Image Compression

Justin Wyss-Gallifent

July 21, 2021

10.1 Image Representation

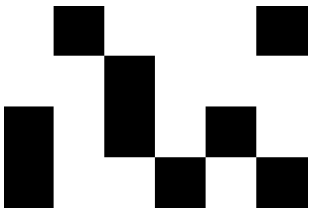
There are a variety of ways to store image data but when graphics are displayed pixel-by-pixel then we need to store data for each pixel.

Typical approaches:

- For a color image combine Red, Green and Blue, each with an integer between 0 and 255 inclusive. This gives a total of $256^3 = 16777216$ colors available.
- For a grayscale image assign a single integer between 0 and 255 inclusive, where 0 indicates Black and 255 indicates White.
- For a grayscale image assign a single real number between 0 and 1 inclusive, where 0 indicates Black and 1 indicates White.

In order to facilitate easy mathematical calculations (no integer truncation, etc.) we're going to stick with the third option.

Example 10.1. For example the following image consists of no grayscale, only black and white:



This image is represented by the matrix:

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

10.2 Image Compression

We saw in the previous chapter that we can use the singular value decomposition of a matrix to find an approximation to that matrix which preserves a certain amount of the original matrix's variance.

It follows that if the matrix represents an image then an approximation to that matrix can be thought of as an image which preserves a certain amount of the original image's variance.

Example 10.2. If we find the SVD of this matrix we have:

$$A = \begin{bmatrix} -0.51 & 0.57 & -0.50 & 0.40 \\ -0.66 & -0.29 & -0.19 & -0.66 \\ -0.40 & -0.64 & 0.18 & 0.63 \\ -0.38 & 0.42 & 0.82 & -0.05 \end{bmatrix} \begin{bmatrix} 3.198 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1.678 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1.235 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.6576 & 0 & 0 \end{bmatrix} V^T$$

If we set the smallest singular value to zero to create Σ' and compute:

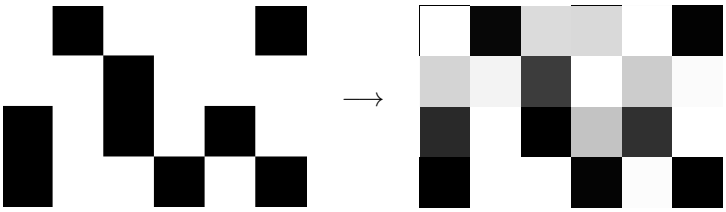
$$A' = U\Sigma' V^T = \begin{bmatrix} 1.103 & 0.030 & 0.856 & 0.848 & 1.122 & 0.011 \\ 0.831 & 0.951 & 0.236 & 1.248 & 0.800 & 0.982 \\ 0.162 & 1.047 & -0.226 & 0.762 & 0.191 & 1.018 \\ -0.012 & 0.997 & 1.017 & 0.018 & 0.986 & -0.001 \end{bmatrix}$$

Before we charge ahead and look at the resulting image we notice an issue. The values in this new matrix are not necessarily between 0 and 1.

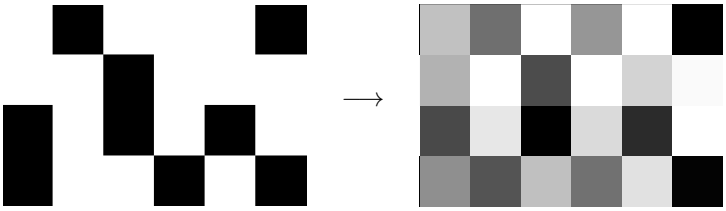
There are two possible approaches to this. The first would be to renormalize the values between 0 and 1. However there's a good argument against this, that being that (for example) a value of 1 is supposed to represent completely White so if we had a value such as 1.1 then rescaling would interpret the first as not completely White but rather as slightly more grey, which is not accurate at all.

Instead we'll take the second approach and simply leave the values alone. Values above 1 will be treated as White (like 1) and values below 0 will be treated as black (like 0).

