operation in matlab shows it's advantages, implicitly use for loop as matrix operation is better than explicitly for loop).

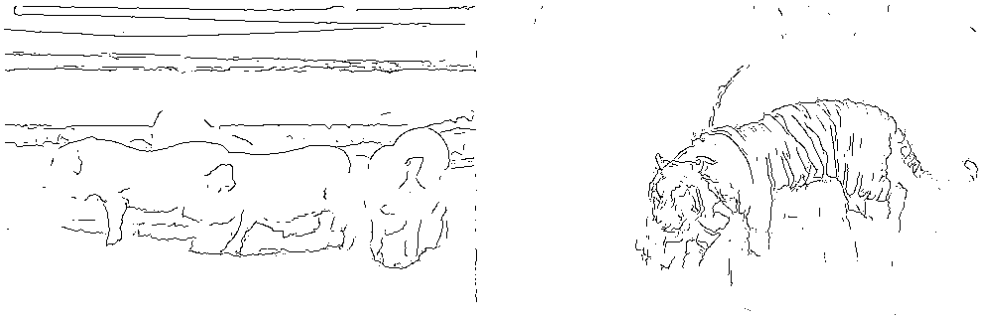
1.4.2 Different Image

For different image, $Pig.raw$ ($Table1$) and $Tiger.raw$ ($Table2$). Result would be different, it seems more easy to detect edges on tiger with strong contrast black stripes and fur which means high gradient. Even though it seems grass in background may disturb result, with proper thresholds, these can be eliminated.

Considering the ground truth, most people would concentrate on shape and stripes only. However, situation becomes more tough when it comes to $Pig.raw$. most people would consider shape of pigs as edges when labeling ground truth. But shadows of pigs and fences would not be considered, which it supported by visualize 5 ground truths, none of them consider shadows. While gradient at these regions is relatively high, and can be regard as edges in Canny and Sobel detector. This is one reason cause that score for $Pig.raw$ on Canny and Sobel only around 0.4.

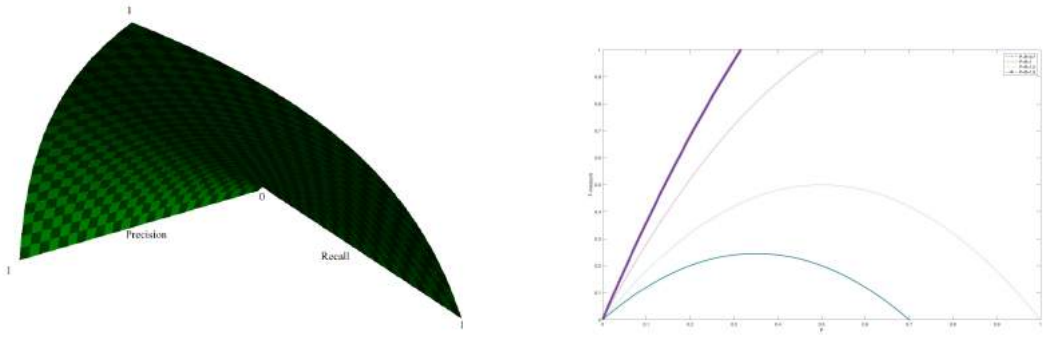
1.4.3 Others

Even though it is true that structure edge detector performs superior to Canny detector, Canny detector is better than Sobel detector. It should be aware that precision, recall and F measure can not tell all. Since structure edge detector is trained by labels have almost same flavor with ground truth of $Pig.raw$ and $Tiger.raw$, for example shadow problem discussed in previous paragraph, it is reasonable to say structure edge detector is trained to consider less on these part due to its training data. This reason would add to the final score as well.



