Local Image Statistics

Lab Exercises for the Machine Vision Course

Rein van den Boomgaard
Informatics Institute
University of Amsterdam
The Netherlands

1 Introduction

The answers to the questions in the LabExercise should be presented in a report in pdf of doc format.

2 Isodata Thresholding

1. In the lecture notes an algorithm for isodata thresholding is given. The goal of image thresholding is to obtain a binary image where the pixels with value 1 correspond with the objects of interest. Implement the isodata thresholding and use it to threshold the following images:`coins.png`, `rice.png`, `bacteria.png`, `cameraman.tif`, `moon.tif`, `flowers.tif` and `kids.tif`. Be sure to convert color images to gray value images (`rgb2gray(..)`) and to convert images to doubles in the range from 0 to 1 (`im2double(..)`). For each of the images report the threshold value that is calculated and give your opinion on whether the isodata algorithm finds 'the right' threshold and whether thresholding could lead to a proper result.

2. In the algorithm presented in the lecture notes all pixels in the image are processed in each iteration of the isodata algorithm. Rewrite the algoritm to use the image histogram as input to calculate the isodata threshold value. An image histogram is calculated with the function `hist` (see the lecture notes). Compare the threshold values calculated (and the resulting images) of the histogram based algorithm with those of the pixel based algorithm.

3 k-Means clustering

Consider the image `beef.tif`. Three gray value peaks are easily distinguishable in the image histogram. Isodata thresholding therefore will not work in this case. In this exercise we will look at *k-means clustering* to find the clusters.

1. Implement the k-means clustering algorithm. You may do so by working on the image itself (working on the histogram as input is also possible here but not required). You may do so writing explicit loops over the image and perhaps you should try to convince yourself that you understand what is going on. Below you find the basics for a Matlab algorithm that uses the 'array processing' capabilities of Matlab. If you are going to use this code

then make sure you do understand what is done (look in the online manual for the use of a dimension specifier in the `min` function and the use of *logical indexing* in `a( ci==1 )`).

   (a) Assume 'a' is the image array

   (b) Make a vector of cluster means and assign initial values: `cm = [0 0.5 1];`

   (c) Calculate a 3D array `d` such that `d(i,j,k)` gives the distance from pixel value `a(i,j)` to cluster mean `cm(k)`: `d(:,:,1) = abs(a-cm(1));` (do this for all clusters).

   (d) For each pixel find the cluster (index) that has shortest distance: `[dummy,ci] = min( d, [], 3 );`

   (e) Given the new cluster indices (classification) the cluster means can be updated: `cm(1) = mean( a(ci==1) );` (do this for all clusters)

   (f) Goto 1c in case a significant number of pixels is changing cluster assignment.

2. The `ci` image gives the cluster number for each pixel. Displaying this image as a gray value image is not too bad in this particular case (why?). To stress the fact that they are indeed cluster numbers it is better to display them as colors: `imshow( label2rgb( ci, @jet, 'k' ) );`

3. What are the clustermeans that your implementation finds for the `beef.tif` image?

4. How could you improve the classification? Hint: look at the image histogram and compare the standard deviations of the different clusters. How would you use this information in the clustering.

5. **(not obligatory)** What are the clustermeans for the `alumgrns.tif` image. How many clusters would you start with.

6. **(not obligatory)** Can you prove that isodata thresholding is equivalent with 2-means clustering of a scalar set of values.

7. **(not obligatory)** Can you generalize the algorithm to work with color images?

4 Image Filters

The main principle of a local neighborhood image operator is: consider all pixels values in a neighborhood around a point (i, j) and from those values calculate a new value and set the resultant image in point (i, j) to that value.

In this exercise you have to implement several types of these *image filters*. For this we define a generic image filter:

```
function r = imageFilter( image, nbhsize, func )
[M N] = size( image );
nbhisize = round((nbhsize(1)-1)/2);
nbhjsize = round((nbhsize(2)-1)/2);
r = image;
for j=1:N
  jind = (j-nbhjsize):(j+nbhjsize);
  jind = max(1, min(N, jind));
  for i=1:M
    iind = (i-nbhisize):(i+nbhisize);
```

```
      iind = max(1, min(M, iind));
      nbhdata = image(iind, jind);
      r(i,j) = feval( func, nbhdata(:) );
    end
  end
```

This function is an alternative for the `nlfilter` function in Matlab. The disadvantage of `nlfilter` is that is uses zero padding of the border (i.e. it uses the values 0 for those points in a neighborhood that are outside the image; see the discussion on the 'borderproblem' in the lecture notes). A second difference with the `nlfilter` function is that the `imageFilter` function passes the values in the local neighborhood as a vector.

