JPEG Image Compression

Math 56

Matt Marcus

June 1, 2014

1 Introduction

In this paper I will discuss JPEG image compression. I'll begin with a discussion of the algorithm, as well as the mathematics and logic that underlie it. I will conclude by showing experimental results derived from code I wrote to test the JPEG algorithm.

2 What is JPEG Compression

JPEG, which stands for Joint Photographic Experts Group (the name of the committee that created the JPEG standard) is a lossy compression algorithm for images. A lossy compression scheme is a way to inexactly represent the data in the image, such that less memory is used yet the data appears to be very similar. This is why JPEG images will look almost the same as the original images they were derived from most of the time, unless the quality is reduced significantly, in which case there will be visible differences. The JPEG algorithm takes advantage of the fact that humans can't see colors at high frequencies. These high frequencies are the data points in the image that are eliminated during the compression. JPEG compression also works best on images with smooth color transitions, which will make sense when I explain below how the algorithm works.

3 JPEG Algorithm

The algorithm behind JPEG is relatively straightforward and can be explained through the following steps:

1. Take an image and divide it up into 8-pixel by 8-pixel blocks. If the image cannot be divided into 8-by-8 blocks, then you can add in empty pixels around the edges, essentially zero-padding the image.
2. For each 8-by-8 block, get image data such that you have values to represent the color at each pixel.
3. Take the Discrete Cosine Transform (DCT) of each 8-by-8 block.
4. After taking the DCT of a block, matrix multiply the block by a mask that will zero out certain values from the DCT matrix.
5. Finally, to get the data for the compressed image, take the inverse DCT of each block. All these blocks are combined back into an image of the same size as the original.

As it may be unclear why these steps result in a compressed image, I'll now explain the mathematics and the logic behind the algorithm.

3.1 Discrete Cosine Transform

In class we studied the Discrete Fourier Transform, which is a way to map a function to the frequency domain by sampling a function over a periodic interval. The DCT is similar, except it only uses the cosine function, therefore not interacting with complex numbers at all. There are a few variations of the DCT, depending on whether the right boundary of the interval is even or odd, and around what point the function is even. The DCT that is most frequently used, and that is used here, is called the DCT-II, which is even at its left boundary at a point halfway between its first point and the point before that. At the right boundary, you have an even extension of the function, which creates a continuous function across the boundary. This causes the DCT coefficients to decay even faster, which is good for image compression, since it allows us to approximate the image data in as few terms as possible. The DST wouldn't have this feature since it's discontinuous across boundaries and thus has slower decaying coefficients. [1, 2]

From deriving the DCT, you'll see that the values of the DCT are half that of the DFT[5], so that you can represent the relationship as $f_m = 2\sum_{j=0}^{N-1} f_j cos(\frac{m\pi}{2}(j+\frac{1}{2}))$, where f_m is the DFT coefficient and the sum on the right is the values of the DCT. Therefore, the values of the DCT are just:
$g_m = \sum_{j=0}^{N-1} f_j cos(\frac{m\pi}{2}(j+\frac{1}{2}))$.

When you take the DCT, the values with the lowest frequency will cluster around the upper-left corner of the DCT matrix. To understand this, we can look at the DFT matrix. In the upper-left corner, the value is ω^0. In the bottom-right, you would have ω^{-N^2}, if the matrix was NxN. Since with the DCT we're only using cosine, the coefficients in the DCT matrix will have a higher frequency when the cosine term's input is larger. This will come into play as we work to cancel the high frequencies in the image.

3.2 Canceling High Frequencies

As discussed before, humans are unable to see aspects of an image that are at really high frequencies. Since taking the DCT allows us to isolate where these high frequencies are, we can take advantage of this in choosing which values to preserve. By multiplying the DCT matrix by some mask, we can zero out elements of the matrix, thereby freeing the memory that had been representing those values. When we're working with 8x8 blocks, an example mask could look like:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

This matrix will only preserve those values at the lowest frequencies up to a certain point, which is exactly what we'd want [3].

3.3 Saving Memory

It is important to understand exactly how this process saves memory. The idea is underlied by the fact that representing numbers requires memory (specifically 1 byte per number). When each pixel in an image has a value to represent it, this requires memory. By zeroing out elements in the DCT, however, we're freeing memory. So the amount of memory used in the image can be determined by calculating the number of non-zero terms in the DCT matrix. This will give you the number of bytes required to store an image. To convert to to kilobytes, you would divide this value by 1024, the number of bytes in a kilobyte.

