17.10.2018

Singular Value Decomposition (SVD) used in image processing

ELE510 Project

Henrik Mjaaland
UNIVERSITY OF STAVANGER

Summary

The scope of this paper is to explain what SVD is and the theory behind SVD, and to show how and why to apply it in image processing by performing experiments. I used SVD for two types of image processing (image compression and image denoising) to demonstrate how it works and its appliances. All the code I wrote for the experiments and tests in this project is written in MATLAB.

Contents

1.0 Introduction

SVD was discovered independently by multiple mathematicians (at least three), but it was first discovered in 1873 by Eugenio Beltrami. It was however not proven before 1936 by Carl Eckart and Gale Young. In 1954 the first algorithm for SVD was invented by the Armenian-American mathematician, Kogbetliantz.

SVD is a form of matrix diagonalization. Matrix diagonalization is performed on a matrix, A, by multiplying A with two matrices B and C, where the resulting product is congruent to the diagonal. It is the same as finding a matrix's eigenvalues.

SVD has many appliances. To mention some of the many uses SVD offers, it can be used for compression and image processing (as I have done in my experiments) or computing the pseudoinverse of a matrix. Image processing is a method used to improve the quality of a digital image by for example denoising or stretching an image. SVD has also been used by many renowned brands like Netflix, Youtube and Google for Collaboratorive Filtering (a technique for predicting user preferences, e.g. user preferences for different movies).

SVD can also be used to compress an image. This is done by first applying SVD on a matrix of image points, and then discarding the singular values with the least information (the singular values are in descending order). As mentioned SVD can also be used for denoising. The concept is the same as for denoising (only the largest singular values are used). I will get more into detail about these techniques in chapter 3 which is about the implementation.

Chapter 2 of this paper explains the theory behind SVD, chapter 3 explains my implementations of SVD denoising and compression, chapter 4 proposes ideas for future work, chapter 5 is the conclusion followed by an appendix which includes a user manual and code, and finally there is a bibliography.

2.0 The Theory Behind SVD

2.1 SVD Parameters

A dimension reduction technique seeks to reduce the number of random variables in question. SVD is a dimension reduction technique in the sense that the smallest singular values are neglected or pruned.

$$A_{[m \times n]} = U_{[m \times r]} \Sigma_{[r \times r]} (V_{[n \times r]})^T$$

Figure 1: SVD Formula (Tran, 2018)

The input is a matrix of size m x n (m rows and n columns), and the result which is the SVD is the product of three matrixes, namely U, ∑ and V transposed as you can see in figure 1 above. U is of size m x r, ∑ is of size r x r and V is of size n x r. This product is called the singular value decomposition.

U contains the left singular vectors (orthonormal eigenvectors of A*A^T), ∑ contains singular values (square roots of the nonzero eigenvalues in A^T*A and A*A^T), and is a diagonal matrix meaning that every non-diagonal element in the matrix is zero. The nonzero elements in the diagonal are the singular values and they are sorted in decreasing order. The matrix V contains the right singular

values (orthonormal eigenvectors of A^T*A). The number of columns, r is usually a small number and it is basically the rank of the input matrix, A. The SVD theorem says that it is always possible (it is possible for any matrix) to decompose a real matrix A (a real matrix is a matrix where all the elements are real numbers) into the product of the matrices U, ∑ and V transposed.

2.2 SVD Properties

U, ∑ and V are unique, meaning that there is only one solution. These three matrixes have the following constraints. The matrices U and V are column orthonormal. This means that the columns in U and V are orthogonal which means that the dot product of the columns in U and V is zero, and that the columns of U and V has Euclidean length 1 (the sum of the squared values in the columns of these two matrices is 1). As said, ∑ is diagonal meaning that all the non-diagonal elements are zero, while the diagonal elements are all positive (and they are sorted in a decreasing order).

