EE 569 Homework 2 Write Up

Armin Bazarjani

February 22th, 2021

1 Edge Detection

1.1 Motivation

Edge detection can be generally described as the process of finding the locations in an image, where the image has a sharp change in contrast or intensity. More formally, we can refer to these as discontinuities. We are generally interested in the process of determining the edges within an image because that can lend itself to image understanding. We can extract the structure and pose of objects, discontinuities and differences in depth within an image, etc. While these applications themselves are incredibly important in their own right, they also serve as the basis for more sophisticated algorithms that can parse much deeper abstractions and meaning from a single image. This is crucial not only in computer vision applications, but also in the development of systems with general intelligence.

Although the edge detection is still an open problem, the advancements that classical computer vision have made are very much near the state of the art algorithms that we have seen through deep learning. For this problem, we implemented three well known classical approaches.

1.2 Approach and Procedures

1.2.1 Sobel Edge Detector

The Sobel edge detector is the simplest of the three that we were tasked with actualizing. It relies solely on using the first derivative to detect discontinuities, and thus edges in the image. The intuition is that points that lie on an edge are able to be detected by the local maxima or minima. Thus, we are able to compute a value of edge strength by computing the magnitudes of the gradient in the x and y directions, those being M_x and M_y. Afterwards, we can combine the magnitudes of the partial derivatives to get the total gradient magnitude $M_{(x,y)} = \sqrt{M_x^2 + M_y^2}$. Afterwards, we can threshold the total magnitude at each point to construct our gradient map. The threshold we choose is essentially the sensitivity of our Sobel edge detector with a lower threshold allowing

most things and a higher threshold allowing fewer things.

Because derivatives are a continuous function, we can easily approximate them in the discrete domain with the following:

$$M_x = \frac{\partial f}{\partial x} = f(x+1) - f(x)$$

$$M_y = \frac{\partial f}{\partial y} = f(y+1) - f(y)$$

Also, as derivatives are linear and shift invariant, we are able to implement them using convolutions. To do this, many filters have been proposed. One of the most popular being the Sobel filter where:

$$M_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, and\ M_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

The intuition behind the choice in filter parameters is that we want add more weight to the neighbors that are closer the the center.

To implement the Sobel filter is rather straightforward. I first zero padded the input image with a padding size of ($P = 1$) to accommodate the 3x3 Sobel filter. Then, I used the standard convolution operations to generate a magnitude map in the x and y directions. Afterwards, I combine the two magnitudes using the equation I showed above and thresholded the output.

1.2.2 Canny Edge Detector

The canny edge detection algorithm can be thought of as an extension of the traditional first order derivative edge detection algorithms that rely on calculating the magnitude of each pixel's gradient. Traditionally, our first order edge detection method will usually (1) filter the image, (2) get the gradient magnitude each pixel in both the x and y directions, (3) combine the two magnitudes, (4) threshold the combined magnitude to construct the final edge map.

Canny edge detection builds through the addition of two new steps to the edge filtering pipeline that I previously described. These two new steps being non-maximum supression and double thresholding. The details of these two steps can be found in the discussion (section 1.4) below.

1.2.3 Structured Edge

The details on the implementation of the structured edge algorithm are described below, instead I will give a very high level overview. Essentially, the structured edge algorithm attempts to build on the previous success of Sketch Tokens and structured learning through the use of random forests (more on this

later as well).

To implement this I used the author's (Piotr Dollar) code from GitHub. There is a pre-trained model on the BDS-500 dataset which I used on the desired images. I kept the default parameters the same as it came out of the box with the reccomended parameters from the author. The only change I made was to enable non-maximum suppression and increasing the number of evaluation trees (although I didn't notice any significant changes with changing the second option). I also added some extra code to threshold the image and construct a binary map. The threshold I arrived at was done through trial and error until I converged on a threshold that was visually pleasing for me.

1.2.4 Precision, Recall, and F-Score

In order to gauge how objectively good the different edge detection algorithms are we cannot rely on our visual inspection (although it is a decent initial measure as a sanity check). Instead we need to rely on a more mathematical approach. Luckily the BDS-500 dataset also has ground truth edge labels from 5 different human labelers. Because human's have different biases in what constitutes an edge, we usually take the average to give a more robust result based on our metric.

The metric we use for evaluation is precision, recall, and F-Score. This is an intuitive approach as we can compare the binary edge mappings using the terminology of true positive, false positive, true negative, and false negative. Another intuitive reason why we might use this metric instead of something like accuracy is if we had a small object. With a small object we can get a high accuracy by labeling every point as a non-edge. However, this would result in a 0 for precision and recall as we are not getting any true positive values.

Precision gives us the proportion of data points that were relevant given that our model told us they were relevant data points. Recall gives the model's ability to find all of the relevant data points.