(a) $Pig.jpg\ p > 0.15$ (b) $Tiger.jpg\ p > 0.1$

Figure 8: Binarized result of Structure Edge Detector



(a) draw by Grapher.app (b) F when $P + R = const$ (draw by Matlab)

Figure 9: Relationship between P, R and F

1.4.4 F Measure

From $Figure9(a)$, it can be conclude that to get a higher F score, both P and R should be high, otherwise would result a low F score, since the smaller one in $\frac{2PR}{P+R}$ would contribute more to F score than larger one. If $P + R = const$ F score would approach its maxine when $P = R$ ($Figure9(b)$). This can be proved by:

$$(\frac{1}{P} + \frac{1}{R})^2 \geq \frac{2}{PR}$$

$$2(\frac{PR}{P+R})^2 \leq PR$$

where

$$PR \leq \frac{(P+R)^2}{2}$$

so that

$$2(\frac{PR}{P+R})^2 \leq \frac{(P+R)^2}{2}$$

$$F = \frac{2PR}{P+R} \leq P + R$$

all the equality above would stand when and only when $P = R$ ($0 \leq P \leq 1$, $0 \leq R \leq 1$, $0 \leq F \leq 1$). Which can be directly observed from $Figure9$.

2 Digital Half Toning

(a) $bridge.raw$ (b) Random Thresholding)

Figure 10

2.1 Dithering

2.1.1 Random Thresholding

Using random thresholding comes from uniform distribution ($U(0, 255)$) and the formula from homework notes.

Result from random thresholding ($Figure10(b)$) seems noisy, lots of details are lost. No delicate grayscale changes in clouds remain. A feeling like watching black and white television produced in 1980s.

If input is a color image ($N \times M \times 3$), it needs $N \times N \times 3$ $double$ multiplications and $N \times M \times 2$ double additions to convert it into a grayscale image (time complexity: $O(N \times M)$). Then for a using Dithering on this grayscale image's time complexity is $O(N \times M)$. If combine these together, time complexity remains $O(N \times M)$, faster speed can be achieved if combine convert grayscale and thresholding together pixel by pixel in same loops (On condition that we do not consider other techniques like blocking or unrolling for loops which can improve cache hit rate.)

