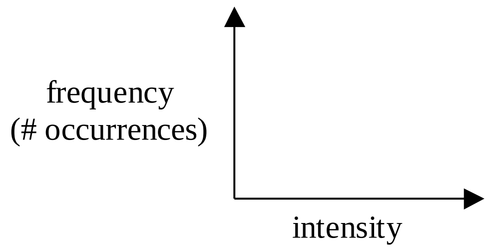
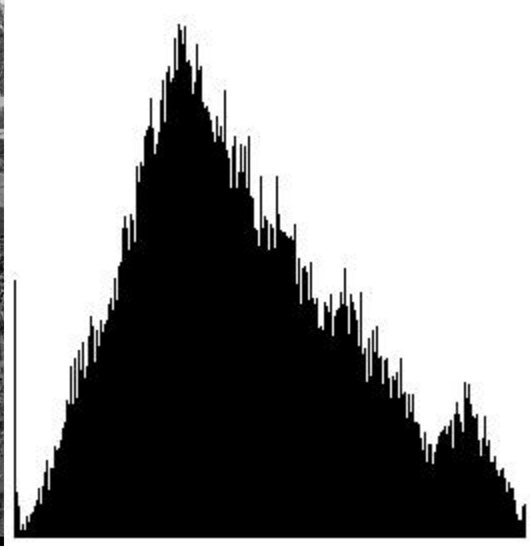
Lecture notes: Histogram, convolution, smoothing

Histogram. A plot of the intensity distribution in an image.



The following shows an example image and its histogram:

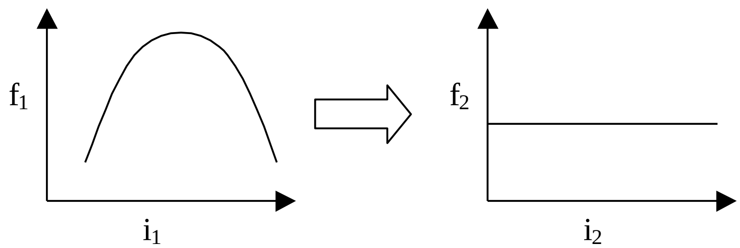


If we denote a greyscale image as I[r,c] then the histgram H[i] can be computed as

$$H[i] = \sum_{r,c} \begin{cases} 1 & I[r,c] = i \\ 0 & I[r,c] \neq i \end{cases}$$

The histogram is often used in image restoration or cleaning.

Histogram equalization. Stretch the contrast evenly through the intensity range by manipulating the histogram. The distribution of intensity is remapped to come as close as possible to uniform:



We desire to find a transform T for each original intensity i_1 to a new value i_2 so that the histogram becomes uniform.

$$i_2 = T(i_1)$$

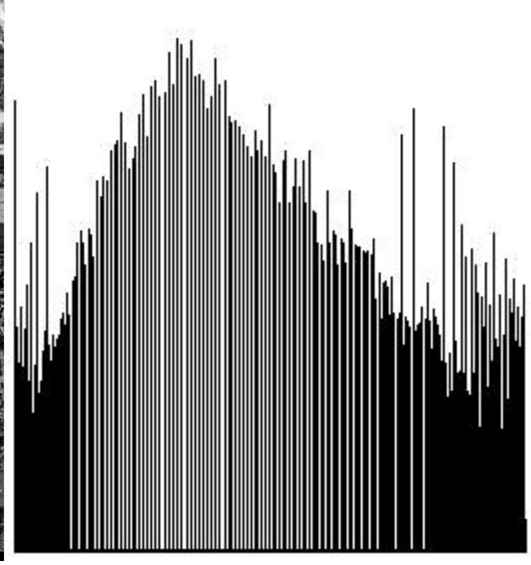
However, because the function H[i] is discrete the output will only be approximately uniform.

Assuming we have an image of ROWS by COLS 8-bit pixels, the histogram equalization transform can be written as

$$i_2 = T(i_1) = \sum_{x=0}^{i_1} H[x] * \frac{1}{ROWS * COLS} * 255$$

where the summation on H[] computes how much of the image has an intensity less than or equal to i_1 (this is the cumulative histogram), the fraction 1/(ROWS*COLS) normalizes these percentages (this is the normalized cumulative histogram), and the value 255 scales the output i_2 to the desired range 0...255.

