Local Binary Pattern

Comparison between a sequential
and parallel implementation

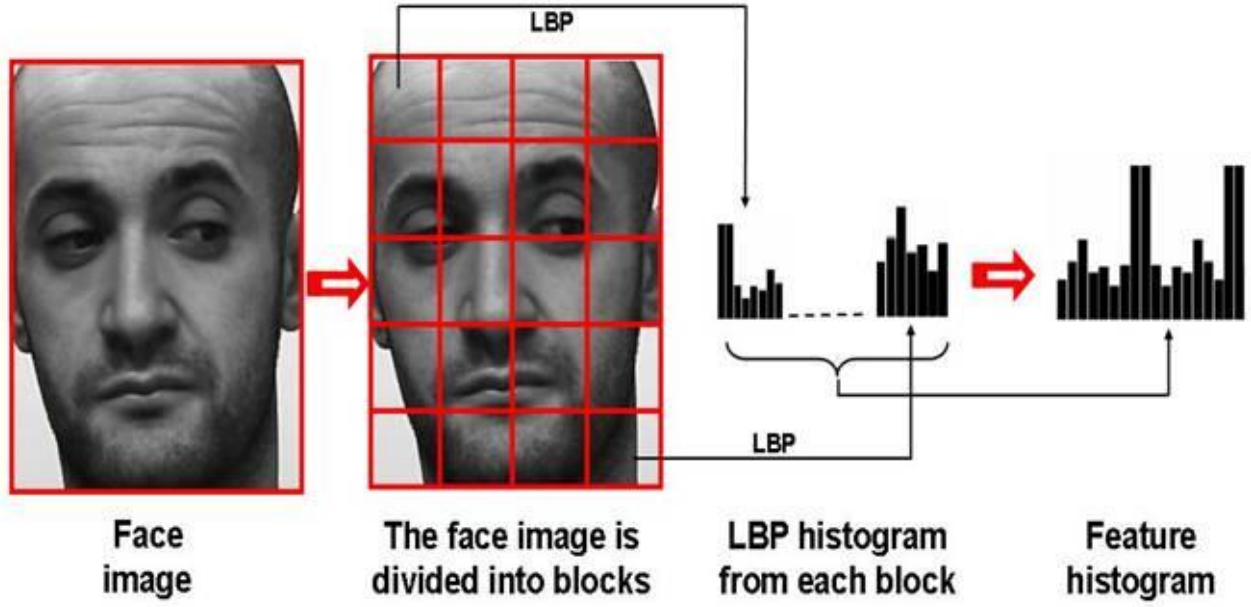
Beatrice Paoli

Firenze – 2/2/2023

Introduction

Local Binary Pattern

Local Binary Pattern (LBP) is a simple and efficient texture descriptor that labels each pixel of a grayscale image with a binary number based on the neighboring pixels in order to create a local representation of the texture.
The histogram obtained from the occurrences of these labels can be used as a texture descriptor.

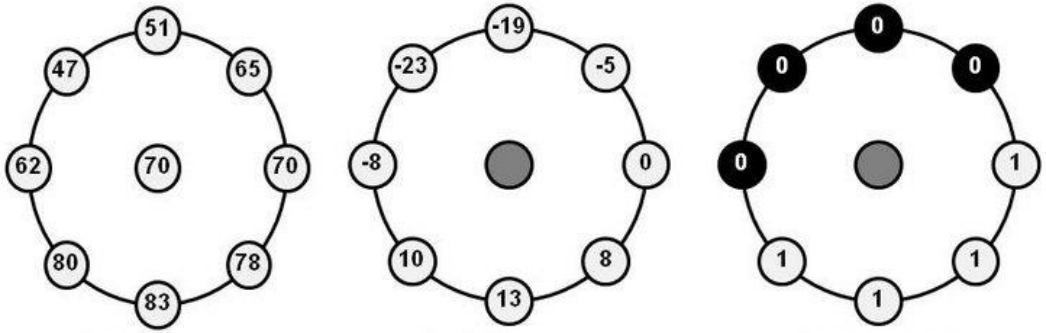


Introduction

Local Binary Pattern

The value of the LBP code of a pixel (x_c, y_c) is given by:

$$LBP_{P,R} = \sum_{p=0}^{P-1} s(g_p - g_c)2^p \qquad s(x) = \begin{cases} 1, if\ x \geq 0; \\ 0, otherwise. \end{cases}$$



1. Sample 2. Difference 3. Threshold

1*1 + 1*2 + 1*4 + 1*8 + 0*16 + 0*32 + 0*64 + 0*128 = 15

4. Multiply by powers of two and sum

Introduction

Local Binary Pattern – Uniform Patterns

A circular neighbourhood of pixels is said to have a **uniform pattern** if there are few spatial transitions.