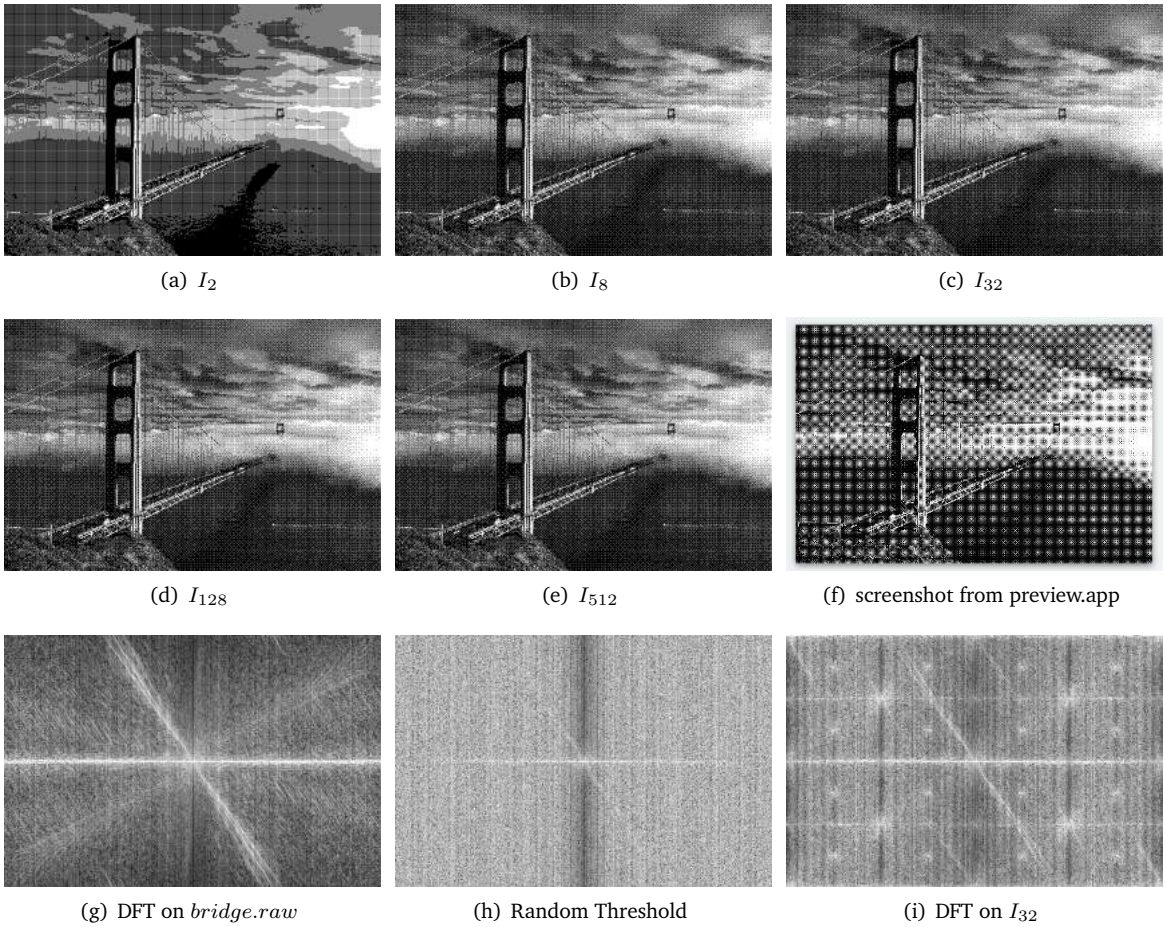
2.1.2 Dithering Matrix

Using index matrix [10] as threshold. Any order of index matrix is generated by using tensor product and similar tensor addition operation.

This method works pretty well especially when index matrix is large. When index matrix is I_2, it keeps less detail and whole image does not seem continuous. However, due to image size not the larger index matrix, the better. I_{128} ($Figure11(d)$) and I_{512} ($Figure11(e)$) almost have same results from human eyes. Problem is that in some part where grayscale is similar, white and black pixel tend to arrange in a same pattern. Bigger problem comes when moving of changing result image in certain way like change its into certain size would cause situation in $Figure11(f)$, because of similarity pattern in index matrix used, it can be proved by convert image to frequency domain ($Figure11(g-i)$). Dithering matrix added some frequencies which do not exist in raw image, when we view result in a monitor have similar frequencies would give terrible visual experience. While uniform noise used in random thresholding blur some frequencies in raw image, resulting losing some details.

Overall, dithering matrix tends to decrease image's contrast, and increase it's brilliance in bridge image where low gray value dominates the images. It would make dark part brighter.

Time complexity for generating an index matrix $(N \times N)$ is $O(log_2(N)N^2)$, and comparing on a $H \times W$ image is $O(H \times W)$. Generally, the time complexity is $O(H \times W)$ compare the small size of index matrix.