2.3 SVD Method

I will use Collaboratorive Filtering as an example to explain the SVD method and how it can be applied. The following example is user-based. Consider an input matrix, A, where the rows represent a set of users, while the columns in A represent a set of movies. Each element in A is a rating of the movie represented by the given column by the user represented by the given row.

The objective of the problem is to group the movies into different categories based on the users' ratings (item-based collaboratorive filtering). For example, if two columns (movies) has the same kind of values/ratings for a certain row (user), this means that most likely these two columns are in the same category, e.g. sci-fi.

However, instead of putting each movie into one category, they will be put in several categories each of varying degree. Each movie can be in terms of linear algebra be represented as Mi = x1*1+x2*2+…+xj*j, where i is the column number, j is the category number, and xj is a constant. To separate the movies into different categories, the basis vectors must be orthonormal to each other.

One might also want to put users (rows) in categories based on their ratings (user-based collaboratorive filtering), e.g. sci-fi lovers. Now, the user category, sci-fi lovers, should match the movie category, sci-fi. The sci-fi category will be represented by the same unit vector as the sci-fi-lovers category. Each user category can also be represented in terms of linear algebra as Ui = y1*1+y2*2+…+yj*j, where i is the user number, yj is a constant, and j is the user number.

The ratings can be estimated by multiplying M and U transposed, R = M*U^T. This is the same as multiplying U and M with singular values. Since the singular values (the diagonal elements in ∑) are in descending order, one can prune the movie or user categories that correspond with the lowest/last singular values. This prevents movies or users being put into categories that corresponds to low ratings. However, if the data is not very linear, SVD might not have the best results. Also, it can be hard to analyse the results.

3.0 Experiments and Results

3.1 Choice of IDE

For my implementations I decided it was best to use MATLAB. One of the main reasons I chose MATLAB for my experiments is because SVD is a product of matrices, and MATLAB is designed to operate on matrices (all MATLAB variables are two-dimensional vectors or matrices). MATLAB also has a lot of toolboxes (libraries) with built-in functions. The image processing and computer vision

toolboxes are especially large. You can open these toolboxes inside MATLAB to view the code and even edit them to as you see fit. Other benefits with using MATLAB is that you can for example execute portions of code at a time, and the 'help' function is also useful for information on MATLAB functions.

3.2 Image Compression using SVD

Compression is the use of algorithms that reduce the size of a file; most files has a lot of redundant content. Since compression reduces the file size, it reduces both the transmitting (upload time) and the receiving time (download time) for sending data when files are sent. It also reduces the necessary storage space, which in turn allows to store files with better quality, because the higher the quality of a file, the more space it will consume (e.g. resolution of an image).

Two different image compression techniques are lossless compression technique and lossy compression technique. As the names suggests, original data can be reconstructed fully from compressed data when using lossless compression, but not when using lossy compression. Even though some data will be lost when using lossy compression, there are still instances where one would prefer lossy compression because it can reduce file sizes more than lossless compression.

Lossy compression is usually used for sound or video because it is hard to notice data loss in those cases. Also, lossy compression reduces mostly redundant data. Lossless compression is not final, while lossy compression however is final. Lossless compression should be used when it is important that the uncompressed file does not deviate too much from the original file. It is used for example in the zip file format.

When it comes to images, neither of the two types is best, because it depends on the image format. For example, for .png images it might be best to use a lossless compression algorithm, because you might want to further edit/analyse it. However, for .jpeg, lossy compression algorithms are used. Image compression is used for digital images.

In general, compression algorithms initially create a statistical model for the given data, then the model is used to remove the most frequent data (e.g. letters) when compressing. When the data is uncompressed afterwards, the removed data is retrieved. Some of the most common lossless compression algorithms are Huffman, arithmetic coding and PPM (prediction by partial matching). PPM ranks symbols based on previously used symbols. In this case the ranks are the same as pdf estimations.

Image compression with SVD also uses ranks, in this case however, the ranks are the number of kept (not neglected) singular values. Image compression using SVD is a type of lossy compression. A rank in correspondence with SVD is the number of singular values that are included in the truncated SVD. A truncated SVD is just an SVD that is missing one or more of the last singular values. For example, for compression, the file size increases as the number of singular values increases, but so does the quality of the file.