More formally, we define the function *U* that calculates the number of spatial transitions (bitwise 0/1 changes) in the neighbourhood pattern. Only patterns with a value of *U* of at most 2 are considered uniform.

$$LBP^{riu2}_{P,R} = \begin{cases} \sum_{p=0}^{P-1} s(g_p - g_c), & \text{if } U(LBP_{P,R}) \leq 2 \\ P+1, & \text{otherwhise} \end{cases}$$

$$U(LBP_{P,R}) = |s(g_{p-1} - g_c) - s(g_0 - g_c)| + \sum_{p=1}^{P-1} |s(g_p - g_c) - s(g_{p-1} - g_c)|$$

Introduction

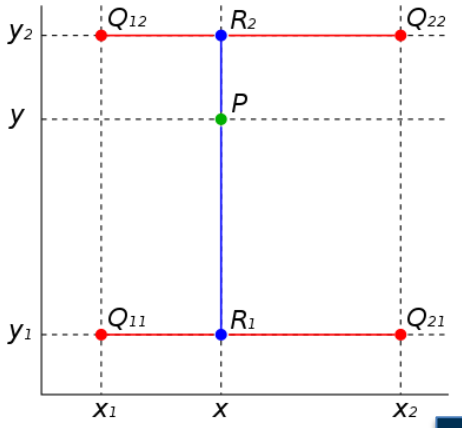
Bilinear Interpolation

If *cx* and *cy* 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.

$$f(x,y) = \frac{1}{(x_2-x_1)(y_2-y_1)}\begin{bmatrix} x_2-x & x-x_1 \end{bmatrix}\begin{bmatrix} f(Q_{11}) & f(Q_{12}) \\ f(Q_{21}) & f(Q_{22}) \end{bmatrix}\begin{bmatrix} y_2-y \\ y-y_1 \end{bmatrix}$$



Implementation

Sequential LBP

```python
def lbp(input_img: np.ndarray, points: int, radius: float) -> np.ndarray:
    height, width = input_img.shape
    output = np.zeros((height, width))
    int_radius = np.ceil(radius).astype(int)
    padded_input = np.pad(input_img, ((int_radius, int_radius), (int_radius, int_radius)), 'constant',
                          constant_values=(0, 0))
    for cy in range(height):
        for cx in range(width):
            lbp_val = 0
            p_vals = []
            for p in range(points):
                px = cx - radius * np.sin(2 * np.pi * p / points)
                py = cy + radius * np.cos(2 * np.pi * p / points)
                p_val = bilinear_interpolation(padded_input, px + int_radius, py + int_radius)
                p_vals.append(p_val)
                if p_val >= input_img[cy, cx]:
                    lbp_val += 1
            s = np.heaviside(np.subtract(p_vals, input_img[cy, cx]), 1)
            u = np.sum(np.abs(s[1:] - s[:-1]))
            if u > 2:
                lbp_val = points + 1
            output[cy, cx] = lbp_val
    output = (np.rint(output)).astype(np.uint8)
    return output
```

Implementation

Interpolation

```python
def bilinear_interpolation(input_img: np.ndarray, px, py) -> float:
    x1 = np.floor(px).astype(int)
    x2 = np.ceil(px).astype(int)
    y1 = np.floor(py).astype(int)
    y2 = np.ceil(py).astype(int)

    q11 = input_img[y1, x1]
    q12 = input_img[y2, x1]
    q22 = input_img[y2, x2]
    q21 = input_img[y1, x2]

    if x1 == x2 and y1 != y2:
        return q11 * ((y2 - py) / (y2 - y1)) + q12 * ((py - y1) / (y2 - y1))
    elif x1 != x2 and y1 == y2:
        return q11 * ((x2 - px) / (x2 - x1)) + q22 * ((px - x1) / (x2 - x1))
    elif x1 == x2 and y1 == y2:
        return input_img[y1, x1]
    else:
        return (1 / (x2 - x1) * (y2 - y1)) * (
                q11 * (x2 - px) * (y2 - py)
                + q21 * (px - x1) * (y2 - py)
                + q12 * (x2 - px) * (py - y1)
                + q22 * (px - x1) * (py - y1)
        )
```

