ECE532 – Final Project

Kyle Daruwalla
daruwalla@wisc.edu

Davis Gilton
gilton@wisc.edu

Shuoyand Qiu
sqiu26@wisc.edu

15 Dec. 2016

Executive Summary

This short section (about 1/2 page) should clearly explain what the project is about, which machine learning techniques are explained and utilized, and what a reader can expect to learn and accomplish if they follow along and work through the examples and problems. This is a very important part of your project, because its the first thing the reader sees. Be clear, be concise, and use this opportunity to convey what makes this project interesting/unique/cool.

0.1 Background

This section should explain the context for the main idea behind the project (you may divide this section into multiple subsections if it helps with readability). Start with a description of the problem that will be solved, a brief history (with citations) of how the problem came about, why its important/interesting, and any other interesting facts youd like to talk about. Feel free to use images/diagrams/etc. from research papers, the internet, or other sources to help with your explanation, so long as you cite your references. Finally, the background section should also give a mathematical description of the problem, with equations as needed.

0.2 Warm-Up

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

0.2.1 Compressing image data

From class, you will remember the Eckart-Young Theorem

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. Note that X is also the solution to the problem

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

We can use this concept to find low-rank approximations to 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. Solutions are included in Appendix A.1.

0.2.2 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;
```

0.2.3 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.

0.2.4 Reshaping video data

This problem is a simple thought experiment. First consider a video frame capturing a series of black (pixel value 1) and white (pixel value 0) stripes moving across the screen as shown in Fig. 1. Each stripe in the example is a single pixel wide. Now focus on a single pixel (e.g. the pixel at (0, 0) in the top left corner). At time $t = 0$, the pixel has value 1, because there is a black stripe across it. At time $t = 1$, the black stripe moves to the right, and a white

stripe takes its place, so the pixel has value 0. This pattern continues infinitely as long as there are stripes moving through the viewing window.

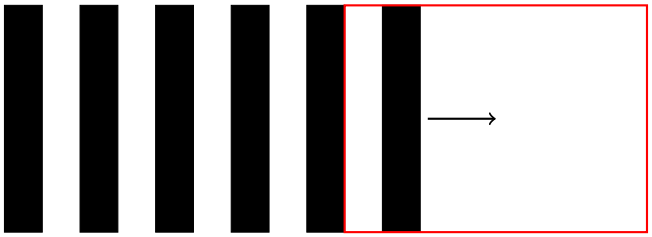

Figure 1: The red box is the viewing frame of the video capture device. The series of black and white strips moves from left to right under the viewing window.

Problem: Imagine the pattern continues for time T steps. Over that interval, you can only use one value to represent the value of the pixel you have been focusing on. What value do you choose? Now suppose you can choose a $T \times 1$ vector where the element at index i represents the value of the pixel at time $t = i$. What vector do you choose? Will this vector be correct if you focused on the pixel adjacently to the right (i.e. pixel at (0, 1))? If you can choose only one vector to represent the values of both pixels over time, what vector do you choose? (*Hint:* think back to your answer for the first question in this problem). Solutions included in Appendix A.4.

Now consider a different example. We have the same viewing window, but instead we are sliding a 1px $\times$ 1px black square across the screen (see Fig. 2 and ignore the numbers for now).

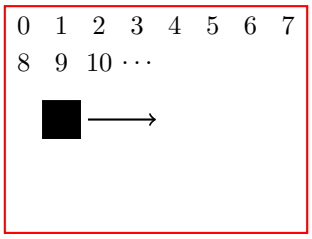

Figure 2: The red box is the viewing frame of the video capture device. The black square moves under the viewing frame from left to right.

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. What single value represents its value over all time? (*Hint:* don't over think it or think about actually calculating the average). Now suppose you vectorize the 2D viewing window – count across the pixels in the frame as shown in Fig. 2

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 like the vectors we were working with in the previous problem – the values of a signle pixel across all values of time. What would this matrix look like for the experiment in Fig. 2? You can now choose a *column* vector, like we previously did, to represent all the columns of this matrix. What vector do you choose? (*Hint:* this is just like choosing a single vector for two pixels like we did before – except we are looking over N pixels instead of just two).

0.3 Lab

0.3.1 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 1 to read in the xylophone.mp4 video and convert it to greyscale. After Warm-Up problem 3, 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 greyscale video matrix into a 2-D matrix.

(*Hint:* ensure that the resulting 2-D matrix has the same number of rows as there are frames in the video, and 320× 240 rows)

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

0.3.2 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 Question 2 of our Warm Up, 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 problem 1.

Should you take the econ 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 part 2, 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?

0.3.3 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 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.

0.3.4 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 greyscale 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?

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;
```

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
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 appear to target low frequency or high frequency portions of 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.

Appendix A.4 Warm-Up Solution #4

For the first experiment, try plotting the value of the pixel over time. It should look like a square wave that is flipping between 1 and 0. Suppose the square wave goes from 1 to 0 to 1 to 0. What would be the integral of this square wave over this time interval? It should be the average value – 0.5. Therefore, if you

could only use a single value to represent the pixel's values over time, it would be 0.5.

If you could use a $T \times 1$ vector, the obvious solution is to use the pixels value. For example, if $T = 4$, then the vector would be

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

But this vector would not represent the pixel at (0, 1) well. The vector for that pixel would be

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

So the vector we chose initially would be incorrect would be wrong for every element. We are trying to represent the how two pixels are changing over time with a single vector. Therefore, it might be a better idea to use the average value of those two pixels. A vector of where every element is 0.5 would be equally correct for both pixels.

For the second 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.