(a) I_2 (b) I_8 (c) I_{32}

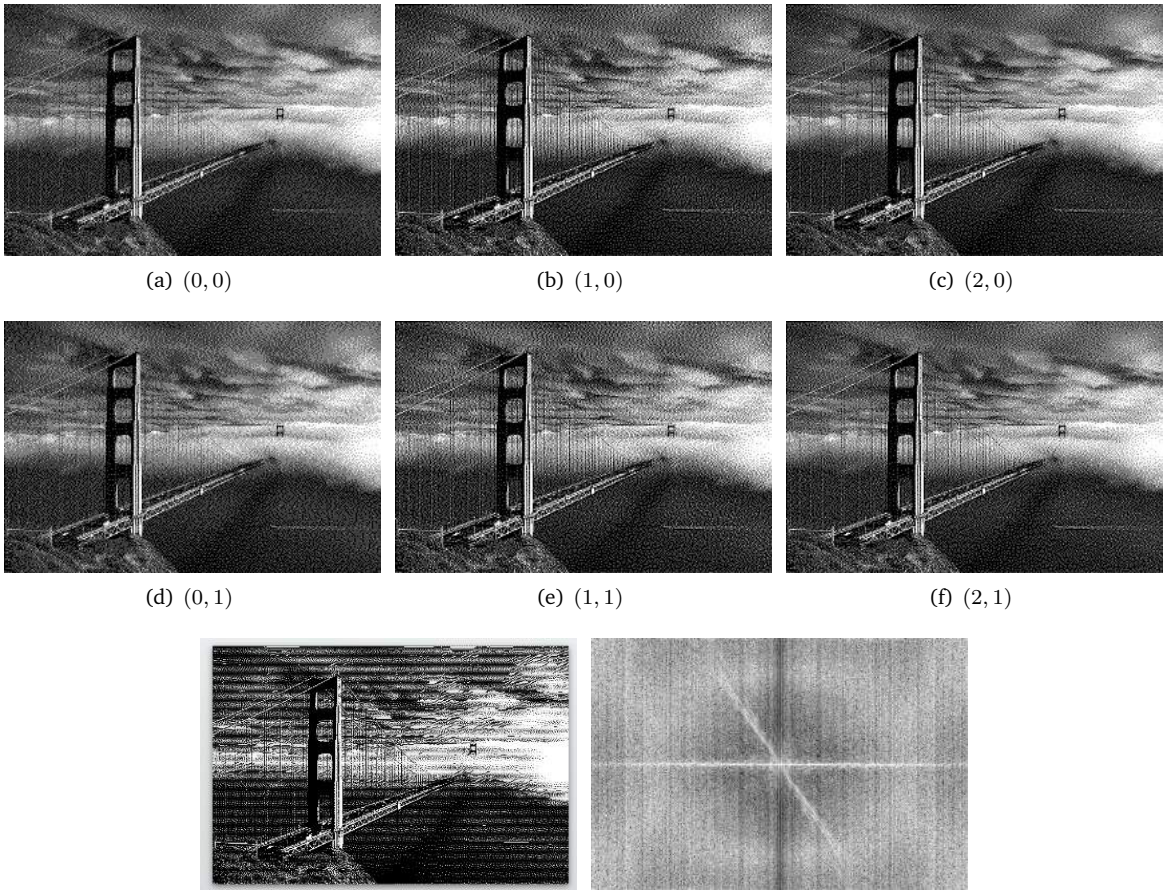
(d) I_{128} (e) I_{512} (f) screenshot from preview.app

(g) DFT on $bridge.raw$ (h) Random Threshold (i) DFT on I_{32}

Figure 11: Dithering Matrix and DFT on $bridge.raw$
Matlab's fft function is used on (g-i)

2.2 Error Diffusion

Notation: In error diffusion $(version, spread)$ means:
$version$: $0 = Floyd - Steinberg$, $1 = JJN$, $2 = Stucki$;
$spread$: $0 = RasterParsing$, $1 = SerpentineParsing$, $2 = HilbertParsing$.



(a) $(0,0)$ (b) $(1,0)$ (c) $(2,0)$

(d) $(0,1)$ (e) $(1,1)$ (f) $(2,1)$

(g) screenshot from preview.app (h) DFT on Error Diffusion $(1,1)$

Figure 12: Error Diffusion on $bridge.raw$
Matlab's fft function is used to get (h)

All of three versions of error diffusion matrix is realized in a single function with parameter $version$ and $spread$ (details see Notation above).