Using the `imageFilter` to implement a simple image filter is easy. Consider the call `b = imageFilter(a,[3 3],@mean);`. This calculates the mean within every 3×3-neighborhood.

1. Implement the Gaussian weighted mean (for this function it is easiest if you modify the imageFilter function to pass extra arguments to the function to be called for every neighborhood). The Gaussian weighted mean filter of an image f resulting in an image g is defined as:
$$g(i,j) = \sum_k \sum_l f(i+k, j+l) G^s(k,l)$$
where
$$G^s(k,l) = \frac{1}{2\pi s^2} \exp\left(-\frac{k^2+l^2}{2s^2}\right)$$

2. Implement the median filter.

3. Implement the percentile filter.

4. Implement the α-trimmed mean filter (i.e. calculate the mean of all values larger then the α-percentile and less then the $(100-\alpha)$-percentile; see the lecture notes: exercise 4 in chapter 3).

5. All the above image filters work as smoothing filters, i.e. they remove noise from the image. Compare the performance of these filters when applied to the cameraman image corrupted with Gaussian noise and salt & pepper noise:

   ```
   a = imread('cameraman.tif');
   a = imnoise( a, 'gaussian', 0, 0.001 );

   b = imread('cameraman.tif');
   b = imnoise( b, 'salt & pepper' );
   ```

   Compare all filters with respect to noise removal (quantify the amount of noise that is removed: the original is known) and edge preservation (you can do that subjectively with scores like '–', '-', '+', '++') when dealing with the different types of noise.

Hint: to make an array of Gaussian weights in a $(2N+1) \times (2N+1)$ neighborhood, the following Matlab code can be used:

```
[X,Y] = ndgrid( -N:N );
W = exp(-(X.^2+Y.^2)/(2*s^2));
W = W / sum(W(:));
```

5 The bilateral filter

This exercise is for extra points.

The bilateral filter is introduced by Manduchi and Tomasi in the last decade of the previous century. Despite the fact that it is a relatively new filter in the image processing toolbox, it is a simple and elegant extension of the well known Gaussian smoothing filter that works remarkably good as a noise suppresion filter while preserving the edges in images.

The classical Gaussian weighted mean filter uses a weight that is inversely proportional to the distance from a pixel to the center of the neighborhood:

$$g(i,j) = \sum_k \sum_l f(i+k, j+l) G^s(k,l)$$

where

$$G^s(k,l) = \frac{1}{2\pi s^2} \exp\left(-\frac{k^2+l^2}{2s^2}\right)$$

The bilateral filter uses a second weight that is inversely proportional to the grey value difference of the pixel value and the value of the central pixel:

$$g(i,j) = \frac{\sum_k \sum_l f(i+k, j+l) G^t(f(i+k, j+l) - f(i,j))\, G^s(k,l)}{\sum_k \sum_l G^t(f(i+k, j+l) - f(i,j))\, G^s(k,l)}$$

Note that whereas in the Gaussian weighted mean filter where the normalization to unit sum of all weights is implicit in the weights, in the bilateral filter the normalization is explicit because the weights are data dependent.

The 'scale' parameter t is called the *tonal scale*.

1. Implement the bilateral filter, i.e. write a function `bilaterFilter( image, spatialscale, tonalscale )`.

   Hint: it is easiest in case you modify the `imageFilter` function in such a way that the function that is called for each neighborhood takes an extra parameter being the value $f(i,j)$, i.e. the value of the central pixel.

2. Of course Matlab is an interpreted language. The `imageFilter` function is therefore bound to be rather slow when compared to a compiled C(C++) program. Nevertheless it is worthwhile to make an execution profile of the bilateral filter that you have implemented. You can do that easily in Matlab:

   ```
   profile on
   b = bilateralFilter( a, 3, 0.01 );
   profile off
   profile viewer
   ```

   What part of your code is taking most of the time? Can you suggest ways to speed things up?

The bilateral filter as it is introduced here may seem a bit ad-hoc. It is possible to derive the bilateral filter as the solution of robust estimation procedure. Remember that the Gaussian mean filter is the solution of a least squares estimator of a local constant image function. A

robust procedure is a generalization of the least squares estimator in which the quadratic error measure is replaced with a robust error measure that does not let outliers (large deviations from the model) influence the result that much. For more information on robust estimators and the bilateral filter we refer to the work of Joost van de Weijer (a PhD student at the UvA computer science department), see `http://www.science.uva.nl/~joostw/isis/publications.html` (a short introduction is the paper with title "On the equivalence of local-mode finding, robust estimation and mean-shift analysis as used in early vision tasks", a more detailed paper is titled "Leasts Squares and Robust Estimation of Local Image Structure".