The equations for precision and recall are as follows:

$$precision = \frac{\#True\ Positives}{\#True\ Positives + \#False\ Positives}$$

$$recall = \frac{\#True\ Positives}{\#True\ Positives + \#False\ Negatives}$$

From precision and recall we can also calculate the F score. The F score is simply a harmonic mean of precision and recall. The reason we use the harmonic mean is because we want to punish having extreme values. This will

point out any glaring inconsistencies in the classifier.

$$F\_score = 2 * \frac{precision * recall}{precision + recall}$$

Also I think it is worthwhile to note that I took some creative freedom when conducting the performance evaluation on the Canny edge detector. The reason is because the built in MATLAB Canny edge detector does not allow for the option of outputting a probability map (nor does the algorithm really allow it). So, for part(1) of the performance evaluation I decided to calculate the precision, recall, and f measure without setting any high and low thresholds on the Canny edge detector. For part(2) of the performance evaluation I took all the different thresholds I tried (11 total) and created 55 different $\binom{11}{2}$) high low threshold pairs to test on the Canny edge detector.

1.3 Experimental Results

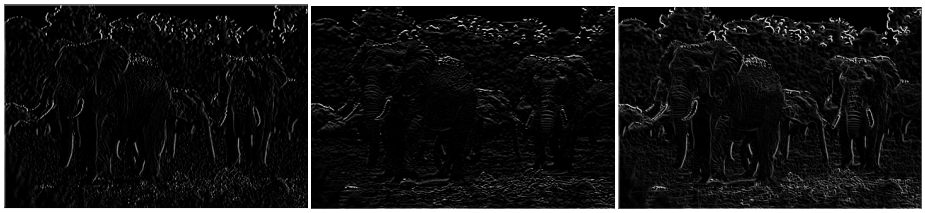
The original elephant and ski person images are as follows:

(a) Elephant (b) Ski Person

Figure 1: Original images we performed edge detection on

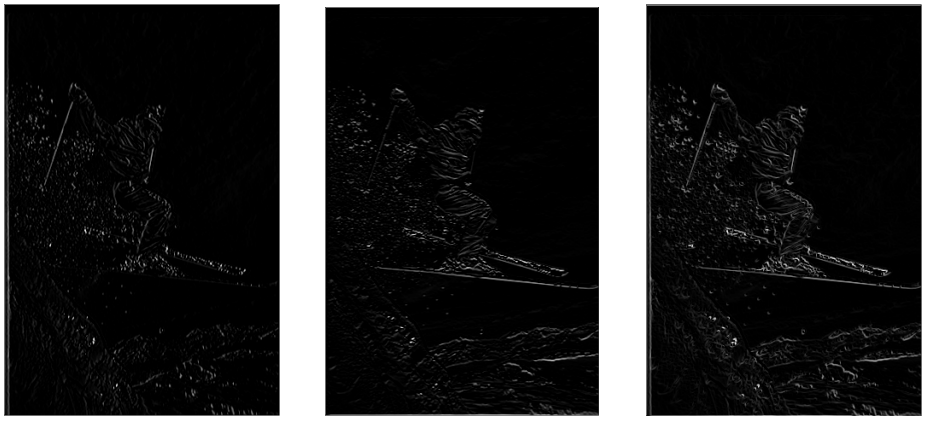
Next, I will show the results of Sobel edge detection as follows. The first set of images will show the normalized X and Y gradients as well as the magnitude map from the combination of these gradients on both images. The second set of

images will show thresholded versions of the magnitude map that received the best F measures on each image respectively.



(a) X Grad (b) Y Grad (c) Combined Grad

Figure 2: X, Y, and combined gradient maps for elephant



(a) X Grad (b) Y Grad (c) Combined Grad

Figure 3: X, Y, and combined gradient maps on ski person

(a) Elephant Thresholded (b) Ski Thresholded

Figure 4: Sobel Elephant and Ski Person Thresholded with value that achieves highest F-Score (0.456 and 0.467 respectively)

Now, I will show the results of Canny edge detection. The first set of images will be the elephant and ski person with no low or high thresholds applied. The second set of images will be with the optimal low and high thresholds I found using the F measure.

(a) Elephant no thresholding (b) Ski no thresholding

Figure 5: Canny Elephant and Ski Person with no thresholding

(a) Elephant with thresholding (b) Ski with thresholding

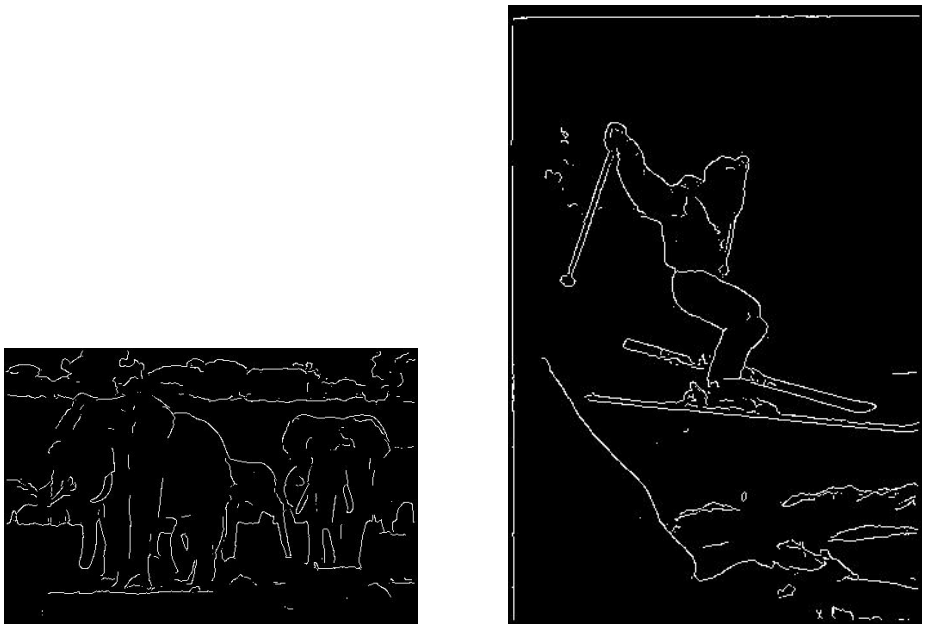
Figure 6: Canny Elephant and Ski Person with high and low thresholds applied

