

# Recognizing Images of American Sign Language Letters using Principal Component Analysis

Ellen Walker  
Computer Science Dept  
Hiram College  
Hiram, OH 44234  
walkerel@hiram.edu

## 1. Overview

Digital images are now ubiquitous and easy to acquire. While humans easily recognize the objects and other semantic content in images, it has been much more difficult to do so automatically. Images of the same object can vary significantly because of lighting, slight differences in orientation, shadows and camera parameters. Even when the set of objects is limited and reference images are available, direct comparison of images on a pixel-by-pixel basis is unlikely to yield satisfactory recognition. Higher-level semantics are required.

To obtain higher-level semantics, a set of *features* must be computed from each image, and these features are then compared to the features of the reference image. Finding the right set of features by trial and error can be time-consuming and difficult. Therefore, in this project, we will build a system to learn an appropriate set of features from a training set of images, and then apply them to a test set of images. The methodology used in this project has been used successfully for face recognition.

## 2. Objectives

The aim of this project is to develop an interpreter for images of American Sign Language letters. The test and training sets will consist of relatively high-contrast images of hands signing each of the 26 letters, and the goal of the project will be to successfully classify all of the images in the test set.

The learning objectives of the project are:

- Learning the basics of Object Recognition in Computer Vision
- Becoming familiar with the concepts of feature spaces and classification
- Familiarity with the techniques of Principal Component Analysis
- Experience analyzing experimental results

### 3. Background: Understanding Classification

In this project, you will classify images of sign language characters based on the characters portrayed (i.e. all A's will be 1 class, all E's will be another, etc). Each class will be represented by a *feature vector* consisting of values for a number of features. When a new image is presented to the system, the feature vector of the new image is computed and compared to the predefined feature vector for each class. The unknown image is given the label that corresponds with the closest predefined feature vector.

To illustrate how classification works, consider the following table of features for the first 10 capital letters of the English alphabet. The features all have integer values: the number of holes (entirely enclosed spaces) in the letters, the number of endpoints (single segment endings, not angles), the number of straight segments, and the number of curves.

Letter	Holes	Endpoints	Straight Segments	Curves
A	1	2	3	0
B	2	0	1	2
C	0	2	0	1
D	1	0	1	1
E	0	3	4	0
F	0	3	3	0
G	0	3	2	1
H	0	4	3	0
I	0	2	1	0
J	0	2	1	1

To use this table, each new character would be analyzed according to the four features, and classified as the letter whose vector is closest to its own vector.

Using this feature space, let's classify the character:  $\emptyset$ . This character has 2 holes, 2 endpoints, 1 straight segment, and 1 curve, i.e. a vector of (2,2,1,1). We'll use a fairly standard distance measure, which is the sum of the squares of the distances between each feature value and the test value. The distance to the letter A (1,2,3,0) is  $1+0+4+1$ , or 6. Of the letters listed above, the closest is J (0,2,1,1) whose distance is  $4+0+0+0$ , or 4. Therefore, using our feature set and models, the character  $\emptyset$  would be classified as J.

The classification process is only as good as the set of features. In our example, J is probably not the character you would choose as closest to  $\emptyset$  – D might be a better guess, but the features do not agree. A good set of features should be easy to compute, differentiate all of the objects in the training set, and generalize well to new objects, i.e. objects that are qualitatively similar should have similar feature vectors.

## Exercises:

1. How would the letter V be classified? Explain.
2. If someone wrote a letter "A" but extended the crossbar beyond the left and right angled vertical segments, how would the modified A be classified? Explain.
3. If this classification system were extended to the full 26-letter alphabet, would all letters be distinguishable? Why or why not?
4. Considering your answers above, how might you modify this feature set to improve classification?
5. (Requires additional knowledge of image processing) Assume that letters are entered as binary images (black on white). Implement this simple classification system and test it on several images of capital letters.

