Local Binary Pattern: comparison between a sequential and parallel implementation

Beatrice Paoli
beatrice.paoli@stud.unifi.it

Abstract

The purpose of this paper is to implement the algorithm to compute the Local Binary Pattern for an image. There are many variations of this algorithm: the chosen implementation allows to choose any radius and number of pixels in the neighborhood but does not consider "uniform patterns". Both a sequential and a parallel implementation are presented and the two versions are compared in terms of execution times and speedup. The implementations are done in Python and the parallel version uses the Joblib package.

1. Introduction

Local Binary Pattern (LBP) is a simple and efficient texture descriptor that labels each pixel of an image with a binary number based on the neighboring pixels in order to create a local representation of the texture.

Used for texture analysis and classification, thanks to its computational simplicity it is widely used for many applications such as face analysis and motion analysis.

Many variants have been developed over time, but the basic idea of the LBP operator is to assign to each pixel of the greyscaled image a value by considering the difference of each of the 3x3 neighboring pixels with the center as binary values and then combine them in a clockwise or counter-clockwise to get the decimal label to assign to the pixel.

The histogram obtained from the occurrencies of these 2^8 labels can be used as a texture descriptor.

An extension of this basic concept allows to use circular neighborhoods of different sizes and different distances from the center. The operator is denoted as $LBP_{P,R}$ where P is the number of sampled neighbors and R is the radius of the circle where the sampled pixels stand.

If c_x and c_y are the coordinates of the center pixel, the coordinates of each neighbour pixel p can be computed as:

$$p_x = c_x - R\sin(2\pi p/P)$$
$$p_y = c_y + R\cos(2\pi p/P)$$

If these coordinates don't end into the center of a pixel, the grayscale value is bilinearly interpolated.

At this point, the LBP operator is computed as:

$$LBP_{P,R} = \sum_{p=0}^{P-1} s(g_p - g_c)2^p$$

where g_c and g_p are respectively the grayscale values of the center pixel and neighbouring pixel and:

$$s(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{otherwhise} \end{cases}$$

This version of the LBP operator is not rotation-invariant as it depends on the order of consideration of the $0...P-1$ neighbours. So rotating the image will lead to different results.

To solve this problem, another variant of the LBP operator considers *uniform patterns* to achieve rotation-invariance, but for the implementation presented in this paper it wasn't used.

After obtaining the labeled image $f_l(x, y)$ the LBP histogram can be defined as:

$$H_i = \sum_{x,y} I\{f_l(x,y) = i\}, i = 0...n-1$$

where n is the number of different labels assigned and:

$$I(A) = \begin{cases} 1, & \text{if } A \text{ is true} \\ 0, & \text{otherwhise} \end{cases}$$

If we need to compare histograms of images of different sizes, they must be normalized first:

$$N_i = \frac{H_i}{\sum_{j=0}^{n-1} H_j}$$

2. Implementation

The implementation is done in Python and uses the *Pillow* package to load and save images in RGB format, and the *numpy* package for handling arrays.

The application first loads the image in grayscale and converts it into a numpy array.

The pseudo-code of the LBP function is the following: