ECE532 – Final Project

Kyle Daruwalla
daruwalla@wisc.edu

Davis Gilton
gilton@wisc.edu

Shuoyan Qiu
sqiu26@wisc.edu

15 Dec. 2016

Executive Summary

This lab uses the Singular Value Decomposition (SVD) on video data to explore the idea of a low-rank approximation to a video and to perform some video editing tasks. The lab begins with an outline of the idea of the SVD, and then begins exploring low-rank approximations to image data. Afterwards, the structure of video data is explored and manipulated to allow the SVD to be applied in a productive way.

By structuring video data in a "flat" way, the concepts underlying the SVD and low-rank approximations are explored through interesting, relevant examples. In addition to building a more solid basis in the theory of the SVD, this lab will allow the student to explore the effects of preprocessing the data that they analyze, and the lab will conclude with an example of how the SVD might be used to edit video data, combining the previous work done throughout the lab.

1 Background

The Singular Value Decomposition (SVD) is a popular mathematical tool, with applications in linear algebra, signal processing, image processing, statistics, and control theory. The SVD factorizes an $m \times n$ complex matrix of rank-r, A, into the form:

$$A = U\Sigma V^\intercal$$

Where U is an $m \times m$ unitary matrix and V is an $n \times n$ unitary matrix. Σ is an $m \times n$ rectangular matrix, which has only r nonzero elements (called "singular values") on its diagonal, which are traditionally arranged in descending order. All other elements of Σ are zero. So we assume $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r > 0$. As expected, $r = \text{rank}(A) \leq \min(m, n)$.

We call the columns of U, $[u_1, u_2, \ldots, u_m]$, the left singular vectors of A. Similarly, we refer to the columns of V, $[v_1, v_2, \ldots, v_n]$, as the right singular vectors of A.

In this lab, we will also refer to the "economy" SVD. This version of the SVD factorizes A as:

$$A = U_1\Sigma_1 V_1^\intercal$$

Where Σ_1 is an $r \times r$ diagonal matrix containing only positive singular value elements. U_1 is an $m \times r$ unitary matrix containing only the first r left singular vectors of A, and V_1 is an $n \times r$ unitary matrix containing only the first r right singular vectors of A. This decomposition is smaller than the full SVD, and so can be useful when dealing with extremely large matrices.

Now we can represent A as a linear combination of rank-1 matrices:

$$A = \sigma_1 u_1 v_1^\intercal + \sigma_2 u_2 v_2^\intercal + \ldots + \sigma_r u_r v_r^\intercal$$

From class, you might remember the Eckart-Young Theorem [1]:
Theorem 1. *Given a matrix, $A \in \mathbb{R}^{m \times n}$ that has rank r, we can find a rank $k \leq r$ approximation to A, X as*

$$\sum_{i=1}^{k} \sigma_i u_i v_i^\top$$

where σ_i are the singular values of A, and u_i and v_i are the singular vectors of A. Moreover, X is the solution to the problem

$$\arg \min_{\text{rank}(B) \leq k} \|A - B\|$$

for any unitarily-invariant norm.

This tells us that the SVD can be used to find the best low-rank approximation possible. In particular, these low-rank methods can be used for image processing.

There are many interesting applications of the SVD in image processing, including several related to Principal Component Analysis (PCA) and advanced filtering techniques [2][3][4]. The SVD can also be used on video and image data to reconstruct fragmented or damaged photos and videos [5].

In this lab, we will first explore some example image processing techniques, particularly low-rank approximations. Afterwards, we will investigate how to structure video data to be analyzable using our methods, and explore the significance of a low-rank video approximation. Finally, we will conclude with a video editing scheme based entirely around the SVD. Let's start with the warm-up!

2 Warm-Up

The following problems are designed to familiarize you with concepts you will use throughout the lab. We will tackle four topics:

1. Performing compression using the SVD on image data
2. Working with video data in MATLAB
3. Performing compression using the SVD on frames of video data
4. Reshaping video data to achieve the main results of the lab

2.1 Compressing image data

Consider a grayscale image where each pixel is a value from 0 to 255. We can represent this image as a matrix of pixel values. Thm. 1 can be used to find a rank r approximation to this matrix. Now, our image is represented by a low-rank matrix X, thereby compressing the image size.