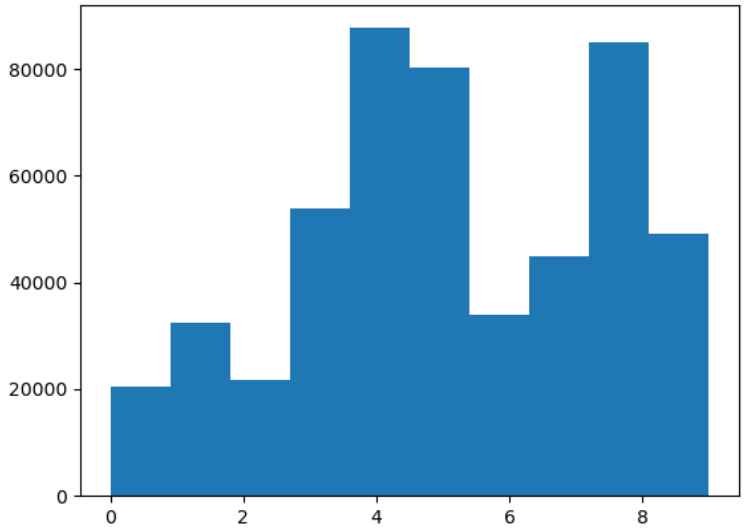
Implementation

Main

```
if __name__ == '__main__':
    img = Image.open(r"./input/test.jpg").convert("L")
    in_img = np.asarray(img)

    pts = 8
    rd = 1

    start_time = timeit.default_timer()
    output_img = lbp(in_img, pts, rd)
    end_time = timeit.default_timer()
    print("Time (s): ", end_time - start_time)

    plt.hist(output_img.flatten(), bins=pts + 2)
    plt.savefig('output/hist.png')

    out_img = Image.fromarray(output_img)
    out_img.save('./output/res.jpg', 'jpeg')
```

Result

Histogram



Parallelization with Joblib

Steps

Joblib is a high-level library that allows to parallelize computation through the use of a backend library (multiprocessing or loki) by spawning **processes** instead of **threads**.

The LBP algorithm is an **embarassingly parallel problem**: the computation of each output pixel is independent from one another, and the input image is only read, **no synchronization** is needed.

However, it isn't convenient to have each pixel computed independently because of the drastic increase in overhead for process management. Therefore, the image is split in sub-images, and each process computes the output for a single slice.

Another problem: each process needs the input image (a numpy array), a large object that needs to be copied every time and slows down the process instantiation. To avoid this, we can use the **joblib.dump()** and **joblib.load()** functions to save and reload the numpy array efficiently.

Parallelization with Joblib

Code

```
def lbp(input_img: np.ndarray, points: int, radius: float, proc: int) -> np.ndarray:
    height, width = input_img.shape

    savedir = mkdtemp()
    input_path = os.path.join(savedir, 'input.joblib')
    dump(input_img, input_path, compress=True)

    slice_height = np.ceil(height / proc).astype(int)
    output_slices = Parallel(n_jobs=proc)(delayed(sub_image_lbp)
                                          (input_path, points, radius, y, y + slice_height, 0, width)
                                          for y in range(0, height, slice_height))

    output = output_slices[0]
    for i in range(1, len(output_slices)):
        if output_slices[i] is not None:
            output = np.vstack((output, output_slices[i]))

    os.remove(input_path)
    output = (np.rint(output)).astype(np.uint8)
    return output
```

Parallelization with Joblib

Code

```
def sub_image_lbp(input_path: str, points: int, radius: float, start_y: int, end_y: int, start_x: int,
                  end_x: int) -> np.ndarray | None:
    input_img = load(input_path)
    height, width = input_img.shape

    start_x = start_x if start_x > -1 else 0
    start_y = start_y if start_y > -1 else 0
    end_x = end_x if end_x <= width else width
    end_y = end_y if end_y <= height else height

    if start_x >= width or start_y >= height or end_x <= 0 or end_y <= 0:
        return None

    int_radius = np.ceil(radius).astype(int)
    padded_input = np.pad(input_img, ((int_radius, int_radius), (int_radius, int_radius)), 'constant',
                          constant_values=(0, 0))

    slice_height = end_y - start_y
    slice_width = end_x - start_x
    output = np.zeros((slice_height, slice_width))
```

Parallelization with Joblib

Code

```
    for cy in range(slice_height):
        for cx in range(slice_width):
            lbp_val = 0
            p_vals = []
            for p in range(points):
                px = cx + start_x - radius * np.sin(2 * np.pi * p / points)
                py = cy + start_y + radius * np.cos(2 * np.pi * p / points)
                p_val = bilinear_interpolation(padded_input, px + int_radius, py + int_radius)
                p_vals.append(p_val)
                if p_val >= input_img[cy + start_y, cx + start_x]:
                    lbp_val += 1
            s = np.heaviside(np.subtract(p_vals, input_img[cy + start_y, cx + start_x]), 1)
            u = np.sum(np.abs(s[1:] - s[:-1]))
            if u > 2:
                lbp_val = points + 1
            output[cy, cx] = lbp_val
    output = (np.rint(output)).astype(np.uint8)
    return output
```

Speedup Analysis

Definition

In order to compare the performance of a sequential algorithm with its parallel version we can use the concept of **speedup**.