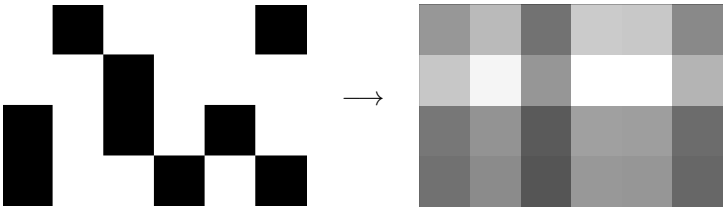
Example 10.3. With this in mind our previous example, preserving three singular values (and 97.12% of the matrix data variance), would be, along with the original:



If we do it again preserving two singular values (and 86.95% of the matrix data variance), along with the original:



and then one singular value (and 68.18% of the matrix data variance), along with the original:



As the number of singular values decreases the image loses variance and becomes more uniform while still trying to retain as much variance as possible.

Here is an even better example:

Example 10.4. The following image is represented by a 200×200 matrix A: This is Justin, age 8.

If we do a SVD for the matrix A we see that there are 200 singular values. We won't list them all but here are the ones that are greater than 0.5:

```
114.6751, 23.1226, 17.6162, 10.0497,
8.8713, 7.4690, 7.0125, 5.5884, 4.7874, 4.4234, 4.1422, 3.7599, 3.1233,
2.9382, 2.8150, 2.6498, 2.3993, 2.1972, 2.0984, 1.9532, 1.8977, 1.7923,
1.7188, 1.5803, 1.4746, 1.3644, 1.3258, 1.3239, 1.1909, 1.1655, 1.0972,
1.0582, 1.0382, 1.0117, 0.9170, 0.9015, 0.8496, 0.7791, 0.7556, 0.7100,
0.6853, 0.6507, 0.6407, 0.6081, 0.5906, 0.5732, 0.5585, 0.5356, 0.5013
```

If we zero out all but those highest 100, recreate the matrix and view we get:

It's really hard to see any difference here.

We can do this preserving any number of values.

Here are the images resulting from preserving 200 (all), 100, 50, 20, 10, 5 and 1 singular values:

200 values

100 values

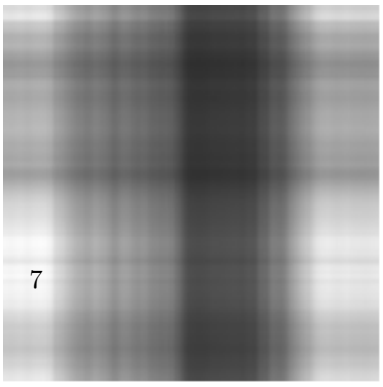
50 values

20 values

10 values

5 values

2 values



1 value

We can see that the rank 50 version is still quite good, it's only when we get to rank 20 that noticeable deterioration of image quality starts to take place.

The beautiful thing about the final rank 1 version is that the entire image is constructed out of a single vertical vector whose values correspond to shades of grey. Each column is a multiple of that value and you can see that easily!

10.3 Image Quality

The image quality can be defined as the proportion of variance preserved in the new image. Recall that this is the sum of the squares of the retained singular values divided by the sum of the squares of all of the singular values.

In the Justin case if we look at the version preserving 50 singular values (the last pretty good one) then we find that the image quality can be defined as:

$$\frac{s_1^2 + ... + s_{50}^2}{s_1^2 + ... + s_{200}^2} = 0.999769$$

meaning that we've preserved 99.9769% of the original image variance. Loosely speaking this image is 99.9769% as good as the original.

Here are the values for all of the versions:

# Singular Values	Variance Preserved	Quality Percentage
200	1.00	100
100	0.999985	99.9985
50	0.999769	99.9769
20	0.997387	99.7387
10	0.991623	99.1623
5	0.979279	97.9279
2	0.945426	94.5426
1	0.908489	90.8489

This is interesting because it gives us a sense of how much variance we would need to preserve to keep a reasonable-looking picture, albeit for only one example. While 90% seems like a lot, in this case the resulting picture (the final one) is not good at all, and we'd probably aim for perhaps 99.5% or more.

If we set a threshold at 99.5% then a quick calculation shows that we need at least 14 singular values to do the job.

10.4 Data Savings

Importantly we need to address why we would do this at all.

Here we'll look at $n \times n$ (square) images. Images which are not square take a little tweaking and are addressed in the exercises.

If we wish to save a grayscale $n \times n$ image in a matrix A using one value per pixel then we need to save n^2 values.