For the elephant image the high and low thresholds are: (low=0.18, high=0.364) and for the ski person image the high and low thresholds are: (low=0.18, high=0.364). The F-Scores are 0.615 and 0.6591 respectively.

(a) Elephant no thresholding

(b) Ski no thresholding

Figure 7: Structured Edge with no thresholding



(a) Elephant no thresholding (b) Ski no thresholding

Figure 8: Structred Edge with optimal thresholding

I found that the best threshold value for both the elephant and ski person images when using structured edge was 0.18. They each got an F-Sscore of 0.6926 and 0.7282 respectively.

1.4 Discussion

1.4.1 part (a) : Sobel

Q(a-1) - Please refer above to section "1.3 Experimental Results to see the X and Y gradients"

Q(a-2) - Please refer above to section "1.3 Experimental Results" to see the normalized gradient magnitude map.

Q(a-3) - Please refer above to section "1.3 Experimental Results" to see the binary thresholded image.

1.4.2 part (b) : Canny

Q(b-1) - Non-maximum suppression works by taking the gradient direction and comparing the current pixel magnitude with that of the magnitudes of the pixels along the gradient direction. The intuition is that non-maximum suppression is

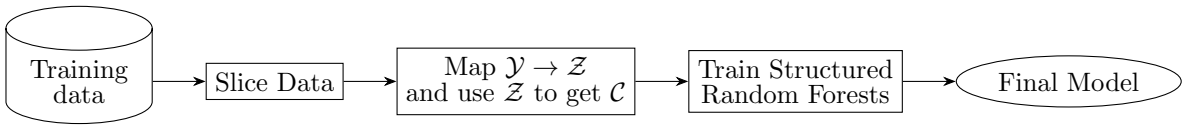
used to "thin" the edges, only allowing what we would hope to be the true edge location to pass through. This naturally works because the gradient direction is normal to the edge, thus if we compare magnitude with those others that are normal to the edge, the one with the highest value should be closer to the true edge location and we can get rid of the other ones.

Q(b-2) - The high and low thresholds are used to define a "true" edge vs. a "non" edge. If the gradient value is above the high threshold, we can be certain that it is an edge, conversely if it is below the low threshold we can be certain that it is not an edge. Now, if a value falls in between the high and low thresholds, it is only considered an edge if it is connected to a "true" edge value that lies above the high threshold. This is used during double thresholding to filter out the remaining spurious edge responses that could be due to noise or color variation. We filter out weak gradient values (non edge) unless they are connected to strong gradient values (true edge).

Q(b-3) - You can observe a few samples of different constructed edge maps above in section "1.3 Experimental Results". As you can see, the most promising results came from lower end of the thresholds of the Canny edge detector. I tried 55 different low-high combinations and found the best for ski to be threshold(low=0.09, high=0.36) which achieved an f measure score of 0.62. For elephant I found threshold(low=0.18, high=0.36) which achieved an f measure of 0.6.

1.4.3 (part (c) : Structured Edge

Q(c-1) - The structured edge algorithm is both intricate and calculated in it's design, yet simple and pragmatic with its procedure. The overview is to use a random forest classifier trained on structured labels to create a robust edge detector. The training phase is as follows: First, we take the training images and create $\mathcal{X} : 32x32$ slices with $\mathcal{Y} : 16x16$ structured segmentation masks. Next, we feed these image slices into a mapping function. For many structured output spaces, computing similarity is not well defined. Thus, we need the mapping function to map $\mathcal{Y} \rightarrow \mathcal{Z}$ where $\mathcal{Z}$ is a 32640 long binary vector of pixel pair information. So, with our new vector space, we can easily calculate the distance and hence similarity over $\mathcal{Y}$. First the mapping, samples only $m = 256$ dimensions of $\mathcal{Z}$ as we only need an approximate distance. On top of that we further reduce the dimensions to 5 through PCA. After PCA, we can obtain class information through K-means with $K = 2$. After mapping, we now have patches with feature labels (where $labels \in \mathcal{C}$) which we can use to train a structured random forest.



Training Structured Random Forest

To use the trained model, we slice the input image in the same way that we sliced our training images. We can then feed each slice into the trained structured random forest which will ensemble the outputs of all the different decision trees to construct a soft edge response.