Speedup is measured as the ratio between the execution time of the sequential algorithm and the parallel one.

$$S = \frac{t_s}{t_p}$$

Ideally, we should look for *perfect speedup* or even *linear speedup*, meaning S should be equal or similar to the number of processors used to perform the parallel algorithm.

Speedup Analysis

Speedup results with different K and N

The experiments were performed on a Intel Core i7-1165G7 with 4 physical cores 8 logical cores.

To evaluate the application performance, different images and values for the parameter P were used.

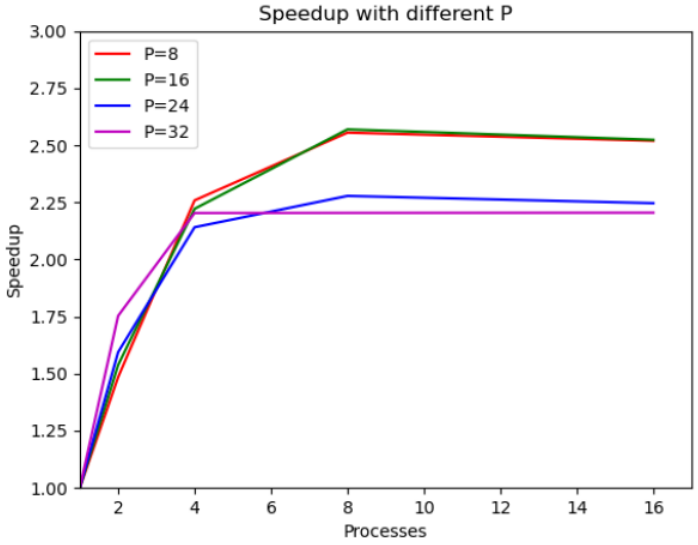


Figure 1. Speedup of the LBP operator on a 734x694 image and $R = 1$ with different values for P.

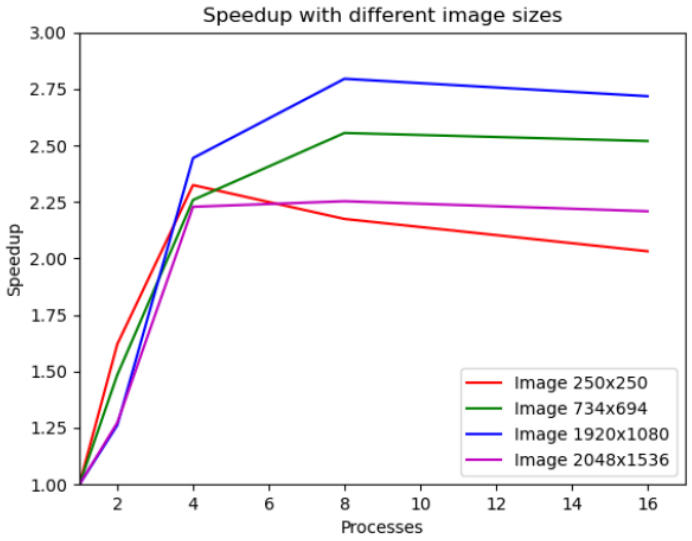


Figure 2. Speedup of the LBP operator with $P = 8$, $R = 1$ with different image sizes.

Speedup Analysis

Execution Times

Processes	Execution Time (s)	Speedup
1	48.630	1.000
2	32.759	1.484
4	21.531	2.259
8	19.032	2.555
16	19.302	2.519

Table 1. Execution Times for P=8.

Processes	Execution Time (s)	Speedup
1	97.452	1.000
2	63.378	1.538
4	43.866	2.222
8	37.927	2.569
16	38.614	2.524

Table 2. Execution Times for P=16.

Processes	Execution Time (s)	Speedup
1	130.883	1.000
2	82.116	1.594
4	61.132	2.141
8	57.453	2.278
16	58.265	2.246

Table 3. Execution Times for P=24.

Processes	Execution Time (s)	Speedup
1	169.365	1.000
2	96.626	1.753
4	76.865	2.203
8	76.845	2.204
16	76.816	2.205

Table 4. Execution Times for P=36.