As we have seen, developing appropriate features for classification is a difficult art. Therefore, we will use a different technique, *learning* the features from a set of training examples. Therefore, we have to acquire and preprocess some training images.

## 4. Phase I: Data Collection

### 4.1. Background

A digital image is simply a two-dimensional array of pixels (picture elements). In a color image, each pixel is represented as a vector of 3 values (usually red, green, and blue) that can be combined to form all visible colors. A black and white, (or more properly grayscale) image contains pixels that have a single value that ranges between the values of 0 and a maximum value, often 255. Increasing values indicate more light, so black is 0, and white is 255. There are many operations that can be performed on images to generate other images; these are called *image processing*. Even more difficult is extracting semantic information from images, i.e. labeling images or portions of images with the objects contained in the images. The field that encompasses this problem is called *computer vision*. In this project, we will touch only a small part of that field. For more information on image processing and computer vision, please refer to [CV references].

Digital images are stored in several forms. You might be familiar with TIFF, JPG, or PNG as some examples. Most of these image formats include some kind of compression, since images are very large. A convenient format for our purposes is the PGM format. The netpbm package ( <http://netpbm.sourceforge.net/> ) contains filters to transform many other formats to and from the PGM format. It also contains some simple programs that might be useful for transforming images, for example scaling image size. A simpler way to do this is to use the GIMP (

<http://www.gimp.org> ). The GIMP is a full-functioned image processing program along the lines of Adobe PhotoShop (but free). It is a visual image editor that can read and write PGM files as well as files of other formats, transform between image formats, and provides a large range of image processing tools that will help in preparing our images for this project.

## 4.2. Acquiring Images

For this project, you will need at least 10 images of each letter that you want to recognize. (More images will likely improve the performance of your system). The images should be carefully taken so that there is little or no information in the images other than the hand forming the letters, and the hand should take up a relatively large portion of the image. Our examples were done by a signer wearing a dark sleeved shirt on a dark background. Once you are set up, do not change the camera position, subject position, or lighting between images. (If you do, your system might “learn” these variations instead of the letters).

## 4.3. Preprocessing the Images

Now, you will use the GIMP, specifically the crop and scale tools. The tutorial at [http://www.gimp.org/tutorials/Lite\\_Quickies/](http://www.gimp.org/tutorials/Lite_Quickies/) will be helpful in learning to do this. Crop your images so that the hand forming the letter is centered, and takes up the majority of the image. It is important that the hand appears in the same position and the same relative size in all images, regardless of which letter is being signed. All images should be the exact same size. Then scale all of your images so that the pixel size is reduced below 50x50. You still need to be able to distinguish the hand shapes in your reduced images, but the images should be as small as possible to reduce the total number of pixels for later processing. Finally, use “save as” to save each image in PGM ASCII (not raw) format. If your image is originally in color, the GIMP will, after warning you, convert to grayscale and export to PGM. When you are done, there should be a new file <filename>.pgm, where <filename> was the name of the original file.

### Exercises:

1. Using a digital camera, acquire 30 images each of at least 5 different sign language letters (e.g. the vowels, as we did in our examples).
2. Process the images down to PGM images whose size is between 30x30 and 50x50 pixels. Crop and scale as appropriate to ensure that your images are recognizable sign language letters. All images need to be the same size.

## 4.4. Understanding the PGM Format (optional)

The PGM format is described at <http://netpbm.sourceforge.net/doc/pgm.html>. Essentially, it begins with the magic number P5 followed by the width and height of the image in decimal, followed by the maximum pixel value (often 255). The remainder of the file is the image data, row by row. Each row consists of W individual integers, where W is the width of the image. Integers range from 0 (black) to the maximum value (white).

### Exercises:

3. Write a program to double the size of a PGM image. Each pixel should be replicated 4 times. For example, the image on the left becomes the image on the right:  

1 2	1 1 2 2
3 4	1 1 2 2
	3 3 4 4
	3 3 4 4
4. Write a program that reads 2 images and writes the average image as a PGM file.
5. Write a program that reads N images and writes the average image as a PGM file. Use your program to create the average of your data images. This average image is needed to normalize your data before we can extract features from it. (Hint: instead of storing all N images, keep a “sum image” as you read in each image. Then, when writing out this image, divide each pixel value by the number of images being averaged).