Q(c-2) - Decision trees work by iteratively selecting the best feature to split on. These features are chosen through a process of maximizing information gain. We can quantify information gain by using entropy, which measures the average level of information in a variable's possible outcomes. The lower the entropy, the lower the uncertainty.

$$Entropy = \sum_i -p_i log_2 p_i$$

Where p_i is the probability of class i. Now we can calculate the information gain if we decide to split on a feature by:

$$IG(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{S} Entropy(S_v)$$

Where S is our collection, $Values(A)$ is the subset of all possible values for feature A, and S_v is the subset of S for which feature A has value v. Here we can see that the first term is the total entropy of our original collection and the second term is the expected value of entropy after S is split up based on feature A. We choose to split on the feature that maximizes this function.

Note that after you split based on a feature, you have to update the collection, S, with the remaining values that couldn't be categorized by the leaf nodes and continue the process again. You would continue this process until each sample in S has been categorized.

As expected, decision trees are very prone to overfitting on the training data. To combat this, an idea known as random forests was proposed. Random forests work to prevent overfitting by constructing multiple decision trees on various random subsets of the training data. The output at test time is the mean or average prediction of all of the different decision trees that constitute the random forest. This process of randomized feature subset selection helps reduce our error due to variance. The fact that we are constructing many of these decision trees means that we will most likely come across each feature in each sample which reduces our error due to bias. It is worthwhile to note that this randomization can come before the construction of the tree, where the entire

decision tree is built on a random subsample of the data. Or, a more popular method, is to randomly subsample the features and splits used to train each node of the decision tree.

1.4.4 (part (d) : Performance Evaluation

Q(d-1) - The precision and recall for each ground truth, their means, and the F measure from the means for both images can be found in the tables below.

Sobel Elephant			
Elephant GT	Precision (P)	Recall (R)	F-Score (F)
GT-1	0.5855	0.2245	0.325
GT-2	0.4347	0.2044	0.278
GT-3	0.6358	0.1883	0.291
GT-4	0.3026	0.1498	0.200
GT-5	0.4273	0.1799	0.253
Mean	**0.477**	**0.1894**	**0.271**

Sobel Ski Person			
Ski GT	Precision (P)	Recall (R)	F-Score (F)
GT-1	0.4855	0.1277	0.202
GT-2	0.4870	0.1175	0.189
GT-3	0.6329	0.1145	0.194
GT-4	0.4509	0.1264	0.197
GT-5	0.5061	0.1226	0.197
Mean	**0.5061**	**0.1226**	**0.197**

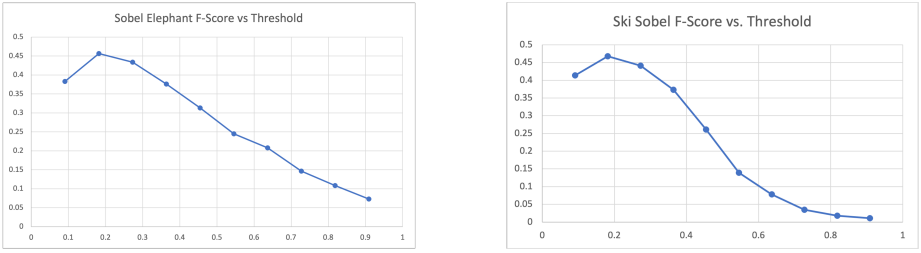
Canny Elephant			
Elephant GT	Precision (P)	Recall (R)	F-Score (F)
GT-1	0.3030	0.9662	0.461
GT-2	0.2486	0.9717	0.396
GT-3	0.3869	0.9526	0.550
GT-4	0.2289	0.9417	0.368
GT-5	0.2695	0.9433	0.419
Mean	**0.2874**	**0.9551**	**0.442**

Canny Ski Person			
Ski GT	Precision (P)	Recall (R)	F-Score (F)
GT-1	0.2134	0.9848	0.351
GT-2	0.2333	0.9878	0.377
GT-3	0.2960	0.9401	0.450
GT-4	0.1997	0.9826	0.332
GT-5	0.2114	0.9927	0.349
Mean	**0.2308**	**0.9776**	**0.373**

SE Elephant			
Elephant GT	Precision (P)	Recall (R)	F-Score (F)
GT-1	0.9604	0.1009	0.183
GT-2	0.9550	0.1230	0.218
GT-3	0.9604	0.0779	0.144
GT-4	0.6025	0.0817	0.144
GT-5	0.8939	0.1031	0.185
Mean	**0.8745**	**0.0973**	**0.175**

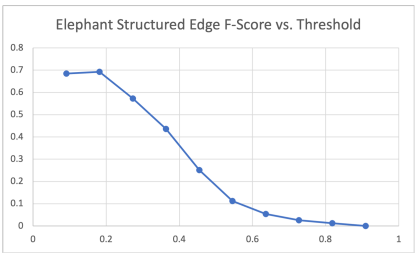
SE Ski Person			
Ski GT	Precision (P)	Recall (R)	F-Score (F)
GT-1	0.6990	0.3290	0.447
GT-2	0.6433	0.2778	0.388
GT-3	0.7030	0.2278	0.344
GT-4	0.6433	0.3229	0.430
GT-5	0.6990	0.3349	0.453
Mean	**0.6775**	**0.2985**	**0.414**

Q(d-2) - The plots of F measure against the different thresholds are as follows.