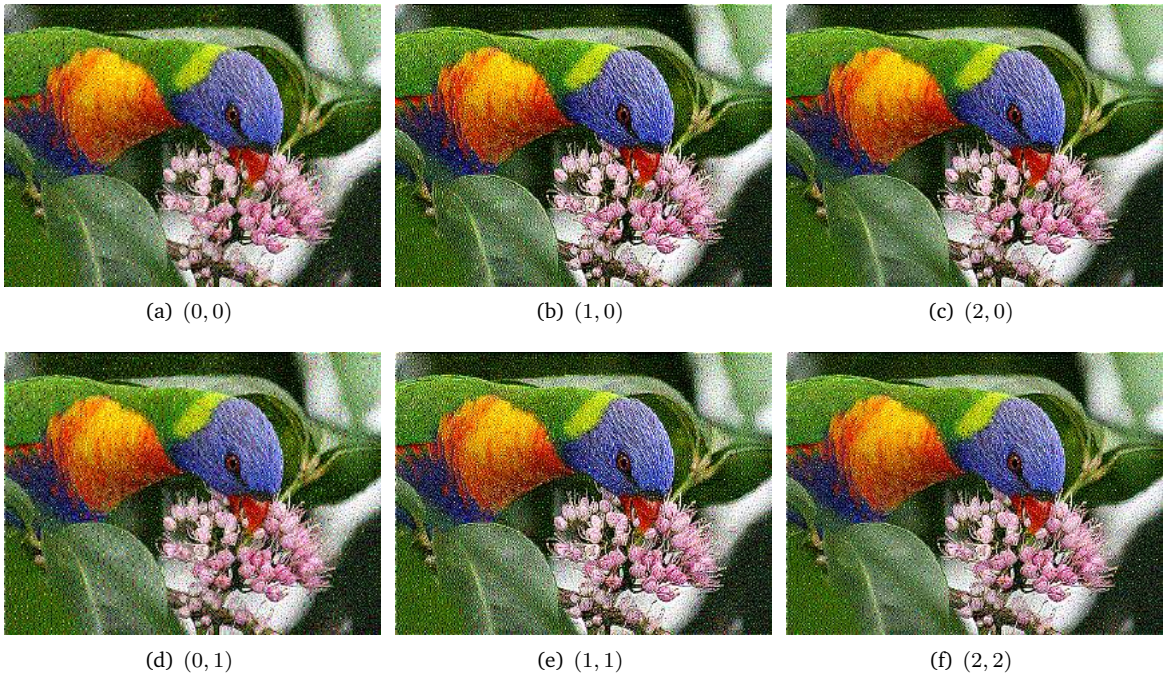
Considering method of sweep $(Figure12)$, Raster parsing would result a relatively lower contrast and less details in comparison with Serpentine parsing which can be interrupted as not evenly distribute errors in raster parsing. However both of sweeping methods have some artificial patterns (checkerboard like) when viewed at certain condition $(Figure12(g))$ which comes from sweeping methods used.

Floyd-Steinberg matrix keeps less detail than JJN or Stucki matrix which leading to much smoother and more subtle images, but difference between these two method are not obvious on visual experience (at least on this image, to me there are no difference). Main difference is related to running speed.

It is quite fast to run Floyd-Steinberg method, including the fact that it small diffuse range and bit shift can be used rather than dividen on normalization. While on JJN division by 48, Stucki division by 42 cannot operate like this, but Stucki operate faster than JJN since all of its coefficients are power of 2 in matrix which can use bit shift to speed up before normalizing.

Considering the memory needed, ideally, for Floyd-Steinberg method, one row size memory is need to store error diffused to next row, while it doubled in JJN and Stucki with the increasing size of diffuse matrix.

Improvement is discussed in 2.4.



(a) $(0, 0)$ (b) $(1, 0)$ (c) $(2, 0)$

(d) $(0, 1)$ (e) $(1, 1)$ (f) $(2, 2)$

Figure 13: Separable Error Diffusion (opposited) on $bird.raw$

2.3 Color Halftoning with Error Diffusion

2.3.1 Separable Error Diffusion

Realized by using function in previous section with parameter $color$. If color is true and input is a color image, it would be converted to CMY and processed as homework required. Results are converted back to RGB in this report for the convenience of comparison. If color is false and input a color image, it would be converted to grayscale and processed as a grayscale image.

