Linear Algebra in Image Compression: SVD and DCT

By: Andrew Fraser

Image compression is a vital tool in sending and receiving images across the web. Its first major use came in during the 1960s when satellites used it to transfer images from space to Earth. It has become even more useful since the implementation of the internet, as smaller sizes became much more important to a media that demanded instant access. The most common use today is in streaming websites like Youtube, Netflix, and Hulu, where 60 images or more are sent in a single second over the internet. Without image compression, ridiculous internet speeds would be needed to do this, but compression allows for up to 10 times less data to be sent in order to get the same picture.

Processes of Image Compression

One of the simplest ways to store an image is in the Raster format, which is essentially an mxn matrix storing the values of each pixel, where m and n are the length and width of the image. For a colored image, the same is done but with three matrices, each one holding the red, green, or blue pixel values for the RGB format. The main linear algebra compression algorithms use this common method of storing images, as it is quite nice to have a matrix when dealing in linear algebra.

One important aspect of image compression is whether it is lossy or lossless. A lossy compression results in some information being lost in the compression, but allows for a more effective compression. A lossless compression stores all of the same information, but just in a compressed state. Usually, the lossless compressions mostly rely on storing bytes differently and don't really apply linear algebra. However, lossy compressions apply multiple linear algebra techniques, including the entire process of SVD and various matrix transformations in the DCT.

One example of a common lossless compression is used in the PNG file format. PNG files are mostly compressed using binary storage methods, with no data lost. One example of a common lossy compression is used in the JPEG file format. JPEG files use a form of the DCT to perform a lossy compression, which is much more effective than the PNG format, but results in a loss in data.

Singular Value Decomposition

SVD, or Singular Value Decomposition, is a matrix factorization in Linear Algebra where $A = U\Sigma V^T$. It is fairly similar to the $AP = PD$ factorization, but it instead uses the square root of the eigenvalues of A^TA, so it can be used on any matrix. To calculate this factorization, the following steps should be performed on a matrix A of size mxn:

Singular values σ_1-σ_n: Found by taking the square root of each eigenvalue of A^TA.
U = A matrix with its columnspace containing the columnspace of A and nullspace of A^T as its columns. All orthogonalized, mxm.

Σ = A diagonal matrix each singular value at its diagonals, from largest at 1,1, to smallest at nxn. All other values are zeros. Mxn
V = A matrix with its columnspace comprised of the eigenvectors of A^TA. Also happens to be the rowspace of A and nullspace of A all orthogonalized. Nxn.
V^T = Transpose of V

This can be used in compression by utilizing the fact that the Σ matrix values go from greatest to least. By removing smaller values from this matrix, most of the information in the image is kept, while the values needed to be stored is reduced. As more and more of these smaller values are set to zero, columns from the U and rows from the V^T matrices can be set to zero as well, as they would simply be multiplied by zeros from the Σ matrix anyway. Thus, for each small value in the SVD that is set to zero, both U and V^T also lose an entire row/column, so much less needs to be stored entirely.

Discrete Cosine Transform

The DCT, or Discrete Cosine Transform, is a transformation that uses matrix multiplications to compress matrix data. First, the image is split into many NxN matrices, with the ideal number for N usually being 8. Then, an NxN transformation matrix is created using a cosine formula based on the i, j, positions in the matrix. Each NxN square of the image is then left multiplied by this matrix, and right multiplied by the transpose of the matrix.

$$D = TMT^T$$

After this process, there are different NxN quantization matrices ranging from compression percentages 0-100% that can be used to compress each transformed matrix. For example, the 50% percent matrix creates a reasonably compressed matrix by turning a decent amount of values into zeros. A 10% matrix would result in large amounts of compression with a lower quality image, while a 90% would perform little compression, but keep almost all of the data. For each zero that results in this compression, the image becomes more and more compressed. Each i,j value in the NxN matrix of data is divided by the i,j value in the quantization matrix, then rounded to the nearest whole number. It is this division that results in many zeros in the matrix, allowing for much less data to be stored overall.

By storing all of these NxN matrices with many zeros each, the matrices become much less compressed. Then, when a user wants to view the image, the reverse process is performed by multiplying each i,j value in the quantization by each value in the data, then left multiplying by the DCT matrix and right multiplying by the . Finally each NxN block is recombined into the full raster image block, which can be viewed like a normal image.