An image, I, is usually not symmetric, which is why we create a symmetric matrix from the image, which is the following product, I*I^T. Now we can get Singular Value Decomposition from I*I^T, and we can show that if I*I^T is a matrix of rank r, matrix I can be written as I = U*∑^0.5*V^T, where U and V are orthogonal matrices of size N x r and ∑^0.5 is a diagonal r x r matrix (Maria Petrou, 2018).

When using SVD for image compression it is critical to minimize the number of rankings so that the file size is reduced as much as possible, while at the same time making sure that data loss does not

get out of hand. I implemented image compression for both grayscale images ('SVD_Compression_Grayscale.m') and coloured images ('SVD_Compression_Colour.m').

In my implementation of compression for coloured images I called the built-in MATLAB function, svds(A,k), for each of the three colour planes, red, green and blue. Then I combined the three SVDs to a compressed image. A is the matrix for the input image, and k is a given approximated rank. For the grayscale implementation, it is only necessary to call svds(A,k) once, since there is only one colour (grey). For this implementation, all the compressed images look the same for any rank.

The images in the figures below (figure 2-6) demonstrates how the quality of the compressed image changes as the rank changes. The first image displays the original image. The next figures display the compressed image with different rankings (10, 20, 30, 40, and 50). As you can see, the quality is decreasing as the rank decreases, and at a rank of 10 there is a severe degradation of quality, which means a higher loss of data when decompressing the image, but also that the file compression rate (ratio between uncompressed size and compressed size) is very high.

Even though SVD yields high compression rates, it is computationally slow because it involves a lot of operations. This is noticeable when running my programs; the running time is quite high. Thus, it is important to use approximations and randomized algorithms (algorithms where one or more decisions are based on random numbers) instead of exact computations for SVD.

Figure 2: Original image before compressing

Figure 3: Compressed image with a rank of 10

Figure 4: Compressed image with a rank of 20

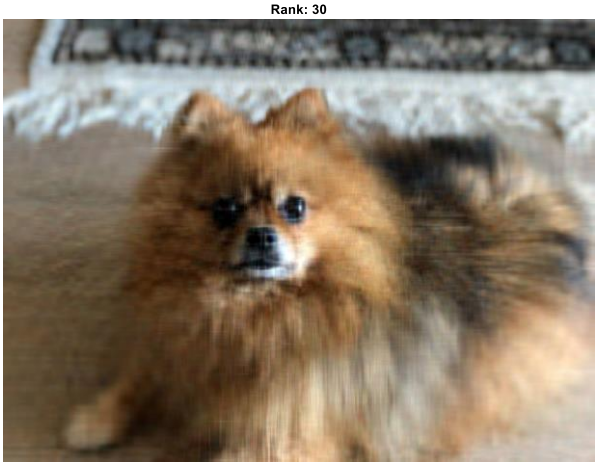

Figure 5: Compressed image with a rank of 30

Figure 6: Compressed image with a rank of 40

Figure 7: Compressed Image with a Rank of 50

3.3 Image Denoising using SVD

Image denoising is the process of reducing noise in a noisy image. There are different types of noise, e.g. Gaussian noise, Poission noise, salt & pepper noise etc. The noise I used to denoise the example image in my implementation of image denoising is 'Salt & Pepper'/impulse noise. It is characterized by scattered noise in the form of white/black spots. It is caused by sudden disturbances.

SVD can as said be used for image denoising. First, I chose an example image to denoise. The image I used didn't have any noise so in my implementation I added noise using the MATLAB function Imnoise with the type of noise explained above. I also had to convert my image from colour to grayscale, because my implementation is for grayscale images.