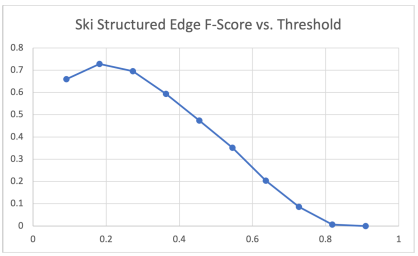


(a) Elephant (b) Ski Person

Figure 9: F-Score vs. Threshold plot for Sobel

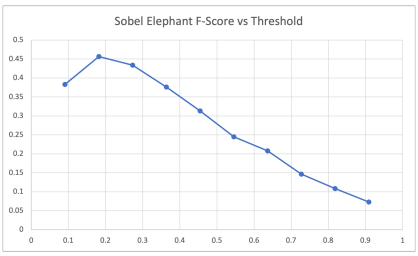


(a) Elephant

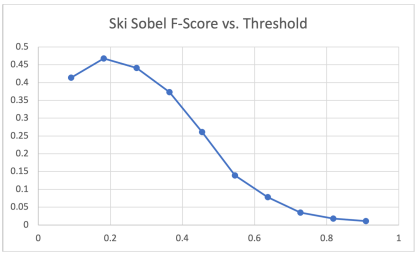


(b) Ski Person

Figure 10: F-Score vs. Threshold plot for Structured Edge

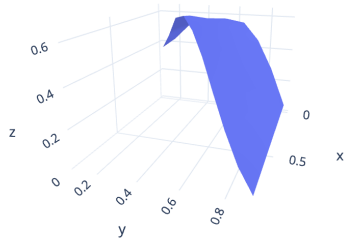


(a) Elephant

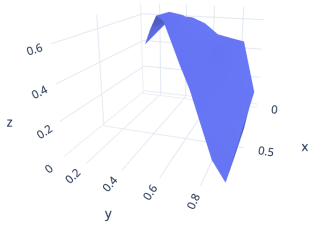


(b) Ski Person

Figure 11: F-Score vs. Threshold plot for Sobel



(a) Elephant



(b) Ski Person

Figure 12: Threshold Low (X) vs. Threshold High (Y) vs. F-Score (Z)

From observing the plots it is clear that the structured edge method outperformed Sobel and very nearly outperformed Canny once we apply thresholding. Interestingly the thresholding range that I found to achieve the best results for both Sobel and Structured Edge was with a threshold of 0.18.

Q(d-3) - As we have seen the ski person image yields a higher F measure on average. This is most likely for two reasons. The first is that in the elephant image there is a lot going on both in the foreground and in the background. We can see in the labels that many of the ground truth labelers decided that some of the things going on in the background weren't worth labeling. Unfortunately, many edge detection methods will pick up on the background information. Conversely, in the ski person image, every single edge detection method will pick up the obvious object in the center being the ski person. The second reason could be that within the elephant image there are also some edges (tusks, trunk, etc.) that the labelers don't choose to label, but they are picked up by edge detection algorithms.

Q(d-4) - As I previously stated the F measure is a harmonic mean of precision and recall, meaning it will punish extreme values for precision and recall. It is not possible to get a high F measure if one is significantly higher or lower than the other. A very obvious example of this would be if precision was 0 and recall was 1. If we used the standard mean we would get a score of 0.5, however the F measure would be 0.

If the sum of precision and recall is a constant, say $P+M=A$, then we can say that $P=B$ and thus $R=A-B$. Then $F=2*\frac{B(A-B)}{B+(A-B)}=2*\frac{BA-B^2}{A}=\frac{m}{2}=2B-\frac{2B^2}{A}$. To show that this is the maximum we need to take the derivative of F with respect to B, set to 0, and solve for B. Doing this we get, $\frac{\partial F}{\partial B}=2-\frac{4B}{A}$. And thus $B=\frac{A}{2}$. So, we can see that in order to maximize the F measure both precision and recall must have the same values.

2 Digital Half-toning

2.1 Motivation

We are concerned with the process of half-toning an image because we want a good way to be able to represent the complexity within an image, with a seemingly continuous representation, through dots. The difference being that now instead of having different intensities at our disposal we have a single one, that being either on or off. Now, if we are able to use different colors then we will have a wider range of values at our disposal. But, in the end it is the same principle. Either the pixel location in our output is on (or has a specific color) or it is off, there is no in between.

This is a very important problem when it comes to printing. How can we print out an image with a single color of ink at our disposal? The half-toning techniques allow us to express a color or multi-level intensity image with a limited, or even binary, set of colors. Interestingly half-toning provides pretty good results as we will see in the coming sections. Intuitively, we can think that even though when we zoom into the image it might look patchy and discontinuous, when

we zoom out of the pixel level representation, our eyes will naturally average everything out and we are still able to represent varying colors and intensities.