$$M = T^TDT$$

Applying the SVD

For performing the SVD, I decided to use Maple because it is what I was most familiar with, and it contains all of the necessary tools to perform an SVD compression. Here is a text copy of the code I wrote:

```
with(LinearAlgebra):
with(ImageTools): # Necessary to read images as matrices and manipulate them
img:= ToGrayscale(Read("/u/class/f/c-fras2/Downloads/Robot.jpg")): # Reads the black and white image
Write("/u/class/f/c-fras2/Pictures/Initial Robot.JPG", img):
U:= LinearAlgebra[SingularValues](img, 'output = U'):
S:= LinearAlgebra[SingularValues](img, 'output = S'):
Vt:= LinearAlgebra[SingularValues](img, 'output = Vt'):

C := 5/100:
for i from (round((RowDimension(S) * C) + 1)) to RowDimension(S) do
S[i] := 0:
end do:
DiagS:= DiagonalMatrix(S, RowDimension(img), ColumnDimension(img)):
CompressedImage:= U.DiagS.Vt:
Write((cat("/u/class/f/c-fras2/Pictures/", convert(round(C * 100), string), "% Robot SVD.png")), CompressedImage):
```

This code first reads an image in grayscale, then finds the full SVD decomposition of the matrix. Then, it turns a set amount of the smaller Σ values to zeros, allowing U and V^T to store that many less values as well. The value C is used to easily set what percentage of compression is desired, a smaller percentage meaning that more compression occurs but more data is also lost. Then, the image is turned back into its original form and written to a file to be viewed.

Applying the DCT

I decided to use Matlab to perform the DCT because it includes many more tools for performing the DCT on a matrix than Maple. Matlab included a method that automatically split a matrix into 8x8 chunks, then performed the compression itself. It also included a method to automatically decompress them. The only missing portion was the quantization matrices, which I had to perform manually. Here is the code that I wrote to do this:

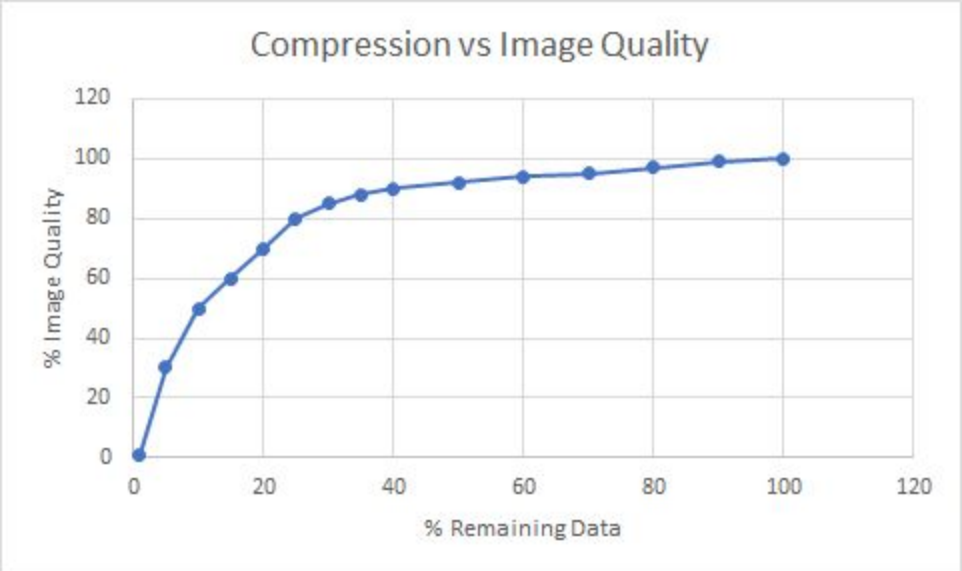
```
A = im2double(imread('/u/class/f/c-fras2/Pictures/Initial Robot.JPG'));
D = dct2(A);
D(abs(D) < .1) = 0;
count = 0;
for m=1: size(D,1)
   for n=1:size(D,2)
      if D(m, n) == 0
         count = count + 1;
      end
   end
end

percent = round((1 - (count/(size(D,1) * size(D,2)))) * 100);
R = idct2(D);
filepath = strcat('/u/class/f/c-fras2/Pictures/', num2str(percent), '% Robot DCT.png');
imwrite(R, filepath);
```

First, the image is loaded in and the dct2 method is used to create the DCT compressed matrix for A. Then, each value below a certain amount (here it is set to .1) is set to zero. This sets the smaller, less useful values in the stored matrix to zero. Then, the number of zeros in the matrix is counted to determine the percentage of compression. Finally, the D matrix is inverted back into a normal raster matrix, then is written to a file.

Effectiveness of Compression Techniques

The SVD, DCT, and other compression techniques tend to have a golden ratio of compression where the image is still very distinguishable, yet a large amount of compression is performed. For the SVD and DCT in particular, an average curve looks somewhat like this:



Based upon this curve, it is clear that the compression starts off by removing many values that are unnecessary and barely matter to the visibility of the piece. However, around 15% of the remaining data, much more important information starts to be lost. Picture quality starts to decrease rapidly. By 10%, things start to get visibly blurry, but still visible. By 1%, the image becomes difficult to distinguish. Thus, the golden ratio of compression tends to lie between 15% and 25%, depending on how high quality the image needs to be. Below that point, it is often not worth the extra bit of compression for such a lower quality image.