One shortcoming of this method: adding artificial patterns remains since general flavor of process remains the same as grayscale images. Another most important problem is that it does not consider human eyes' characters in it, human eyes have different sensitivity on brilliance, contrast, saturation etc. This method is originally designed for grayscale image, when converting to color image, it just regards all of these error as same weight including some errors unique to color images like different brilliance in different color. Situation like error from contrast would be given to brilliance, resulting a un-nature feeling. To make it perform better, weighted error should be used on different type of errors like different color, brilliance etc.

A multiplier is added to time complexity and memory needed of grayscale error diffusion because of 3 color channels.

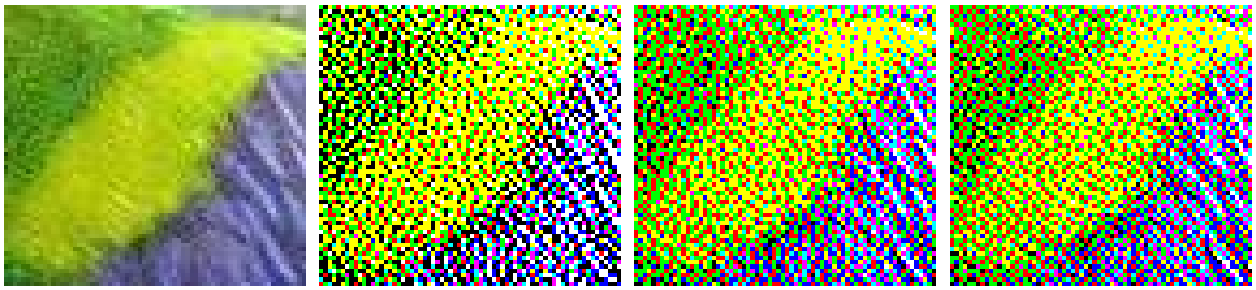
2.3.2 MBVQ-based Error Diffusion

Using method from [11]. Basic idea is to modify error diffusion on color image to make it more suitable for human eyes through minimize brilliance changes and reduce number of color used. They presented minimal brightness variation criterion to transfer color to minimize brightness variation while maintain average color by selection a color which minimize brightness change in tetrahedra which this pixel belongs to in RGB cube. Since human eyes are more sensitive to brightness change than color variation, and tend to average neighbor

(a) $bird.raw$ (b) MBVQ use DT $(0, 1)$ (c) MBVQ use DT $(1, 1)$

(d) MBVQ use L_1 norm $(0, 1)$ (e) MBVQ use L_1 norm $(1, 1)$

Figure 14



(a) $bird.raw$ (b) ED $(1, 1)$ (c) MBVQ use DT $(1, 1)$ (d) MBVQ use L_1 norm $(1, 1)$

Figure 15: crop from $bird.raw$ (neck part)

pixels as a low pass filter, this method can provide more vivid and pleased image to human than separable error diffusion.

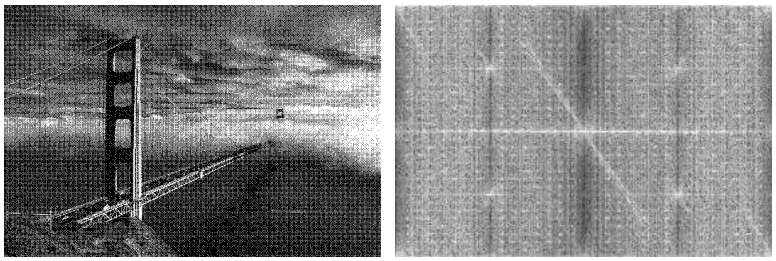
Using decision tree provided needs super long codes $Figure14(b, c)$. Even though L_p norm is equivalent to decision tree only inside RGB cube (but it would happen that RGB would get outside to cube when adding with error), for the sake of laziness, I tried L_1 norm ($Figure14(d, e)$, far fewer codes) to find closest vertex after deciding which tetrahedra a pixels belongs to as well. Result in this image seems almost the same.