Suppose instead we use the SVD and preserve $k < n$ singular values. In the previous chapter we saw that this means we're essentially recalculating the matrix A' as follows:

$$\begin{aligned}
\bar{a}'_1 &= s_1 v_{11} \bar{u}_1 + s_2 v_{12} \bar{u}_2 + ... + s_k v_{1k} \bar{u}_k \\
\bar{a}'_2 &= s_1 v_{21} \bar{u}_1 + s_2 v_{22} \bar{u}_2 + ... + s_k v_{2k} \bar{u}_k \\
&\vdots \qquad\qquad\qquad \vdots \\
\bar{a}'_n &= s_1 v_{n1} \bar{u}_1 + s_2 v_{n2} \bar{u}_2 + ... + s_k v_{nk} \bar{u}_k
\end{aligned}$$

If we think of the right hand side as simply linear combinations of $\bar{u}_1$, ..., $\bar{u}_k$ then all we really need to save are the vectors $\bar{u}_1, ..., \bar{u}_k$, each in $\mathbb{R}^n$ and therefore consisting of n values each for a total of kn values, and the coefficients to find each $\bar{a}'_i$, consisting of k values each for a total of kn values again. Thus in sum total we need to save $2kn$ values.

The value $\frac{2kn}{n^2} = \frac{2k}{n}$ can be throught of as the compression ratio and provided $2nk < n^2$, or $k < \frac{n}{2}$, we've saved space.

Example 10.5. If we again focus on the Justin example, preserving 20 singular values, we can store the image using $2kn = 2(20)(200) = 8000$ values instead of $200^2 = 40000$ values for a compression ratio of $\frac{8000}{40000} = 0.20 = 20\%$.

Preserving 50 singular values, quite a decent picture, we can store the image using $2kn = 2(50)(200) = 20000$ values instead of $200^2 = 40000$ values for a compression ratio of $\frac{20000}{40000} = 0.50 = 50\%$.

10.5 Matlab

A matrix with values between 0 and 1 can be displayed in Matlab as follows, where 0 is black and 1 is white:

```
>> A = [1 0.5 1 1 0.9 0;1 1 0 1 1 1;0 0.1 0 1 0 1;0 1 1 0 1 0.3];
>> imshow(A,'border','tight','InitialMagnification',1000)
```

Note that the `'InitialMagnification',1000` setting makes the picture 1000% of the original size, otherwise our image (which is only a few pixels wide and high) would be too small to see.

The `'border','tight'` just gets rid of the border around the image. This isn't really necessary unless you're going to save the image and don't want to save the border.

We can then do a singular value decomposition on it and manipulate it as before. For example the matrix above has four singular values so let's zero out the smallest and redisplay:

```
>> [U,S,V] = svd(A);
>> SP = S;SP(4,4)=0;
>> AP = U*SP*transpose(V);
>> imshow(AP,'border','tight','InitialMagnification',1000)
```

Now then, if we'd like to read in a graphics file we can do so quickly. We use `imread` to read the image into a matrix. Next we use `rgb2gray` to ensure the image is grayscale (otherwise the matrix might have an extra dimension containing red, blue and green values). After that we use `im2double` to convert the values from `uint8` (the default unsigned integer values, which we can't directly use svd on) to double-precision real numbers (more than sufficient for our purposes but Matlab default). Lastly we scale them between 0 and 1 by subtracting the minimum of all of them (making the minimum 0) and then dividing by the maximum of all of them (making the maximum 1).

Note that `A(:)` refers to all elements in the matrix `A` so that wrapping those in `max` and `min` gets the maximum and minimum of all elements.

All together:

```
>> A = imread('justin.jpg');
>> A = rgb2gray(A);A = im2double(A);A = A-min(A(:));A = A/max(A(:));
```

If we'd like to view it:

```
>> imshow(A,'border','tight');
```

Now then, if our file has lots of singular values and we want to zero out a bunch of them, a for loop can help. For example the above `justin.jpg` file is the image from this chapter and has 200 singular values. If we only want to keep the first 100 and plot:

```
>> [U,S,V] = svd(A);
>> SP = S;for i=[101:200];SP(i,i)=0;end;
>> AP = U*SP*transpose(V);
>> imshow(AP,'border','tight');
```