Problem: Download the image bucky.csv. Your task is to use MATLAB and Thm. 1 to compress this image so it can be represented by a rank 50 matrix. Recall from a previous homework set that this will entail setting some of the singular values to 0. Solutions are included in Appendix A.1.

2.2 Video data in MATLAB

So far, we've only worked with image data in MATLAB. These particular images contained only two dimensions – width and height. Videos contain four dimensions – width, height, channels, and time. First, a video is a series of frames in time. Each frame is still three dimensions. The third dimension, known as the channel, corresponds to the red (R), green (G), and blue (B) channels of a color image. Each channel is only two dimensions like the images we've worked so far. Previously, our image data was in grayscale, which is only one channel instead of three. In order to perform the problems in this lab, we will need to convert the video data from color to grayscale.

Problem: Use the code snippet below to read in a video built in to MATLAB. Then go through the video frame-by-frame and convert it from RGB to grayscale. (*Hint:* you might find the MATLAB documentation on `VideoReader`, `rgb2gray`, and `implay` helpful). Solutions are included in Appendix A.2.

```
v = VideoReader('xylophone.mp4');
videoMatrix = v.read;
```

2.3 Compressing frame by frame

Based on the previous two problems, the obvious next step to applying SVD compression to video data might be to apply it frame by frame.

Problem: Use your result from the previous warm-up problem to compress the grayscale video frame by frame. Use the same compression value ($k = 50$). What do you notice about the output? Is it as you expected? Try it with a lower compression value ($k = 20$) to see more drastic results. Solutions are included in Appendix A.3.

2.4 Reshaping video data

This problem is a simple thought experiment. As shown in Fig. 1, we have a viewing window frame (represented by the red box), and we are sliding a 1px × 1px black square across the screen. Ignore the numbers in Fig. 1 for now.

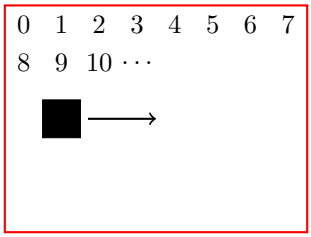

Figure 1: The red box is the viewing frame of the video capture device. The black square moves under the viewing frame from left to right.

Problem: Focus on the pixel at (0, 0). What single value represents its value over all of time? Now focus on a pixel in the same row as the black box. If you had to guess at a random time what its value was, what would be the best value to guess? (*Hint:* don't over think it or actually calculate the average).

Now suppose you vectorize the 2D viewing window – count across the pixels in the frame as shown in Fig. 1 and place them in a row vector in that order. Suppose there are N total pixels in the frame, then the row vector should be $1 \times N$. This vector represents the values of all the pixels at one moment in time. Now create another such row vector for the next moment in time, and so on, until you have T $1 \times N$ vectors. Arrange these vectors in a matrix so that each row of the matrix is one of the row vectors, and as you go down the rows of the matrix, you are advancing in time. Note that each *column* of this matrix is the values of a single pixel across all values of time. What would this matrix look like for the experiment in Fig. 1? You can now choose a *column* vector to represent all the columns of this matrix. What vector do you choose?

The standard way to reshape a matrix into a vector is to "stack" the columns so that the matrix $(M \times N)$ becomes an $MN \times 1$ column, with the first column of our matrix on top, the second immediately below it, and so on. We will use this method for reshaping videos in this lab.

3 Lab

3.1 Setup

During the course of this lab we will operate on the "xylophone.mp4" video that you used in the warm-up section.

If you have not already done so, use the code developed in Warm-Up Problem 2 to read in the "xylophone.mp4" video and convert it to grayscale. After Warm-Up Problem 4, you should have a good idea of how to reshape the data to make the three-dimensional video data into two dimensions. Convert the 3-D grayscale video matrix into a 2-D matrix.

(*Hint:* Ensure that the dimensions of the resulting 2-D matrix are $(320 * 240) \times$numberOfFrames)

Test that you can recreate the original grayscale video using only the results of the SVD.

3.2 SVD and Eckart-Young With Video Data

We will now take the SVD of the video data, to try to see what a low-rank approximation of a video might look like. As we saw in Warm Up Problem 3, computing the SVD for every frame is a valid approach, but doesn't produce any effects unique to videos. So we want to find a way to compute the SVD across an entire video.

The SVD is only defined for 2-D matrices, so we have to use the flattened matrix we created in Lab Problem 1.

Should you take the economy SVD, or the full SVD? What will be the dimensions of the resulting U, S, and V matrices in both cases? How much storage space would these matrices require?

In the Warm-Up Problem 1, you reviewed how to do a low-rank approximation of image data. There, you let $k = 20$. Here, let $k = 6$, and reconstruct an approximation to "xylophone.mp4" letting only the first 6 singular values be nonzero.

Convert this into a video, and watch the video using MATLAB's `implay()` function (remember that you need to convert the data to `uint8` before watching the video!). What do you observe? Why might this be?

Given another video, what would you expect a very low-rank approximation of that video to look like?

3.3 Inverse Eckart-Young?

In the previous problem, we were able to extract the low-rank approximation to our video data. In practice what this ended up meaning is that we removed those objects in the video that weren't constant over the time dimension – they were moving.

The result was extracting only the background. But what happens to the moving objects (and the lighting changes)? Which singular values would you set to zero to extract only the objects in the video that are moving?

Using the results of the SVD that you took in Lab Problem 2, extract the moving parts of "xylophone.mp4" and convert them back into a video. Play back the video to ensure that your method worked.

3.4 Video Editing with the SVD

We now have all the tools we need to do some simple video editing. The conclusion of our lab will be using the ideas we have developed so far to replace the background of our "xylophone.mp4" video.

This can be done with any image of appropriate size, as long as that image is grayscale and 240×320 pixels. MATLAB has a built-in image that can be used for this purpose when downsampled using the following code snippet:

```
backgroundImage = imread('AT3_1m4_03.tif');
backgroundImage = downsample(downsample(backgroundImage,2)',2)';
```

Reshape this image, and then take the SVD as if it were a single-frame video. The reshaped image should be a single column vector. What size do you expect your U, S, and V matrices to be? After taking the SVD, did the result make sense?

Because this is a single-frame video, the V vector is 1×1. Is there a way to make this V vector more closely resemble the V vector produced when taking the SVD of the "xylophone.mp4" data? (*Hint:* You could create a video where every frame was this image; what would your V vector look like in that case?)

Replace the first left singular vector of the "xylophone.mp4" 2D matrix with the U from our image. Similarly, replace the right singular vector with the V vector you just created. You should also replace the top singular value with the singular value you obtained in this problem.

Recreate and play the new video. What do you observe?

Experiment. Can you remove the xylophone keys, but leave the arm and stick? How about the other way around?

Bibliography

[1] C. Eckart and G. Young, "The approximation of one matrix by another of lower rank," *Psychometrika*, vol. 1, no. 3, pp. 211–218, 1936.

[2] H. Andrews and C. Patterson, "Singular value decompositions and digital image processing," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 24, pp. 26–53, Feb 1976.

[3] D. Muti, S. Bourennane, and M. Guillaume, "Svd-based image filtering improvement by means of image rotation," in *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 3, pp. iii–289–92 vol.3, May 2004.

[4] W. Murase and M. Lindenbaum, "Partial eigenvalue decomposition of large images using spatial temporal adaptive method," *IEEE Transactions on Image Processing*, vol. 4, pp. 620–629, May 1995.

[5] H. Ji, S. Huang, Z. Shen, and Y. Xu, "Robust video restoration by joint sparse and low rank matrix approximation," *SIAM Journal of Imaging Sciences*.