Time complexity remains $O(N \times M \times 3)$ with slightly increase in coefficient and no more additional memory requirement .

From $Figure16$ which is the crop of $bird.raw$ from bird's neck region, it can be found, Comparing with separable error diffusion, MBVQ based error diffusion tends to use less black color in place where brilliance is high for example in yellow part using red rather than black make result less variance in brilliance while maintain average color, gives a more comfortable visual experience to human (more similar to raw image). Difference between using decision tree ans L_1 norm to decide the closet vertex is not obvious.

2.4 Improving Ideas

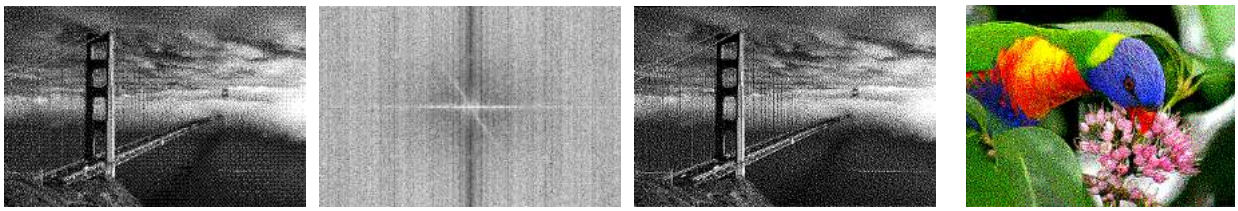
2.4.1 Noisy Index Matrix



(a) Noisy Index Matrix NI_{32} (b) DFT

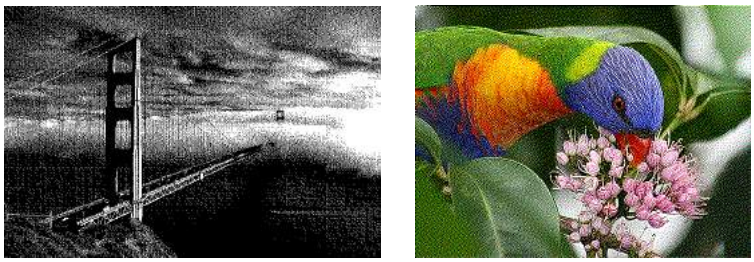
Figure 16: Result for Noisy Index Matrix and its DFT

Artifacts added to images when using index matrix or error diffusion using rhythmic sweeping method would result a poor image visual experience. One idea is to add some noise to threshold when deciding a pixel value $(0/255)$ This is by using parameter $noise$ in normal Dithering Matrix. $noise = 0$: no noise added, else add uniform noise, $NoisyIndexMatrix = IndexMatrix + U(-abs(noise), abs(noise))$. From $Figure16$, it can be found some of frequency are suppressed. Using blue noise might get better result.



(a) Mess Index Matrix MI_{32} (b) Mess Index Matrix's DFT (c) Mess up on ED $(1, 1)$ (d) Mess up on MBVQ $(1, 1)$

Figure 17: Result for Mess Up

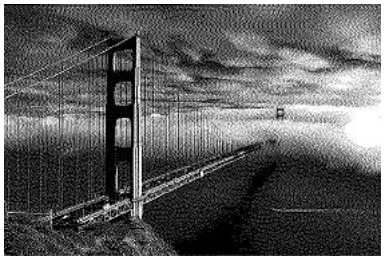


(a) Error Diffusion (b) Separable Error Diffusion

Figure 18: Error Diffusion using Hilbert Curve to Sweep

2.4.2 Mess Up

Another idea is to add randomness after get result from using index matrix or general sweeping methods. Let me call it $MessUp$, which get a better result and almost no artifacts ($Figure17$). It look much better than adding random noise to thresholding index matrix (Noisy Index Matrix above). Mess up in a small range of pixel would destroy artificial patterns while preserve as much details as it can.