## 4.5. Image manipulation in Octave

We can conveniently perform all of our computations in GNU Octave, which also has image display capabilities. GNU Octave is a free package for solving linear and nonlinear computations, comparable to Matlab. Octave can be found at <http://www.gnu.org/software/octave/>. It can be installed on Linux, Mac OS X, or Windows systems.

To allow Octave to work with PGM images, you will also need to install the Image package and software called ImageMagick (see links below). Once you have made these installations, you should be able to read and display images easily. Before proceeding, you should skim over the Octave manual (an HTTP version is at <http://www.gnu.org/software/octave/docs.html>) to familiarize yourself with the environment. At minimum, you should read Chapters 1 and 2, and experiment with some of the examples. You should also look over Chapter 8 on expressions, especially the first section on Index Expressions. You will also find Chapters 10 and 11 useful for learning to program in Octave. Octave deals very powerfully with matrices, and we will use this power extensively.

```
A1 = imread("A1.PGM"); # read the image into unsigned 8-bit array A1
A1      # display the pixel values on the screen
imshow(A1); #display the image in another window
```

You can also easily crop your images using array indexing. For example, if you only want to show the upper 20x20 pixels in your image, type

```
imshow(A1(1:20,1:20))
```

Since indexing ranges that are unbounded take the bounds of the matrix, you can also show half of the image:

```
imshow(A1(1:rows(A1)/2, :)) #shows top half
imshow(A1(:, 1:columns(A1)/2)) #shows bottom half
```

Octave can easily do mathematical manipulation of the images. For example, to create the average of three images, you can read them in, add them up, and divide by 3.

```
AvgImg = (img1+img2+img3)/3
```

The only problem with this, is that the average image is also an unsigned 8-bit image, so all the pixel values have been rounded to the nearest integer. This might be acceptable in some implementations, but if accuracy is needed, the images should be cast to doubles.

```
AvgImg=(double(img1)+double(img2)+double(img3))/3
```

Next, we can normalize our data by subtracting the average image from each original image. But, when we display this new image, there will be large chunks of black – zeros wherever the original value was less than 0. So to display a signed image correctly, we need to add 128 (half of the maximum value) back to the image, then cast the result back to unsigned 8-bit integers

```
imshow (uint8(img1-AvgImg+128));
```

Since we are likely to do this often, we can define a function that performs this calculation for us:

```
function showdata(img)
    imshow(uint8(img+128));
endfunction
```

However, this function assumes that the range of values is close to -128:128. If the values are really in the range -1:1, the image will be uniformly gray. We can generalize our function to show any array of data as an 8-bit image by including the maximum and minimum values in the image as part of the calculation:

```

function showdata(img)
    minval = min (min(img));
    maxval = max (max(img));
    printf("min is %f, max is %f\n", minval, maxval);
    imshow(uint8((img - minval) * (256/(maxval - minval))));
endfunction

```

To call this function, we can put any expression that generates a signed matrix in the parameter. To replicate our original example, we put `img1-AvgImg` as the parameter:

```
showdata(img1-AvgImg);
```

Octave is a complete programming language that includes conditionals and iteration. Chapters 10 and 11 of the user manual describe statements and functions in more detail. Here is one example from chapter 11.

```

function [max, idx] = vmax (v)
    idx = 1;
    max = v (idx);
    for i = 2:length (v)
        if (v (i) > max)
            max = v (i);
            idx = i;
        endif
    endfor
endfunction

```

The function's name is `vmax` and it returns two values (`max` and `idx`). It takes 1 parameter, a vector `v`. The variable `idx` (index of the maximum value) is initialized to 1, and the variable `max` (maximum value) is initialized to the first vector element. (Unlike many programming languages, Octave uses 1-based indexing). Next, a for loop considers each of the remaining elements in the vector, updating `idx` and `max` as appropriate.