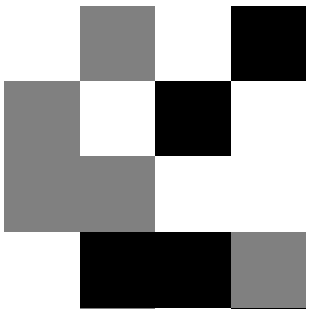
Okay, so how about calculating the proportion of variance? Well, we can use the `diag` command to extract the values out of our Σ and play with the resulting vector of values. For example here's the proportion of variance in the first 100 singular values of the original Justin picture from this chapter, loaded earlier:

```
>> svals = diag(S);
>> vpa(sum(svals(1:100).^2)/sum(svals.^2),6)
ans =
0.999985
```

Notation Note: The reason we use `.^` instead of `^` is that `diag(S)` and hence `svals` and `svals(1:100)` are all vectors which are treated as matrices in Matlab. Simply doing, for example, `svals^2` would attempt in this case to multiply a 200×1 matrix by itself, which doesn't work. What we really want to do is take each entry in that matrix and square it, and this is what `.^2` does. Basically it applies the `^2` operation element-by-element.

10.6 Exercises

Exercise 10.1. Consider the following image:

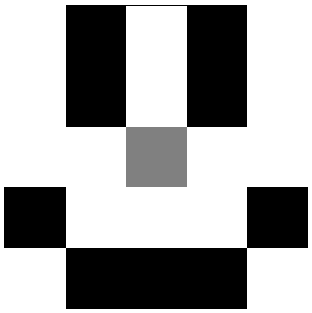


(a) Enter this image into a 4×4 matrix A. Assume the shades you see are only 0, 0.5 and 1.

(b) Find the SVD for A.

(c) Simplify the matrix preserving 3, 2 and then 1 singular value. Draw the best image representation you can of each matrix.

Exercise 10.2. Consider the following image:



(a) Enter this image into a 5×5 matrix A. Assume the shades you see are only 0, 0.5 and 1.

(b) Find the SVD for A.

(c) Simplify the matrix preserving 4, 3, 2 and then 1 singular value. Draw the best image representation you can of each matrix.

Exercise 10.3. Find a square image on the internet of reasonable size; 200×200 or thereabouts is good.

(a) Load this into the Matlab matrix A, convert to grayscale if necessary and scale the values between 0 and 1.

(b) Find the SVD for A.

(c) How many singular values are there?

(d) Simplify the image by preserving only 75% of the singular values. What is the percentage of data quality preserved? Print the result.

(e) Simplify the image by preserving only 50% of the singular values. What is the percentage of data quality preserved? Print the result.

(f) Simplify the image by preserving only 25% of the singular values. What is the percentage of data quality preserved? Print the result.

(g) What is the minimum number of singular values that must be preserved in order to preserve 99.9% of the data quality? What level of data compression would this achieve? Simplify the image accordingly and print the result.

Exercise 10.4. Try to find two images, both 200×200, both photographs, one requiring as few singular values as possible and one requiring as many singular values as possiboe, both to achieve 99.9% image variance. Identify, if you can, what about the images leads to this disparity. Lots of darks and lights? Regular patterns? Anything you can find!

Exercise 10.5. Assume for each of the following that the singular values have

been given for a 10×10 image. Determine the minimum number of singular values in decreasing order that would need to be preserved to keep 99.9% of the image variance and what the resulting data compression ratio would be.

(a) $\{80.2608, 63.3520, 20.5871, 8.4696, 2.8841, 1.6763, 0.7962, 0.6926, 0.6553, 0.6520\}$

(b) $\{138.5481, 49.5181, 16.5869, 4.9396, 3.2379, 1.5248, 1.1992, 1.0277, 1.0208, 0.9786\}$

(c) $\{446.1649, 163.4218, 43.8892, 13.7979, 4.7417, 1.3437, 1.0500, 0.6422, 0.4532, 0.4484\}$

(d) $\{209.4851, 28.1916, 22.8720, 11.8304, 6.3254, 2.7847, 1.2043, 0.9255, 0.8765, 0.8722\}$

Exercise 10.6. Suppose we simplify an $m \times n$ (not necessarily square) image using k singular values. What is the resulting data compression ratio? What criteria on k would make this worth doing?