After converting the image from colour to grayscale, I partitioned the grayscale image into patches. The more patches that are included, the higher quality the compressed image will have, but also more noise and bigger size. Thus, one should choose the number of patches so that the image is denoised to an acceptable degree, whilst at the same time not losing too much sharpness. I set the number of patches to 5 (5 rows and 5 columns), which reduced the noise considerably and the image still had a decent quality. Also, the running time is not that long for 5 patches (running time increases with the number of patches). I then applied SVD to all the resulting chunks.

I used the eigenvalue decomposition of the chunks to find their SVDs. As mentioned, every matrix has a singular value decomposition, but this is not true for eigenvalue decompositions. Only symmetric matrices have an eigenvalue decomposition. Therefore, I multiplied the input image with the input image transposed and used that to find the SVDs as explained in 3.2.

For a given matrix, A, v is the eigenvector and λ is the eigenvalue if $A*v = \lambda*v$. Also, if v is an eigenvector then v*x is also an eigenvector and $\lambda*x$ is an eigenvalue. The eigenvalue(s) is computed by expanding $\det(A-\lambda*I) = 0$, and solving for λ. The eigenvectors are found by computing $\det(A-\lambda*I)$.

I set the number of singular values for each patch to be 10. The number of singular values should be minimized as to prune as much noisy pixels as possible, but at the same time not so small that if affects the image quality.

The singularity value decompositions are in my implementation of image denoising denoted by U*prunedS*255*V', where U is the left singular vectors, V' is the right singular vectors transposed,

and prunedS is the eigenvalues. U is orthogonal (U^-1=U^T the columns are orthonormal, the numbers are real, and the columns are normalized eigenvectors). These SVDs are the same as the reconstructed image patches because any symmetric matrix can be reconstructed using its eigenvalues and eigenvectors according to the eigen decomposition theorem. The eigen decomposition theorem says that if a square matrix, A, is symmetric and P is a square matrix, then A can be written as an eigen decomposition, A = PDP^-1, where P is a matrix of eigenvectors, and D is a diagonal matrix with the eigenvalues on the diagonal (Wolfram Alpha, 2018).

In the end, after SVD has been applied to the groups, the image is reconstructed by aggregating the groups into one image. The figures below show how the image changes throughout the process and finally the denoised image. Figure 8 is the original coloured image. Figure 9 is the grayscale image. Figure 10 is the grayscale image with noise added ('Salt & Pepper'). Finally, figure 11 shows the aggregated denoise image.

Figure 8: Original Image with Colour

Figure 9: Original Image in Grayscale

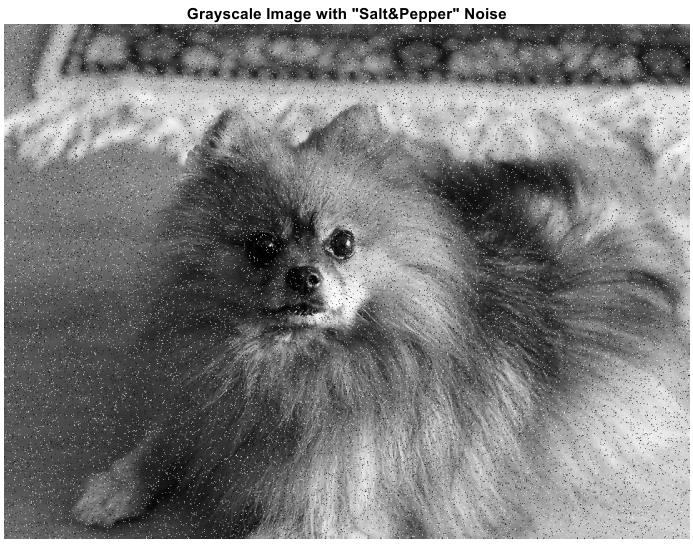

Figure 10: Original Image in Grayscale with Salt & Pepper Noise

Figure 11: Partitioned Noisy Grayscale Image

4.0 Future Work/Improvements

4.1 Introduction

I have left several experiments and tests for future work due to a lack of time. Below follows a couple of examples of how my work can be further improved (I have only shown a couple of SVD's many appliances, so you can also find and implement other appliances for SVD, like for example finding the pseudo-inverse of a matrix or finding the nearest orthogonal matrix.