#### Exercises:

1. Read 6 images, 3 of one letter and 3 of another, into Octave. Compute the average of these 6 images and display the average image. Also display the difference of each of the original images from the average image (using the `showdata` function defined above).
2. Write a function that reads a vector of image file names, and returns an image that is the average of the images in the files. The vector of image file names needs to be a cell array of strings (see section 6.2.3 of the manual). An example of how your function would be called:

```
avgImg = computeAvgImg( { "img1.pgm"; "img2.pgm"; "img3.pgm" } )
```

Test your function on different subsets of your image files. Verify that the results for the same files from Exercise 1 are correct.

## 5. Phase 2: Training and Feature Extraction

The method we will use for determining features is Principal Component Analysis. The general idea of Principal Component Analysis is to start with a feature set (in this case, each pixel in an image is treated as an independent feature of that image), and to determine a new set of features that better delineate the classes. Each new feature is a linear combination of the original features. Essentially, what we are doing is to treat each image as a point in a high-dimensional space, and performing a linear transformation of that space to create axes that best characterize the information. Therefore, the result of PCA will be as many features as we had before, but ranked in terms of significance (measured by variance in the data). We then select a relatively small number of most significant features (or axes in the data) for feature-based classification.

To understand the method better, you should read the following papers:

- Smith, L.I., *A Tutorial on Principal Component Analysis*. February, 2002. [http://www.cs.otago.ac.nz/cosc453/student\\_tutorials/principal\\_component\\_s.pdf](http://www.cs.otago.ac.nz/cosc453/student_tutorials/principal_component_s.pdf)
- Shiens, J., *A Tutorial on Principal Component Analysis*. December, 2005. <http://www.snl.salk.edu/~shlens/pub/notes/pca.pdf>
- Kirby, M., and L. Sirovich. "Application of the Karhunen-Loeve Procedure for the Characterization of Human Faces." *IEEE Trans. Patt. Anal. Mach. Intell.* 12.1 (1990): 103-108.

We will use GNU Octave's built-in matrix manipulation facilities extensively.

Our first goal is to create the covariance matrix, which will be very large square matrix. The number of rows and columns of this matrix will be the total number of pixels in an image. Again, our goal here is to treat each pixel of each image as a feature, so the covariances relate all possible pairs of features. To get the data into the correct form, we need to concatenate one vector per image, where each image's vector contains all of its pixels in order. (Which order doesn't really matter, as long as we're consistent). This can easily be done for a single image by using the built-in `vec` function: `imgVector = vec(img)`; You will need to concatenate all of these vectors together to create one big matrix with (number of pixels) rows, and (number of images) columns. Each row of the big matrix corresponds to one feature (corresponding pixel in all images), and each column of the big matrix corresponds to one data set (image). This big matrix is called  $X$  in the Shiens tutorial. Since each

element in the big matrix is of the form  $F_i - \bar{F}$  where F is a feature (pixel) and the subscript i denotes a particular image's feature, we can compute the complete covariance matrix by multiplying this X matrix by its transpose and dividing by the number of images - 1 (Shiens, equation 5).

Finally, we take the eigenvectors of the covariance matrix to define a new set of coordinate axes for the image space. Multiplying one of these vectors by an image yields a single value, which is a linear combination of all of the pixel values in the image. It can also be interpreted as the image's position along the coordinate axis denoted by the eigenvector. Hence, each eigenvector defines a feature of the image. The significance of the eigenvector is described by its eigenvalue – larger eigenvalues denote more significant features. Octave has a built-in function *eig*, that computes eigenvalues and eigenvectors. The results are a diagonal matrix of eigenvalues, and a matrix where each column is the eigenvector that corresponds to the eigenvalue in the same column. You will need to manipulate these matrices to create a matrix that contains the eigenvectors (one per column) sorted according to the values of the eigenvectors. Octave provides a *sort* function that will help you. The sort function provides two results – the first is a sorted vector, and the second is a list of indices showing how the elements were rearranged. Here's an example:

```
octave-3.0.3:19> [x, i] = sort ([1, 3, 2, 4, 6]);
octave-3.0.3:20> x
x =

    1    2    3    4    6

octave-3.0.3:21> i
i =

    1    3    2    4    5
```