(a) $threshold = 99$

(b) $threshold = 69$

(c) MBVQ-based $threshold = 59$

Figure 19: Error Diffusion$(1, 1)$with threshold on error

2.4.3 Change Way of Sweeping

Following Hilbert Cure to sweep the whole image, which helps to reduce artificial patterns. Result $Figure18$ seems pretty nice more details, higher contrast and more similar to original image.

2.4.4 Error Threshold

Another improvement idea is to restrict error (by using $threshold$, if $threshold < 0$ no threshold restrict) in a range $[-threshold, threshold]$ when using error diffusion or separate error diffusion $Figure19$. This modification leads to higher contrast and saturation. While MBVQ with threshold gives a more uniform increase on contrast make looks better than ordinary error diffusion.

References

[1] [Online]https://en.wikipedia.org/wiki/Grayscale#cite_note-4

[2] [Online]http://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm

[3] J. Canny, “A computational approach to edge detection,” IEEE Transactions on pattern analysis and machine intelligence, no. 6, pp. 679–698, 1986.

[4] [Online]https://en.wikipedia.org/wiki/Canny_edge_detector

[5] [Online]https://docs.opencv.org/3.1.0/da/d22/tutorial_py_canny.html

[6] P. Dollar and C. L. Zitnick, “Structured forests for fast edge1 detection,” in Proceedings of the IEEE International Conference on Computer Vision, 2013, pp. 1841–1848.

[7] [Online]https://pdollar.github.io/toolbox/

[8] [Online]https://github.com/pdollar/edges

[9] [Online]http://graphics.cs.cmu.edu/courses/16-824/2016_spring/slides/seg_1.pdf

[10] B. E. Bayer, “An optimum method for two-level rendition of continuous-tone pictures,” SPIE MILE- STONE SERIES MS, vol. 154, pp. 139–143, 1999

[11] D. Shaked, N. Arad, A. Fitzhugh, I. Sobel, "Color Diffusion: Error-Diffusion for Color Halftones", HP Labs Technical Report, HPL-96-128R1, 1996.

[12] Baqai F A , Lee J L H , Agar A U , et al. Digital color halftoning[J]. IEEE Signal Processing Magazine, 2005, 22(1):87-96.

Appendices

To evaluate result, I use the following code:

```
% suppose evaluate edge image named Pig_SE.jpg
E = imread('Pig_SE.jpg');
E = im2double(E);

% load ground truth
load('Pig.mat');

% some modification is made on edgesEvalImg function
% compare with original code in:
% https://github.com/pdollar/edges/blob/master/edgesEvalImg.m
% replace Line55 with:
% n=length(G);

% for evaluating on one single ground truth
% ex: the first groundTruth in following code:
[thrs1,cntR1,sumR1,cntP1,sumP1, V1] = edgesEvalImg( E, groundTruth(1));
% for evaluating on all ground truth:
[thrs,cntR,sumR,cntP,sumP, V] = edgesEvalImg( E, groundTruth);

% Then to get R, P, F, I use functions:
% function [R,P,F] = computeRPF(cntR,sumR,cntP,sumP)
% function [bstR,bstP,bstF,bstT] = findBestRPF(T,R,P)
% in Line116-119 and Line121-131 from
% https://github.com/pdollar/edges/blob/master/edgesEvalDir.m
% ex: get R, P, F according to all ground truth:
[R,P,F]=computeRPF(cntR,sumR,cntP,sumP);
[bstR,bstP,bstF,bstT]=findBestRPF(thrs,R,P);
% R, P, F represented in Table1 and Table2 are:
% bstR, bstP, bstF
```