2.2 Approach and Procedures

2.2.1 Dithering

Dithering, in the context of image processing, is a method of compressing an image further. For our purposes, we converted a grayscale image into black and white. However, this is done in a unique way so as to maintain some semblance of the complexity in the original signal. A more intuitive definition of dithering would be that we are attempting to simulate depth of color in an image with a very limited color palette. The idea is that we can group pixels together in specific ways so as to create an illusion of visual depth when taking the image in as an overall whole.

There are various ways of dithering a grayscale to binary image. The first two naive methods we used had to do with applying a threshold to every single pixel. If we are above the threshold the pixel gets the maximum value (255 in our case), and if we are below the threshold the pixel gets set to the minimum value (0). The threshold can be both fixed and random (changes for every pixel).

A more sophisticated approach is to use a dithering matrix. This matrix stores patterns of thresholds and are power-of-2 matrices. By turning on the pixels in a specific order, hence the use of a predefined matrix, we can create the illusion of continuous change in color even though we are only using black and white pixels. We use a Bayer matrix, specifically the ones of size 2x2, 8x8, and 32x32. The larger the matrix, the more we can simulate (to a certain degree) the color variation.

The Bayer matrix can be defined as such:

$$I_2(i,j) = \begin{bmatrix} 1 & 2 \\ 3 & 0 \end{bmatrix}$$

We can recursively build upon this initial matrix to get:

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

Finally, we can threshold using the matrix:

$$T(i,j) = \frac{I(i,j)+0.5}{N^2} * 255$$

$$G(i,j) = \begin{cases} 255 & 0 \leq F(i,j) \leq T(i \ mod \ N, j \ mod \ N) \\ 0 & T(i \ mod \ N, j \ mod \ N) \leq F(i,j) \leq 255 \end{cases}$$

The implementation of the thresholds was very simple. The dithering matrix was more interesting as its construction at a higher level depends on its values at a previous level. To implement this, I decided to use vectors and recursion. I initially passed in the original starting 2x2 matrix, an empty nxn matrix that I wanted to fill, the previous size, and the final size. I then would recursively construct the next size matrix until $prev\_size == final\_size$. Once this condition was met, I returned the final matrix.

2.2.2 Error Diffusion

The basic idea of error diffusion is to make the pixels after a previously thresholded pixel compensate for the error in the original pixel's thresholding. An intuitive example would be if we decided to threshold a pixel prematurely and set it to 0 when it should have been 255, this error will be propagated (through the use of one of the error diffusion matrices) to the pixel's neighbor values in order to lighten up the neighbors a bit. The hope is that by lightening up the neighbors we can offset the error in the original pixel. This is then done through the image, looping through it in a serpentine fashion, until we have theoretically propagated the error from every pixel to all of its relevant neighbors.

Using a larger error diffusion matrix only means that you can propagate the error further along in an image to more pixels. Depending on how you are looping through the image, you might have to mirror the matrix in order to match the direction of motion.

For example, when using the serpentine method you would this matrix on even rows (note that the matrix shown is for the Floyd-Steinberg method):

$$FS\_normal = \frac{1}{16}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 7 \\ 3 & 5 & 1 \end{bmatrix}$$

and this matrix on odd rows

$$FS\_mirrored = \frac{1}{16}\begin{bmatrix} 0 & 0 & 0 \\ 7 & 0 & 0 \\ 1 & 5 & 3 \end{bmatrix}$$

To implement this I decided to zero pad the image based on the filter size so the output dimensions would match the input dimensions of the image. I also used serpentine scanning as described above.

2.3 Experimental Results

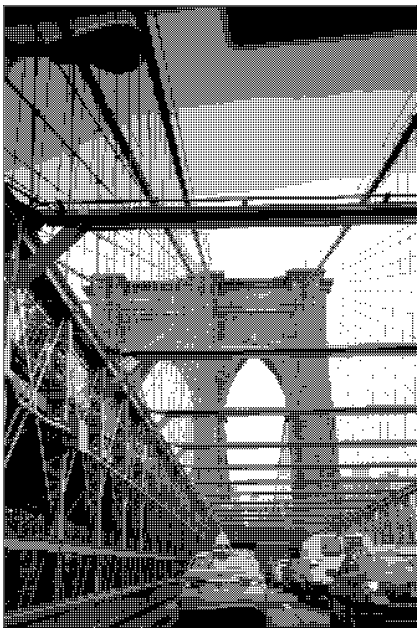
First, I will display the original image along with fixed and random thresholding. Next, I will show the results of using different size dithering matrices. Finally, I will show the three different error diffusion methods.