4.2 Image Decompression using SVD

I only compressed (reduced file size of) coloured and grayscale images. Decompression can be implemented by building on my work, because decompressing images (restoring compressed images)

usually just requires one to reverse the compression process. Also, one should try to reduce the data loss after decompression as much as possible.

4.3 Denoising for Coloured Images

In grayscale images it is usually easier to separate edges from other features in an image. Colours in an image increases the required complexity of the denoising model to be used, thus increasing the running time and the amount of coding or work in general. The latter is the main reasons why grayscale has been preferred in the past (80's and early 90's), as opposed to colours for image processing. However, today the complexity is not an issue when it comes to performance, because we have a lot more advanced and powerful CPU's now. Also, for some features it might be better to use coloured images than grayscale (for example one can separate objects based on colours). Therefore, I suggest implementing denoising for coloured images as well.

4.4 Error and Quality Estimations

Measure the compression rate and the compression quality using MSE (Mean Square Error). Also, compare the quality of the original image with the quality of the denoised image using MSE again.

4.5 SVD Function

As mentioned, I used a MATLAB built-in SVD function (svds) for the compression. You can build on my implementation of SVD-based compression, by developing your own SVD function from scratch. This is mostly for educational purposes, but you can of course adapt it to your own liking.

5.0 Conclusion

5.1 SVD

SVD is the resulting product of factorizing a matrix, A, into three matrices, U, D and V transposed. The matrices, U and V are orthogonal, and they contain the left and right singular values. The matrix, D, is diagonal and contains the singular values. The values in the matrices are positive real numbers.

5.2 Appliances

SVD has many appliances. Two of its many appliances is compression and denoising, which I have implemented. SVD is efficient because the singular values are in descending order, which means that insignificant pixels in an image can be pruned. On the other hand, SVD is not the best for non-linear data.

Appendix A

A.1 - User Manual

Make sure that the image, 'max_original.jpg', is in the folder images before running the programs. As the names suggest, 'SVD_Compression_Grayscale.m' compresses the grayscale version of 'max_original.jpg', 'SVD_Compression_Colour.m' compresses the original coloured version of 'max_original.jpg', and 'SVD_Denoising.m' denoises 'max_original.jpg'.

A.2 – Code for 'SVD_Compression_Grayscale.m'

```
clear;close all;clc;

img = imread('images/max_original.jpg');
```

```
doubleImg = double(img);
figure(1);
imshow(img)
title('Original Image')
img_gray = rgb2gray(doubleImg);

rankings = [10,20,30,40,50];
for i= 1:length(rankings)
    [U,S,V] = svds(img_gray,rankings(i));
    SVD = U*S*V';

    figure(i+1);
    imshow(uint8(SVD))
    header=(['Rank: ',num2str(rankings(i))]);
    title(header)
end
```

A.3 – Code for 'SVD_Compression_Colour.m'

```
clear;close all;clc

img = imread('images/max_original.jpg');
doubleImg = double(img);
figure(1);
imshow(img)
title('Original Image')

rankings = [10,20,30,40,50];
for i= 1:length(rankings)
    %Convert all the rows and columns in the first colour plane in
    %img_gray from intensity to double
    red = doubleImg(:,:,1);
    %Convert all the rows and columns in the second colour plane in
    %img_gray from intensity to double
    green = doubleImg(:,:,2);
    %Convert all the rows and columns in the third colour plane in
    %img_gray from intensity to double
    blue = doubleImg(:,:,3);

    [U,S,V] = svds(red,rankings(i));
    compressedRed = U*S*V';
    [U,S,V] = svds(green,rankings(i));
    compressedGreen = U*S*V';
    [U,S,V] = svds(blue,rankings(i));
    compressedBlue = U*S*V';

    CompressedImg(:,:,1) = compressedRed; %Red = first image plane
    CompressedImg(:,:,2) = compressedGreen; %Green = second image plane
    CompressedImg(:,:,3) = compressedBlue; %Blue = third image plane

    figure(i+1);
    imshow(uint8(CompressedImg))
    header=(['Rank: ',num2str(rankings(i))]);
    title(header)
end
```