The array of indices can be used to reorder another array so the corresponding elements are rearranged in the same way the elements of the original sorted array were rearranged:

```
octave-3.0.3:23> a = [10, 20, 30, 40, 50];
octave-3.0.3:25> a(i)
ans =

    10    30    20    40    50
```

You'll be doing almost the same thing, but you'll be sorting the matrix of eigenvectors based on the order of their eigenvalues. Use the sorting result to index the columns of the eigenvectors (and, since you don't want to rearrange rows, use : to index the rows).

Because our eigenvectors are in the same space as our image, we can visualize an eigenvector by rearranging the vector into a matrix with the original number of rows and columns as the image, and displaying it. (Note that the values will have to

be remapped to unsigned integers in the range 0-255 to use the `imshow` function). Pixels that are very bright or very dark are important in the feature computation, while pixels that are medium gray are less important in the computation. Images with high eigenvalues should look like reasonable “features” of the images, while images with low eigenvalues will look more like random noise.

### Exercises:

1. Write a function that creates the covariance matrix for a list of images. This function will need to compute the average of the images, create the data matrix that consists of one normalized image per row<sup>1</sup>, and calculate the covariance matrix by multiplying the data matrix times its transpose and dividing by the number of images - 1. Test your function first on some smaller matrices and then on a subset of your data. The form of this function might be:

```
[C] = createCovMat(imList);
```

The parameter `imList` can either be a list of images, or a list of file names that contain images, depending on what is more convenient for your implementation. This function could be a modification of your `computeAvgImg` function from Phase 1, or could call that function, depending on your implementation.

Test your function carefully. You will probably want to work with very small matrices at first so that you can verify your results by hand.

2. Write a function that computes the eigenvalues and eigenvectors of a covariance matrix. The list of eigenvectors should be sorted according to their eigenvalues.

```
[evecs, evals] = sortedEig( covMat )
```

Test your function on small square matrices before testing on a real covariance matrix. Make sure that the eigenvectors are all unit vectors and that they are sorted according to the eigenvalues, in descending order.

3. Use the functions you have written so far to compute the eigenvectors of your small data set (5 images from each of 2 letters). You can visualize the  $n$ 'th eigenvector with the following command, where `nrows` and `ncols` are the number of rows and columns of the original data.

```
showdata(reshape(evecs(:,n),nrows, ncols))
```

---

<sup>1</sup> Be sure to cast your matrix to a general numeric data type such as `double`; `uint8` matrices cannot be multiplied in Octave.

Look at the first 10 eigenvectors of your set. Do they look like they reflect important features of your data? How do the first 10 eigenvectors differ qualitatively from the last 10 eigenvectors?

4. Based on your results, decide how many eigenvectors from this set to carry forward as features for your recognition system. Since you are (at this point) only classifying two letters, you should probably not need more than one or two. In your decision, consider two things. First, at what point does there seem to be a large drop in the eigenvector value? Second, looking at the images of the eigenvectors, at what point do the images seem to concentrate on less significant features?

## 6. Phase 3: Letter Recognition

Once we have chosen a subset of eigenvectors as our features, we are ready to perform letter recognition based on the training data. Our first task is to compute the features for each image in the training data. We will compute each feature by multiplying its eigenvector by the vectorized image. (Again, this is simply computing a linear combination of pixel values). If we construct a matrix ( $E$ ) of selected eigenvectors, one per row (note this requires taking the transpose of the selected columns of the eigenvector matrix computed above), we can multiply this matrix by the  $X$  matrix from phase 2 (one column per image, where the average image has been subtracted out), and the result will be a matrix with the feature vectors in the columns. Each column corresponds to the feature vector for one image. This matrix represents the final result of training – a set of computed features for each training image.