(a) Original Bridge

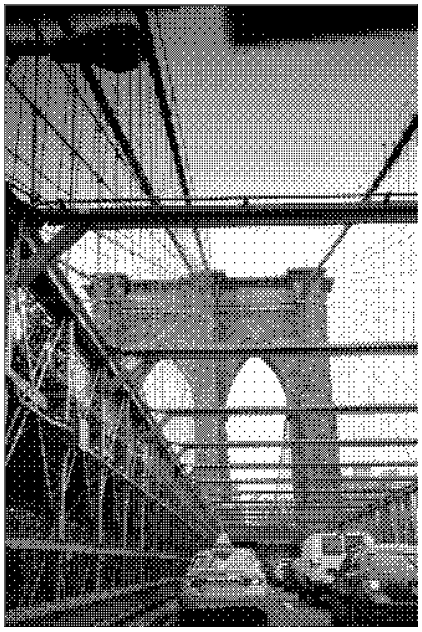
(b) Fixed Thresholding

(c) Random Thresholding

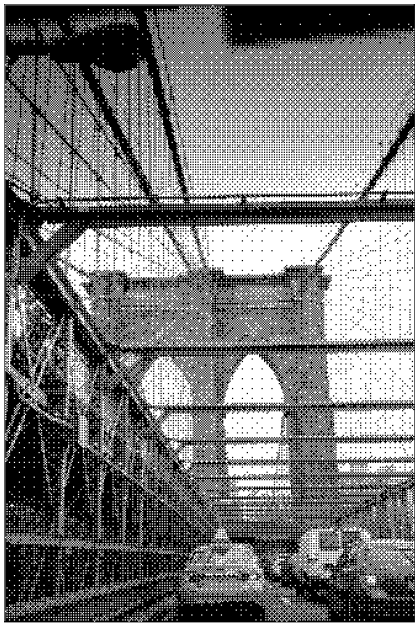
Figure 13: Fixed and Random Thresholding on Bridge



(a) I_2 Dithering

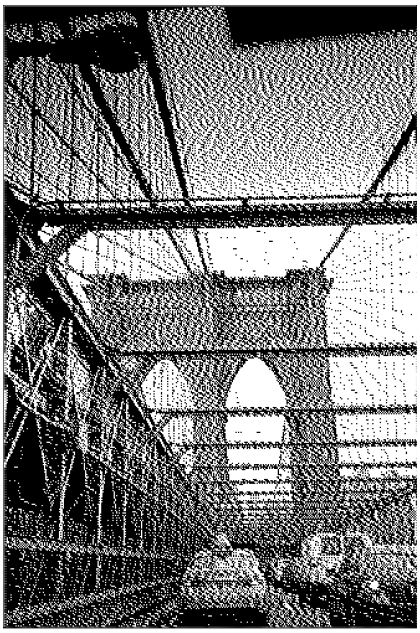


(b) I_8 Dithering

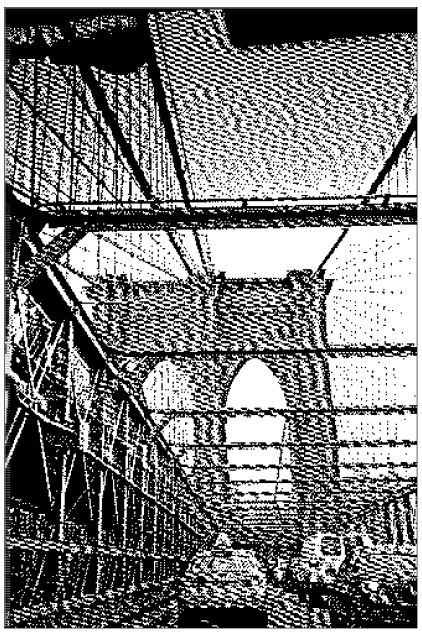


(c) I_{32} Dithering

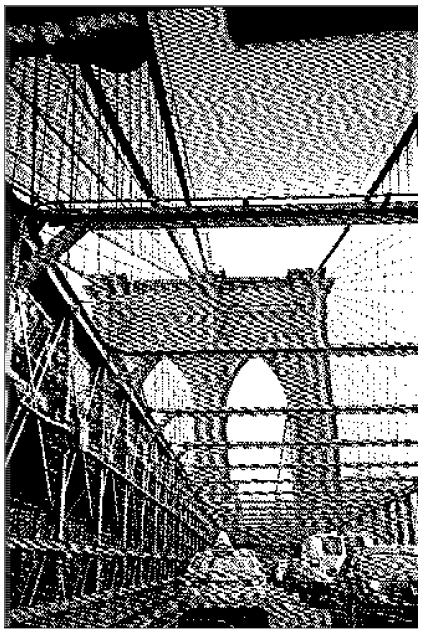
Figure 14: Dithering Matrix on Bridge



(a) Floyd-Steinberg



(b) Jarvis, Judice, and Ninke



(c) Stucki

Figure 15: Error Diffusion on Bridge

2.4 Discussion

2.4.1 part (a) : Dithering

Q(a-1) - Please view section "2.3 Experimental Results" for the results of the fixed thresholding. As you can see, the results are actually not too bad. However, the image certainly lakes depth in the sense that it is very obviously black and white. We will see with the upcoming methods how this can be handled to give the illusion of differing intensities.

Q(a-2) - Please view section "2.3 Experimental Results" for the results of the random thresholding. The results from random thresholding are quite interesting. There seems to be some presence of the input images edges and intensity changes. However, because we are randomly turning on and off different pixel values the resulting image is incredibly noisy. Thus, this is also not a very good approach.

Q(a-3) - Please view section "2.3 Experimental Results" for the results of using different size dithering matrices. As we can see, using a smaller dithering matrix doesn't allow as much of the error to be propagated through the image, so using a larger dithering matrix of size 8x8 or 32x32 actually produced slightly better results. It is worthwhile to note that there is very little and almost no significant change between the results of the I_8 and I_{32} dithering matrices.