A.4 – Code for 'SVD_Denoising.m'

```
clear;close all;clc;
```

```
figure(1);
img = (imread('images/max_original.jpg'));
imshow(img);
title('Original Image');

img = im2double(img);
imgGray = rgb2gray(img);
figure(2);
imshow(imgGray);
title('Grayscale Image');

figure(3);
%Add 'salt%pepper' noise to grayscale image:
imgNoise = imnoise(imgGray, 'salt & pepper');
imshow(imgNoise);
title('Grayscale Image with "Salt&Pepper" Noise');

%Patch Grouping:
[rows, columns] = size(imgNoise);
nPatches = 5; %number of patches/singular values
patch = zeros(nPatches^2,rows/nPatches,columns/nPatches);
numRows = size(patch,2);
numCols = size(patch,3);

idx = 1;
for i = 0:nPatches-1
    for j = 0:nPatches-1
        for k = 1:size(patch,2)
            for l = 1:size(patch,3)
                patch(idx,k,l) = imgNoise(i *numRows + k , j*numCols + l);
                imgVals(k,l) = imgNoise(i *numRows + k , j*numCols + l);
            end
        end
        idx = idx + 1;
    end
end



% ------ SVD on each patch

%images are usually not symmetric, therefore, calculate SVD for img x img
%transposed
symmetricImg = imgNoise'*imgNoise;
[V,eigVals] = eig(symmetricImg);
S = sqrt(eigVals);
Vimg = imgNoise*V;
U = zeros(rows,columns);
for i = 1:columns
    U(:,i) = Vimg(:,i)/S(i,i);
end

denoisedImg = zeros(size(patch));
%Number of singular values for each of the patches:
nRanks = 10;

for i = 1:nPatches*nPatches
    for j = 1:numRows
        for k = 1:numCols
            imgVals(j,k) = patch(i,j,k);
        end
```

```
    end
    symmetricImg = imgVals' * imgVals ;
    [V,eigVals] = eig(symmetricImg);
    U = zeros(numRows,numCols);
    S = sqrt(eigVals);
    Vimg = imgVals * V;
    for j = 1:numCols
        U(:,j) = Vimg(:,j)/S(j,j);
    end
    prunedS = zeros(numCols,numCols);
    for j=size(S,2)-nRanks:size(S,2)
        prunedS(j,j) = S(j,j);
    end
    denoisedImg(i,:,:) = U*prunedS*255*V'; %SVD (U*S*V^T)/denoised image
end

idx = 1;
aggImg = zeros(numRows*(nPatches-1)+numRows, numCols*(nPatches-1)+numCols);
for i = 0:nPatches-1
    for j = 0:nPatches-1
        for k = 1:numRows
            for l = 1:numCols
                %Aggregation: Im_rec aggregated denoised image.
                imgVals(k,l) = denoisedImg(idx,k,l);
                aggImg(numRows*i+k, j*numCols+l) = denoisedImg(idx,k,l);
            end
        end
        idx = idx + 1;
    end
end

figure(4);
imshow(aggImg, []);
title('Aggregated Denoised Image')
```

A.5 Image, 'max_original.jpg'

References

Maria Petrou, C. P. (2018). 2.1 Singular value decomposition. In C. P. Maria Petrou, *Image Processing, The Fundamentals* (p. 51). Stavanger: Wiley.

Tran, V.-T. (2018, 05 10). *pca and svr*. Retrieved from slideshare: https://www.slideshare.net/microlife/4-pca-and-svr

Wolfram Alpha. (2018, 10 08). *Eigen Decomposition Theorem*. Retrieved from Wolfram Alpha: http://mathworld.wolfram.com/EigenDecompositionTheorem.html