You might find it useful to visualize these features. If you have chosen three or fewer features, you can simply use a plotting program to plot each image's feature vector as a point. Label each point with its class (the letter being signed). If your system has trained well, you should see a single cluster for each class. Not as good a result, but still one that is useful for training is if a class has multiple clusters, but the clusters are distinct. A poor result would be classes whose clusters overlap considerably. If you have such a case, you might want to add one or more additional features to your system, by choosing the eigenvectors with the next values below the ones that you have already chosen.

To recognize a new image, we must compute the features of the new image. This is done by first subtracting the average image (from training) from the new image, then multiplying our  $E$  matrix by the image vector. The result will be a column vector of the features of the new image.

```
imgFeatures = E*vec(double(img)-avgImg);
```

Now, we have a database of features from the training images, and a single feature from each test image. The simplest method for classifying the new image is to assign it the same class as the *most similar* of the test images. Since the test images are represented in a  $k$ -dimensional feature space (where  $k$  is the number of eigenvectors we chose), we can define *most similar* as *least distance* in the  $k$ -dimensional space spanned by our features. While there are a variety of distance measures, Euclidean distance is a good starting point. The formula for computing Euclidean distance between two points is computed by taking the square root of the sum of the squares of the differences in each dimension, i.e.

$$dist = \sqrt{\sum_i (A_i - B_i)^2}$$

So, all we have to do to recognize an object is to compute the distance to each of the training images and choose the closest one as the letter that we've recognized.

Choosing the closest training image is called a 1-nearest neighbor strategy. A more robust system would look at more neighbors and let each one "vote" on the classification. For example, a 3-nearest-neighbor strategy would consider the 3 closest training images and choose the class of the majority of these neighbors.

More sophisticated systems might use enhanced distance measures or some form of clustering to determine a single feature for each class (which would speed up recognition significantly, especially for a relatively large training set). Another enhancement is to allow a "none of the above" response if the result is too far from the entire training set.

## Exercises

1. Write a function that computes the Euclidean distance between two vectors.
2. Compute the database of vectors for your training image set.
3. Write a function to find the training image that is most similar to a new image.
4. Test your classification system. First, make sure that every training image is recognized as itself, with a distance of 0. Then try other examples of the same letters, and see how well your classification system does. Finally, test letters that are not in the database, and see whether the answers make any sense.
5. The classification system that we are using is a 1-nearest-neighbor system, i.e. the single nearest neighbor determines the classification. An extension to this system would be to consider  $k$  nearest neighbors (say, 3). Each neighbor votes for its class and the class with the most votes wins.

## 7. Phase 4 Testing and Analysis

Up to now, you have been training and testing with a small subset of your data. In the process of completing the exercises in the previous phases, you should have created an infrastructure that will allow you to do some full-scale testing, using more classes (letters) and more examples of each class in your training set. Be sure to continue to hold out some data for testing – do not put all images in the training set, or you will not be able to observe how your system will react to a new image.

Your final report should discuss the results of your system, showing a selection of images, the visualizations of the eigenvectors, the feature clusters, and of course, testing results. You should also evaluate your results – where does your system succeed, and where does it fail? What could be done if you had more time to improve your system?

### Software required

ImageMagick [www.imagemagick.org](http://www.imagemagick.org) (Image conversion)

Octave [www.gnu.org/software/octave](http://www.gnu.org/software/octave) (matrix manipulation)

Image package for Octave <http://octave.sourceforge.net/image/index.html> (image i/o and additional processing)

### Acknowledgement

This assignment was inspired by an Integrated Research Project for Computer Vision performed in 2004 by Art Geigel, III. All images copyright 2004 by Art Geigel, III.