2.4.2 part (b) : Error Diffusion

Please find the results of the three error diffusion different methods [Floyd-Steinberg (FS); Jarvis, Judice, Ninke (JJN); and Stucki]above in section "2.3 Experimental Results". Upon inspecting the results we can see that error diffusion is a better approach to half-toning than the dithering methods. Although, to be fair, it is not incredibly noticeable once we get the the larger dithering matrices.

Comparing the three different methods to one another, you are again hard pressed to find an obvious favorite. All three error diffusion methods look somewhat similar to me and produced good results. I find that the ones that use larger matrices (JJN and Stucki) barely edge out the one that used a smaller matrix (FS). I believe the reason for this is that because we can pass the error to more neighbors, we are not only expanding the reach of a pixel, but we are also downplaying its affect on flipping its immediate neighbors which can lead to more salt and pepper noise.

To improve results, I would be interested to see what a slightly larger matrix could produce, say a 7x7. Intuitively, I think that allowing the pixel value to reach a few more neighbors should do more good than harm as it is more information to work with. Of course the weighting for each value will have to be adjusted accordingly. I also think that using a more sophisticated scanning or-

der, for example Hilbert scanning. The rational behind Hilbert scanning would be because we can also pass the error to the top row of the image as well as it is a more natural way to walk through an image rather than the regimented row by row approach of serpentine scanning.

3 Color Half-Toning with Error Diffusion

3.1 Motivation

The motivation for color half-toning is the same as described above in section "2.1 Motivation". As a quick recap, I will just say that when we are printing images, we are limited with amount of colors at our disposal and it can be very costly to mix and match colors to the minute detail that we would need to produce the image verbatim. Thus, we can easily expand the half-toning techniques described above across each color channel and bring the results back together. There are also some more sophisticated methods to deal with color images as a whole, the one we explored was MBVQ-Based Error Diffusion.

3.2 Approach and Procedures

3.2.1 Separable Error Diffusion

Separable error diffusion is really just an extension of the error diffusion method described above in section "2.2.2 Error Diffusion". Except we first convert our input image from the RGB color space to the CMY color space.

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Once in the CMY color space, we applied the Floyd-Steinberg error diffusion algorithm across each channel C, M, and Y. Afterwards, we convert back to the RGB space to produce the output image.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix}$$

3.2.2 MBVQ-Based Error Diffusion

The MBVQ method attempts to overcome the shortcomings of separable error diffusion by taking into consideration each channel of a pixel rather than dealing with each one on its own.

The motivation comes from the fact that if we take the entire color channel into account, we have 8 different colors.

$$R = (1, 0, 1),\ G = (0, 1, 1),\ B = (1, 1, 0),\ K = (1, 1, 1)$$

$$Y = (0, 0, 1),\ C = (0, 1, 0),\ M = (1, 0, 0),\ W = (0, 0, 0)$$

The minimum brightness value criterion states that the RGB cube can be constructed using these 8 basic colors. However, in practice, we see that we actually don't need more than 4. So, we can approximate any color with 4 colors instead of 8, makings things slightly more computationally efficient.

I implemented MBVQ by first writing a function to determine which quadrant we are in. The function takes as input the RGB values of the current pixel. Then I use this quadrant, along with the RGB values to find the nearest vertex. This nearest vertex will be what we set our output image to and from what we will need to calculate the error to be diffused (this is what we replace the thresholding step with in traditional error diffusion methods). The next steps are the same as in the other error diffusion methods, the only thing that has changed is that we calulate our error with respect to the nearest vertex that we found based off our MBVQ quadruple.

A useful addition as to why this is also better can be understood through the intuition of why we use minimum brightness variation quadrants (MBVQ's) to begin with. The assumption is that in small regions of an image, the brightness is not going to change too much. As has been alluded too, the brightness variation in small regions is reduced and thus visual quality will be improved.

3.3 Experimental Results

The results of color half-toning can be seen as follows.

(a) Original Fish

(b) Separable Color Diffusion

(c) MBVQ-Based

Figure 16: Color Half-Toning on Fish Image

3.4 Discussion

3.4.1 3-1 Separable Error Diffusion

Please refer above to section "3.3 Experimental Results" to see the resulting color half-toned image. The results look good, although it is quite obvious that there is some granularity that is associated with this method.

The main shortcoming of this method would have to be the fact that we are dealing with each color channel separately. This means we are loosing a lot of information about how the different channels are actually interacting with one another.

3.4.2 3-2 MBVQ-Based Error Diffusion

Q(3-2-1) - I believe I answered this question satisfactorily in section "3.2.2 MBVQ-Based Error Diffusion". Just to restate, the main difference is that the MBVQ method cleverly takes into account the entire channel's information rather than operating on each channel separately.

Q(3-2-2) - The results of each can be seen above in section "3.3 Experimental Results". Comparing the two methods visually, we can see that the MBVQ method produces favorable results. We can also see that the MBVQ method

does, in fact, have less bright regions too and is overall a more visually appealing color half-toning method.