Results - Bridge

100%

75% SVD

69% DCT

50% SVD

47% DCT

30% SVD

35% DCT

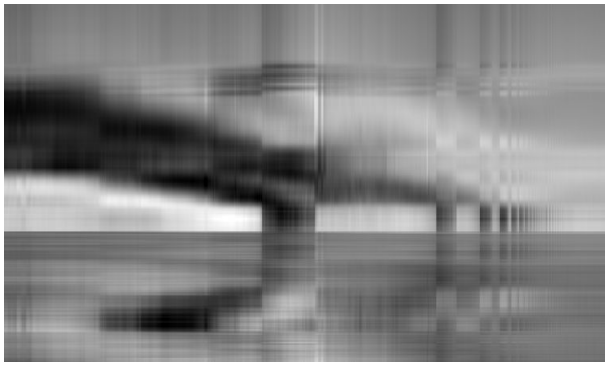
20% SVD

20% DCT

10% SVD

12% DCT

1% SVD



2% DCT

It is interesting to see how the images get lower in quality in different ways. As you can see with the SVD images at 10% and 1%, blurry lines start to appear in the image. It is as if entire lines of the image are stripped away. This makes sense in relation to the transformation because as each singular value is removed, an entire eigenvector is removed, so lines or "vectors" are removed from the image at a time.

In contrast, the DCT becomes spottier in the entire picture, which makes sense because values are taken out of the matrix one at a time based on value, not vector by vector. This is visible in the sky at the 20% mark, and becomes more visible in other spots at 10% and 2%. The image is still recognizable at 2%, which is quite a feat, but the quality is still poor enough that the sign can't be read.

Overall, these results show why the JPEG format uses the DCT. The DCT retains more quality overall because it affects the whole image the same way, so everything stays somewhat visible even at single percentages of data remaining. However, since the SVD removes entire vectors, parts of the image become quite messed up, such as the streaks in the sky at 10%, whereas other parts are less affected.

Results - Robot

100%

75% SVD

76% DCT

50% SVD

57% DCT

30% SVD

30% DCT

15% SVD

15% DCT

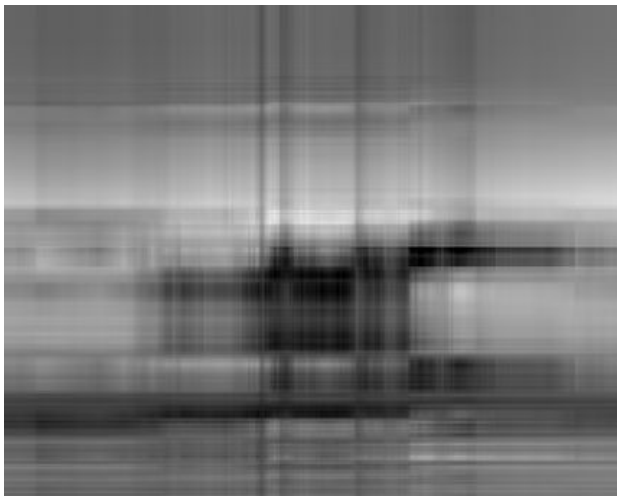
10% SVD

11% DCT

5% SVD

4% DCT

1% SVD



1% DCT

These results further illustrate why the DCT is used in the JPEG algorithm. Even at the level of 10% data remaining for the robot, everything is still fairly visible to the viewer. In fact, it is possible that the golden ratio for the DCT may even drop into around 10%, as it still retains reasonable quality at that point where the SVD does not. Of course, higher percentages are still needed if quality is important, but otherwise 10% is actually quite viable.

Conclusion

The SVD and DCT techniques are both very useful for data compression, and can easily compress an image to 30% of its original size with almost no visual difference. However, based on these results, the DCT appears to be more effective because it takes its losses throughout the entire photo evenly, instead of removing entire vectors of data at a time. To create a high quality image that loses almost no important data, the 30% mark seems to be about the spot to stay, as very little differences can be seen between that point and the original image. For images that don't worry about quality, about 15% for the SVD and 10% DCT is where the line should likely be drawn, as quality becomes too poor past those points to be worth it.

Sources

https://www.math.cuhk.edu.hk/~lmlui/dct.pdf
http://videocodecs.blogspot.com/2007/05/image-coding-fundamentals_08.html
http://www.mvnet.fi/index.php?osio=Tutkielmat&luokka=Yliopisto&sivu=Image_compression
https://ntrs.nasa.gov/search.jsp?R=19920024689
https://www.sitepoint.com/gif-png-jpg-which-one-to-use/