The following shows the image from above after histogram equalization, along with the equalized histogram:



In C code, it can be computed as follows:

```
unsigned char     *image;
int               ROWS,COLS;
int               hist[256],x;
double            nhist[256],chist[256];

for (x=0; x<256; x++)
  hist[x]=0;
for (x=0; x<ROWS*COLS; x++)
  hist[image[x]]++;
for (x=0; x<256; x++)          /* normalized distribution */
  nhist[x]=(double)hist[x]/(double)(ROWS*COLS);
chist[0]=nhist[0];
for (x=1; x<256; x++)          /* cumulative distribution */
  chist[x]=chist[x-1]+nhist[x];
for (x=0; x<ROWS*COLS; x++)    /* remap pixels according to chist */
  image[x]=(unsigned char)(255.0 * chist[image[x]]);
```

What purpose does histogram equalization serve? It tends to sharpen the details visible in an image, by increasing their contrast. For a human viewer, this can be quite useful. For a machine vision system, it is generally useless, as no new information is gleaned through the process.

Convolution. Combining local-area information.

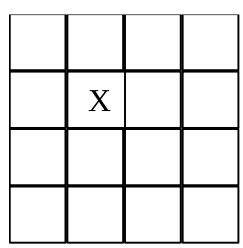
Image convolution can be written as

$$O[r,c]=\sum_{dr=-W}^{+W}\sum_{dc=-W}^{+W} I[r+dr,c+dc]*f[dr,dc]$$

where the range -W...+W is a **window** of local-area information. The function f[] is called a **filter**, and weights how much each pixel in the local area contributes to the output. I[] is the input image and O[] is the output image.

Smoothing. Suppressing noise in an image.

Consider a portion of an image



in which a pixel X is corrupted by noise. How could we go about suppressing this noise, and determining a good value for the pixel?

One way is to take the average of all the pixels in the local neighborhood. For example, we could convolve the image with W=1 and

$$f = \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

This is a **mean filter.** Mean filtering is good when nothing is known about the type of noise affecting the image.

Often we assume that the noise has a Gaussian distribution (for no better reason that because lots of naturally occurring things have a Gaussian distribution). In this case we can perform **Gaussian smoothing** using a Gaussian-shaped filter:

$$f[dr, dc] = \frac{1}{2\pi\sigma^2} e^{\frac{dr^2 + dc^2}{-2\sigma^2}}$$

where σ is the standard deviation of the Gaussian noise, and the stuff in front of e is a normalizing constant (may need to be adjusted).

Suppose the corrupted pixel X is a spike, caused by a temporary loss or saturation of signal? In that case, averaging would be bad, because the spike would clearly bias the mean. This type of noise is often called **salt-and-pepper noise.**

A **median filter** is good for spike noise. Each pixel X is replaced by the median (middle) value in its local neighborhood. A median filter cannot be implemented by convolution.

When working with a segmentation, another convenient smoothing filter is the **mode filter.** Each pixel X is replacted by the mode (most commonly occurring) value in its local neighborhood. A mode filter cannot be implemented by convolution.

The following shows the above image smoothed with a 3x3 mean, median, and mode filter. Note the very different results.

An example of salt-and-pepper noise will be demonstrated in class, along with the result from using these different methods to smooth it.

Separable filters. Convolution can be slow as W gets large. Separating a 2D filter into two 1D filters can greatly speed convolution.

$$O_1[r,c] = \sum_{dc=-W}^{+W} I[r,c+dc] * f_c[dc]$$

$$O[r,c] = \sum_{dr=-W}^{+W} O_1[r+dr,c] * f_r[dr]$$

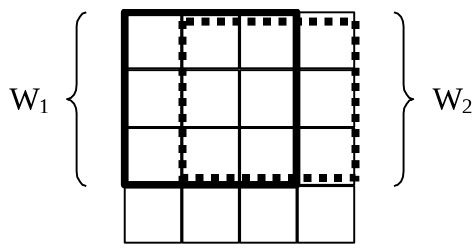
For example, the mean filter could be implemented using W=1 and separating f[] into the filters

$$f_c = \begin{bmatrix} 1/3 & 1/3 & 1/3 \end{bmatrix}$$

$$f_r = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

The choice of which filter f_c or f_r to convolve first is arbitrary. Note the need of an intermediary result image $O_1[]$.

Sliding window. In the case where W is large, convolution can also be sped up by using the summation from the preceeding pixel. For example:



How does the summation $f*W_1$ differ from $f*W_2$? Only by the subtraction and addition of a single column at each end. As W gets large, computing the summation this way can save a great deal of time.

The sliding window and separable filter tricks can be applied together, speeding the computation even more.