3.4 Why 8x8 Blocks

Another key aspect to the JPEG algorithm is the decision to perform these operations on 8x8 blocks. These dimensions may seem somewhat arbitrary, and that is in part because they are. However, there are also a few reasons to support this decision.

First, if the patch sizes were larger, then it is possible that the image would have larger color gradients between these blocks. It's helpful to think about the masking step as basically averaging together the values in the block before it's returned back to the viewer. If there are finer differences in an image between smaller blocks of pixels, this process won't capture those differences well, which will make your compression's result worse.

However, this would seem to indicate that an even smaller block size, such as 4x4 or 2x2 should be used. This is not the case because that would make the compression more arduous to perform. For each block, you have to take the DCT, multiply by the mask, and take the inverse DCT. With more blocks, this process would take longer. As we'll see with experimental results later, it is also harder to compress an image to the same accuracy while achieving smaller file sizes when you're using smaller blocks.

Therefore, the 8x8 block size approach has emerged as the dominant way to split up the image [4].

4 Experiments

I wrote code in Matlab to compress an image using the JPEG algorithm. Since I explained the code in the Readme file and in the code itself, I'll focus on the results I saw here.

4.1 Comparing Block Size

A significant aspect of this algorithm is the choice of an 8x8 block size. Therefore, I ran a few experiments to see the impact of this choice. A table of my results is below. Note that the size of the original image was 64KB.

Block size	Output Size (KB)	L2-Norm	Time (sec)
4x4	12	13.75	.406
8x8	10	13.72	.113
16x16	9	13.65	.035
32x32	8.5	13.60	.022

Figure 1: In order of block size used: 4x4, 8x8, 16x16, 32x32

Looking at these results, it would appear that the 32x32 block may be the best. One of the measures that would indicate this is the L2-Norm. Although this isn't officially a metric of image quality, it can still be useful when looking at the deviation of the result image from the original. With this metric, the 32x32 block size had the best results (albeit not by much). It also had the smallest file output size and the fastest runtime.

What these results don't capture is what the images actually look like. In this case, the images looked relatively similar. However, as explained before, in an image with sudden jumps in color, the 32x32 block

size would not accurately portray these areas. It is actually possible to see this phenomena in these four images. If you zoom in closely, you can see the divisions between the blocks. Now look at the middle of the tripod, where there is a light-colored center column. While its bottom has a crisp edge in the first and second photo, is is blurred in the third and fourth. This is likely due to the background color influencing the color displayed in those blocks.

Ultimately, these results are very close together, which drives home the point that the block size choice is relatively arbitrary. As we can see, 8x8 is right in the middle in terms of overall quality, so it seems to strike a good middle ground.

The code to generate these results is in the file CompareBlockSize.m, and the compressed image results are, in order, the files out5.jpg, out1.jpg, out3.jpg, and out4.jpg.

4.2 Comparing 8x8 Masks

Another interesting experiment takes a look at the 8x8 masks that we use for reducing the DCT. The general format for the masks that I used was as follows: the first row had N ones and 8 - N zeros. The second row had N-1 ones and 8 - N + 1 zeros. This pattern repeated, until a row had all zeros. I did these experiments with values of N ranging from 1 to 8. I looked at the impact these masks had on compression size and error for an image whose original size was 256KB. The graphs of the results are below:

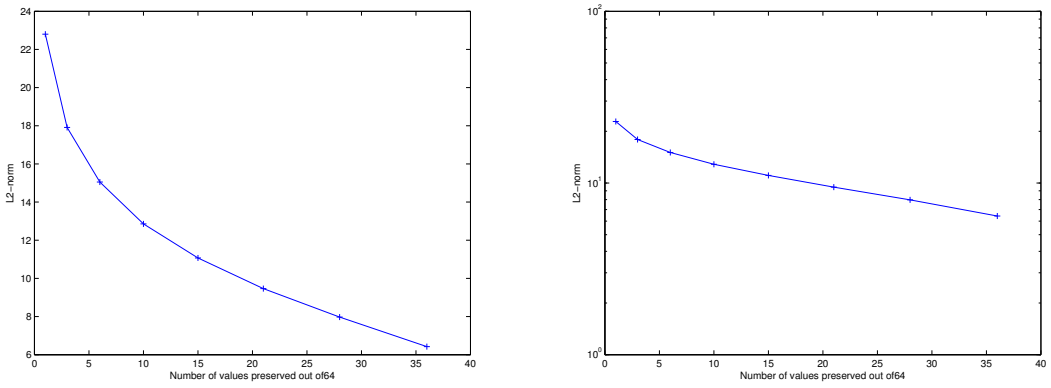

Figure 2: Graph of L2-Norm vs. Number of values preserved out of 64. Left y-axis = regular, Right = log(y)

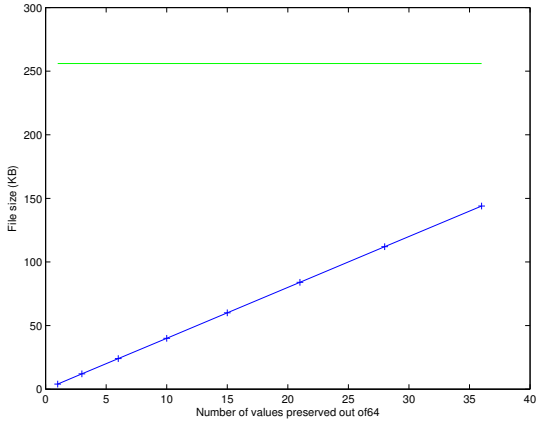

Figure 3: Graph of File size vs. Number of values preserved out of 64. The original file size is plotted in green above

I'll examine the results from the Figure 2 first. These graphs shows how quickly the error from the compression decreases. It appears to be exponentially decreasing, as the second graph with the log(y) scale is basically linear, except for at the beginning. By preserving only about 10 more digits as you move from 1 to 11 on the x-axis, the L2-norm decreases by a factor of about one half. Therefore, you can see huge improvements in your image from an incrementally better mask. Looking at the image results (which are generated when the script is run), you can see a huge image quality jump between each of the first few images. For the last few, however, there is little perceptible difference.

Figure 3 clearly shows a linear relationship between the number of values preserved and the resulting file size. Going back to the explanation of how we calculate memory usage, this makes sense. Every number in the DCT that is preserved is an additional byte of memory. So there should be a linear relationship between keeping values and the memory used.

To confirm this, we can look at this graph. At 36 digits used, the file size is 144KB. Since the original file size is 256KB, this is 0.5625 times as big. By keeping 36 of 64 digits, we're keeping 56.25% of the values in the DCT. Therefore, these results are consistent.

The code to generate these graphs and the compressed images is in the script CompareMasks.m.

5 Conclusion

This paper has explained how the JPEG image compression algorithm works. From looking at experimental results, it seems that for someone writing code for JPEG image compression, there is wiggle room in terms of how they tune the block size and choose the mask they use. Although the standard block size, which is 8-pixels by 8-pixels, is usually the most well-rounded choice, a different block size may be better suited to a particular image. By varying the mask, the image quality and output file size can be changed as well. JPEG is definitely an interesting extension of some of the concepts we learned in class and I hope this paper left you interested in learning more about image compression algorithms.

6 References

1. Zhang, Zhihua and Naoki Saito. High-dimensional data compression via PHLCT. https://www.math.ucdavis.edu/~saito/publications/saito_phlcthd.pdf

2. Yamatani, Katsu and Naoki Saito. Improvement of DCT-Based Compression Algorithm's Using Poisson's Equation. https://www.math.ucdavis.edu/~saito/publications/saito_phlct_final.pdf

3. Austin, David. Image Compression: Seeing What's Not There. http://www.ams.org/samplings/feature-column/fcarc-image-compression

4. MPEG Software Simulation Group. http://www.tns.lcs.mit.edu/manuals/mpeg2/FAQ

5. Moore, Ross. http://fourier.eng.hmc.edu/e161/lectures/dct/node1.html

6. Wallace, Gregory. The JPEG Still Picture Compression Standard. http://nutkin.cs.umu.se/kurser/5DA002/HT11/articles/wallace.pdf