Speedup Analysis

Execution Times

Processes	Execution Time (s)	Speedup
1	5.218	1.000
2	3.220	1.620
4	2.245	2.324
8	2.399	2.175
16	2.568	2.032

Table 5. Execution Times for the 250x250 image.

Processes	Execution Time (s)	Speedup
1	48.630	1.000
2	32.759	1.484
4	21.531	2.259
8	19.032	2.555
16	19.302	2.519

Table 6. Execution Times for the 734x694 image.

Processes	Execution Time (s)	Speedup
1	203.600	1.000
2	161.339	1.262
4	83.312	2.444
8	72.849	2.795
16	74.913	2.718

Table 7. Execution Times for the 1920x1080 image.

Processes	Execution Time (s)	Speedup
1	272.129	1.000
2	213.772	1.273
4	122.107	2.229
8	120.782	2.253
16	123.202	2.209

Table 8. Execution Times for the 2048x1536 image.

Parallelization with Joblib

Code

```
def lbp_multi(input_list: list[np.ndarray], points: int, radius: float, proc: int) -> list[np.ndarray]:
    savedir = mkdtemp()
    input_paths = []
    for i in range(len(input_list)):
        input_path = os.path.join(savedir, f'input{i}.joblib')
        dump(input_list[i], input_path, compress=True)
        input_paths.append(input_path)

    outputs = Parallel(n_jobs=proc)(delayed(lbp_single_load)(input_path, points, radius)
                                    for input_path in input_paths)

    for path in input_paths:
        os.remove(path)
    return outputs


def lbp_single_load(input_path: str, points: int, radius: float) -> np.ndarray:
    input_img = load(input_path)
    return lbp(input_img, points, radius)
```

Speedup Analysis

Speedup results with different datasets

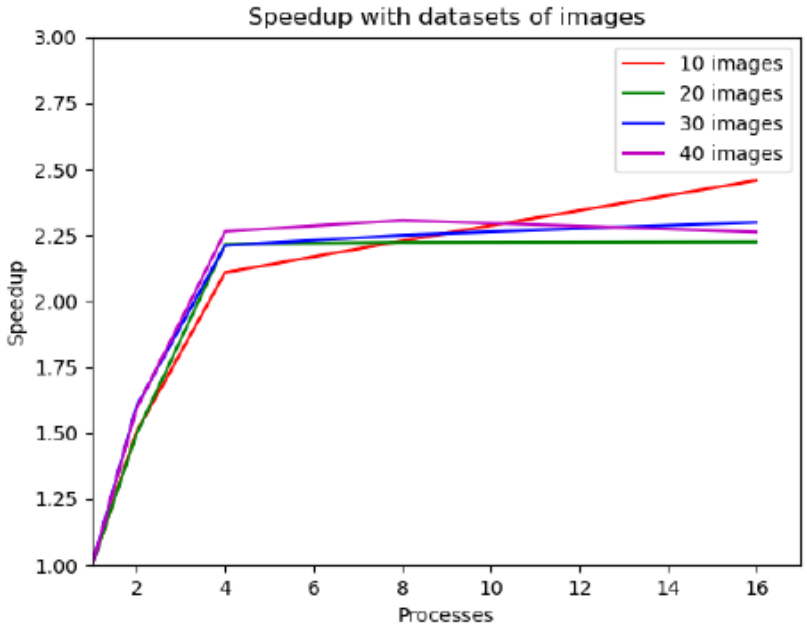


Figure 3. Speedup of the LBP operator with $P = 8$, $R = 1$ with different sizes of datasets of 450x450 images.

Speedup Analysis

Execution Times

Processes	Execution Time (s)	Speedup
1	189.700	1.000
2	126.128	1.504
4	89.952	2.109
8	85.126	2.228
16	77.210	2.457

Table 9. Execution Times for the dataset of size 10.

Processes	Execution Time (s)	Speedup
1	339.472	1.000
2	226.982	1.496
4	153.173	2.216
8	152.700	2.223
16	152.570	2.225

Table 10. Execution Times for the dataset of size 20.

Processes	Execution Time (s)	Speedup
1	522.823	1.000
2	326.594	1.601
4	236.412	2.211
8	232.357	2.250
16	227.539	2.298

Table 11. Execution Times for the dataset of size 30.

Processes	Execution Time (s)	Speedup
1	693.219	1.000
2	435.379	1.592
4	306.265	2.263
8	300.660	2.306
16	306.603	2.261

Table 12. Execution Times for the dataset of size 40.