[6] R. Kakarala and P. O. Ogunbona, "Signal analysis using a multiresolution form of the singular value decomposition," *IEEE Transactions on Image Processing*, vol. 10, pp. 724–735, May 2001.

Appendix A Solutions

This appendix contains solutions to the warm-up problems and lab.

Appendix A.1 Warm-Up Solution #1

The code below will produce a rank 50 approximation to the original image. Notice how the image is blurred from compression. Play with the value of k and see what the results look like.

WarmUp1.m

```
clear variables, close all

% Read in image data and display it
A = csvread('bucky.csv');
figure(1); imagesc(A,[0 1])
colormap gray; axis image; axis off

% Compute the SVD of A
[U, S, V] = svd(A);
s = diag(S);

% Set our compression to rank 50
k = 50;

figure(2);
[m, n] = size(S);
% Zero out the > k singular values of A
S_prime = diag(s(1:k));
S_prime = [S_prime zeros(k, n - k); ...
    zeros(m - k, k) zeros(m - k, n - k)];
% Reconstruct the image matrix and display it
A_prime = U * S_prime * V';
imagesc(A_prime,[0 1]);
title(['Compression␣with␣k␣=␣', num2str(k)]);
colormap gray; axis image; axis off;
```

The image that should be produced is:

Appendix A.2 Warm-Up Solution #2

The code below reads in a video file and converts each frame from RGB to grayscale.

WarmUp2.m

```
% Read in all frames of built-in MATLAB video data
v = VideoReader('xylophone.mp4');
video_matrix = v.read;

% Get the dimensions of the video data
num_frames = v.Duration * v.FrameRate;
video_dim1 = size(video_matrix, 1);
video_dim2 = size(video_matrix, 2);

% Initialize a 3D matrix to store the grayscale video data
gray_video = zeros(video_dim1, video_dim2, num_frames);

% Convert from RGB to grayscale, frame by frame
for i = 1:num_frames
    gray_video(:, :, i) = rgb2gray(video_matrix(:, :, :, i));
end

% Play the results
implay(uint8(gray_video));
```

Appendix A.3 Warm-Up Solution #3

The code below will compute video compression by doing standard SVD image compression on each frame individually. It uses a compression value of $k = 20$, but the value of k can be changed in the code easily to compute the output for different compression values.

WarmUp3.m

```
% Read in all frames of built-in MATLAB video data
v = VideoReader('xylophone.mp4');
video_matrix = v.read;

% Get the dimensions of the video data
num_frames = v.Duration * v.FrameRate;
video_dim1 = size(video_matrix, 1);
video_dim2 = size(video_matrix, 2);

% Initialize a 3D matrix to store the grayscale video data
gray_video = zeros(video_dim1, video_dim2, num_frames);

% Convert from RGB to grayscale, frame by frame
for i = 1:num_frames
    gray_video(:, :, i) = rgb2gray(video_matrix(:, :, :, i));
end

% Set our compression to rank 20 (change to 50 if desired)
k = 20;

% Compress the video frame by frame
compressed_video = zeros(video_dim1, video_dim2, num_frames);
for i = 1:num_frames
    % Compute the SVD of the frame
    [U, S, V] = svd(gray_video(:, :, i));
    s = diag(S);
    [m, n] = size(S);
    % Zero out the > k singular values of A
    S_prime = diag(s(1:k));
    S_prime = [S_prime zeros(k, n - k); ...
        zeros(m - k, k) zeros(m - k, n - k)];
    % Reconstruct the frame matrix
    compressed_video(:, :, i) = U * S_prime * V';
end

% Play the results
implay(uint8(compressed_video));
```

When we performed compression on images, the results just looked like lower resolution images. The video compression results appear to have a similar characteristic. So far, this makes sense, but it doesn't create any interesting effects to the video like we wanted. However, if you bump up the compression to $k = 20$, then the video doesn't just look like it is lower resolution, but it is also blurry (almost as if someone used the smudge tool in Photoshop on it). This is because we are compressing frames individually and not relating them to each other. The next warm-up exercise will help you

explore how we can apply the same SVD techniques across the time dimension.

Videos can be found at:

https://github.com/NiceanLoH/videoData/blob/master/compressed_video_20.zip

https://github.com/NiceanLoH/videoData/blob/master/compressed_video_50.zip

Appendix A.4 Warm-Up Solution #4

For the thought experiment, the pixel at (0, 0) is white for all time, so we can use the value 0 to represent it. Similarly, a pixel in the same row as the black box is white almost all of the time, except it is black for one time instance. Therefore, we can just use a pixel value of 0 to represent it, and we would be write most of the time.

The matrix we are trying to form looks like this:

$$\begin{bmatrix} 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & \dots & 0 & 0 \end{bmatrix}$$

It is a matrix with a small identity matrix in the center that is surrounded by zeros all around. If we look at each column of this matrix, it is either all zeros, or it is zeros everywhere except one element is one. If we can only use one vector to represent these vectors, we would choose a vector of all zeros, because placing a single 1 in any position would only be correct for one column and wrong for all others.

Notice that this matrix has reduced an entire video to just two dimensions – position along the columns and time along the rows. Applying the SVD techniques to this matrix will do what we have been doing in this warm-up exercise – it tries to use a single vector to represent several positions in the image across time. These vectors may not be all ones or zeros like we have been constructing. Think about positions in the image whose *change* over time is similar. The background pixel positions are all subject to the same change over time (lighting, etc). So, a single unit vector can represent that normalized change. Then each pixel is just a scalar multiple of that vector. So what does using the SVD to find a low-rank approximation to this matrix do? Simply, it tries to represent the matrix using only a few basis vectors. The number of basis vectors is the determined by the rank chosen for the low-rank approximation.

Appendix A.5 Lab Solution #1

The following code will provide an example solution to what we ask for in the lab's first section.

Lab1.m

```
% Importing the video
v = VideoReader('xylophone.mp4');
videoMatrix = v.read;
% Constants pulled from the video
numberOfFrames = v.Duration*v.FrameRate;
```

```
videoDim1 = size(videoMatrix,1);
videoDim2 = size(videoMatrix,2);

greyVideo = zeros(videoDim1, videoDim2, numberOfFrames);

% Convert from rgb video to gray video, just as a matter of course
for ii=1:numberOfFrames
    greyVideo(:,:,ii) = rgb2gray(videoMatrix(:,:,:,ii));
end

implay(uint8(greyVideo));

% Flatten the video in the standard way with reshape
flattenedVideo = reshape(greyVideo,[videoDim1*videoDim2,
   numberOfFrames]);

% Reshape the video to ensure that this worked
greyVideo2 = reshape(flattenedVideo,size(greyVideo));
implay(uint8(greyVideo2));
```

We convert the video into a matrix that has dimensions 76800×141. Doing the reverse (141×76800) produces results which look more like doing a low-rank approximation for every frame.

Video results can be found at:

https://github.com/NiceanLoH/videoData/blob/master/greyVideo.zip

Appendix A.6 Lab Solution #2

The following code will provide an example solution to what we ask for in the lab's second section.

Lab2.m

```
% Econ svd is required here, unless you like matrices that are
   around 50 GB
% total stored on your computer.
[U,S,V] = svd(double(flattenedVideo),'econ');

% Reconstructing the video
reconstructedVideo = uint8(reshape(U*S*V',[videoDim1,videoDim2,
   numberOfFrames]));

% Find a low-rank approximation
singularValues = diag(S);
largeSVs = singularValues;
largeSVs(7:end) = 0;

% Recreate the videos. Use implay() to play them back in MATLAB
largeSVVideo = uint8(reshape(U*diag(largeSVs)*V.',[videoDim1,
   videoDim2,numberOfFrames]));
```

It is necessary to take the econ SVD in this section, because to do otherwise will produce matrices with size 76800×76800, and that combined with a double variable size of 8 bytes gives us a matrix

that takes up around 50 GB of data. And that's just U. As-is, the U, S, and V matrices are of size 76800×141, 141×141, and 141×141 respectively.

What happens is that by taking the SVD and doing a low-rank approximation, we are basically trying to express our video with as few variations between frames as possible, since each vector is in reality a whole frame. The counterintuitive thing that you have to wrap your mind around is that each basis vector is no longer a vector, but an entire image. So the best rank-1 approximation is the outer product of the background and an all-ones vector.

Alternatively, movement represents pixel value variations when you travel along the time dimension. A low-rank approximation ignores small variations, and if there's a relatively small amount of movement in the video, like this one, then it is ignored by the low-rank approximation.

So we expect for a video with a non-moving camera and something that could be called a background, that the low-rank approximation will be the background of the image, with much of the moving parts removed. It's the low-frequency in the time dimension components.

The central point of this lab is that we have structured the data in such a way that we get interesting results. If we put the time dimension on a different axis, you get different results. How might you, the reader, structure your data in order to pull out patterns that you want? This lab is a demonstration of how important that question is (along with the SVD stuff).

Video results can be found at:

https://github.com/NiceanLoH/videoData/blob/master/largeSVVideo.zip

Appendix A.7 Lab Solution #3

The moving objects remain in the data, but they are not expressed because we set the singular values that correspond to their singular vectors to be zero. If we set the large singular values to zero instead of the relatively small ones, we'll pull out the high-frequency components. That is, the movement will come back, and the background will disappear.

The following code will provide an example solution to what we ask for in the lab's third section.

Lab3.m

```
% Zero out only the largest singular values and examine the results.
smallSVs = singularValues;
smallSVs(1:5) = 0;

smallSVVideo = uint8(reshape(U*diag(smallSVs)*V.',[videoDim1,
    videoDim2,numberOfFrames]));

% What's the best rank-1 approximation to a video? A constant image!
singleSV = singularValues;
singleSV(2:end) = 0;
topSVVideo = uint8(reshape(U*diag(singleSV)*V.',[videoDim1,videoDim2
    ,numberOfFrames]));
```

Video results may be found at:

https://github.com/NiceanLoH/videoData/blob/master/smallSVVideo.zip

Appendix A.8 Lab Solution #4

In order to produce something that will work nicely, we need to reshape the image in such a way that it will resemble a 1-frame video. So it should be "flattened" to have dimensions 76800×1. So U, S, and V will have dimensions 76800×1, 1×1, and 1×1 respectively. This makes sense, because essentially U *is* our image, but normalized. If we wanted to make this a video, we would just convert V to be a matrix of all ones that was 141×141, say, and multiply our S scalar by the 141×141 identity matrix.

When replacing the left singular value, just replace the first left singular value by the U vector we found earlier. The top S value should be replaced by the singular value we just found, and we can replace the first right singular vector by a vector of all ones.

This will mean that what we have made is no longer a proper SVD, but we weren't out to do that anyway. We are producing a particular video editing effect, not do a particularly rigorous transformation.

The following code will provide an example solution to what we ask for in the lab's final section.

Lab4.m

```
% Our new background has too many pixels, so we downsample along
    both
% dimensions.
newBackground = imread('AT3_1m4_03.tif');
newBackground = downsample(downsample(newBackground,2)',2)';

vectorNewBackground = reshape(newBackground,[videoDim1*videoDim2,1])
    ;
[Ux,Sx,Vx] = svd(double(vectorNewBackground),'econ');

U(:,1) = Ux; V(:,1) = ones(numberOfFrames,1);
S_prime = singularValues; S_prime(1) = Sx;

newBackgroundVideo = uint8(reshape(U*diag(S_prime)*V.',[videoDim1,
    videoDim2,numberOfFrames]));

implay(newBackgroundVideo);
```

The result should be the xylophone keys and the hand playing them, superimposed on the top of our image, which in this case is a microscope slide image with some cells.

The video may be found at:

https://github.com/NiceanLoH/videoData/blob/master/newBackgroundVideo.zip