HOG 人脸识别算法开发文档 V1

一、主要内容

Hog 特征是图像特征提取三大法宝（Hog 特征,LBP 特征,Haar 特征）之一，全称为方向梯度直方图（Histogram of Oriented Gradient, 简称 HOG），它是目前计算机视觉、模式识别领域很常用的一种描述图像局部纹理的特征。首先将图像分成小的连通区域，我们把它叫细胞单元。然后采集细胞单元中各像素点的梯度的或边缘的方向直方图。最后把这些直方图组合起来就可以构成特征向量。本文采用 Hog 算子提取人脸图像特征，得到特征向量之后，最后采用余弦夹角计算两张人脸图片相似度。经过测试，本算法取得较好识别效果，而且计算速度快，0.5s 左右。

二、HOG 特征提取算法步骤

1、将一个 image 灰度化（将图像看做一个 x,y,z（灰度）的三维图像）；

2、采用 Gamma 校正法对输入图像进行颜色空间的标准化（归一化）；目的是调节图像的对比度，降低图像局部的阴影和光照变化所造成的影响，同时可以抑制噪音的干扰；

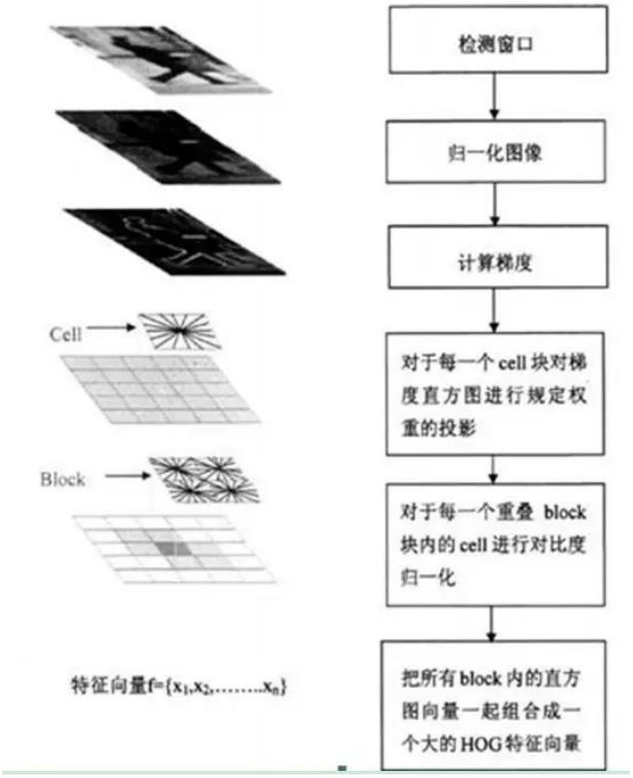
3、计算图像每个像素的梯度（包括大小和方向）；主要是为了捕获轮廓信息，同时进一步弱化光照的干扰。

4、将图像划分成小 cells（例如 8*8 像素/cell）；

5、统计每个 cell 的梯度直方图（不同梯度的个数），即可形成每个 cell 的 descriptor；

6、将每几个 cell 组成一个 block（例如 3*3 个 cell/block），一个 block 内所有 cell 的特征 descriptor 串联起来便得到该 block 的 HOG 特征 descriptor。

7、将图像 image 内的所有 block 的 HOG 特征 descriptor 串联起来就可以得到该 image（你要检测的目标）的 HOG 特征 descriptor 了。这个就是最终的可供分类使用的特征向量了。



三、建模过程

3.1、读入人脸图像，归一化大小

利用 OpenCV，imread 函数读入人脸图像，并且统一归一化大小（96*96），核心代码如下：

```
img1=cv2.imread("E:/656.jpg")
img1 = cv2.resize(img1, (96,96), interpolation=cv2.INTER_CUBIC)
img2=cv2.imread("E:/657.jpg")
img2 = cv2.resize(img2, (96,96), interpolation=cv2.INTER_CUBIC)
cv2.imshow("img1", img1)
cv2.imshow("img2", img2)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

查看效果：

3.2、人脸图像预处理

将读入的人脸图像进行灰度化，灰度化之后进行 gamma 校正。采用 Gamma 校正法对输入图像进行颜色空间的标准化，目的是调节图像的对比度，降低图像局部的阴影和光照变化所造成的影响，同时可以抑制噪音的干扰。

3.2.1、图像灰度化

```
img1_gray = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
img2_gray = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
cv2.imshow("img1", img1_gray)
cv2.imshow("img2", img2_gray)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

查看效果：

3.2.2、gamma 校正法

所谓伽玛校正就是对图像的伽玛曲线进行编辑，以对图像进行非线性色调编辑的方法，检出图像信号中的深色部分和浅色部分，并使两者比例增大，从而提高图像对比度效果。

Gamma 校正公式如下：

$$I(x,y) = I(x,y)^{\frac{1}{\text{gamma}}}$$

查看效果：

gamma 校正原理：

假设图像中有一个像素，值是 200 ，那么对这个像素进行校正必须执行如下步骤：

1、归一化 :将像素值转换为 0 ~1 之间的实数。算法如下 :(i + 0. 5)/256 这里包含 1 个除法和 1 个加法操作。对于像素 A 而言，其对应的归一化值为 0. 783203 。

2、 预补偿 ：根据公式, 求出像素归一化后的 数据以 1 /gamma 为指数的对应值。这一步包含一个求指数运算。若 gamma 值为 2. 2 , 则 1 /gamma 为 0. 454545 , 对归一化后的 A 值进行预补偿的结果就是 0. 783203 ^0. 454545 = 0. 894872 。

3、反归一化 :将经过预补偿的实数值反变换为 0～255 之间的整数值。具体算法为 :f*256 - 0. 5 此步骤包含一个乘法和一个减法运算。续前例 ，将 A 的预补偿结果 0. 894872 代入上式 , 得到 A 预补偿后对应的像素值为 228 , 这个 228 就是最后送入显示器的数据。

如上所述如果直接按公式编程的话，假设图像的分辨率为 800*600 ，对它进行 gamma 校正，需要执行 48 万个浮点数乘法、除法和指数运算。效率太低，根本达不到实时的效果。

针对上述情况，提出了一种快速算法，如果能够确知图像的像素取值范围 , 例如 ，0 ~ 255 之间的整数 , 则图像中任何一个像素值只能 是 0 到 255 这 256 个整数中的某一个 ; 在 gamma 值已知的情况

下 ,0～255 之间的任一整数 , 经过“归一 化、预补偿、反归一化”操作后 ， 所对应的结果是唯一的 ， 并且也落在 0 ～255 这个范围内。

如前例， 已知 gamma 值为 2. 2 , 像素 A 的原始值是 200， 就可求得经 gamma 校正后 A 对应的预补偿值为 228 。基于上述原理 , 我们只需为 0 ～255 之间的每个整数执行一次预补偿操作 ,将其对应的预补偿值存入一个预先建立的 gamma 校正查找表 , 就可以使用该表对任何像素值在 0 ～255 之间的图像进行 gamma 校正。

Gamma 校正实现代码如下：

```
############gamma 校正查找表
def BuildTable(gamma):
    table=[]
    for i in range(0,256):
        x1=(i+0.5)/256
        x2=1/gamma
        x3=np.power(x1,x2)
        x4=x3*256-0.5
        table.append(x4)
    return table
#############gamma 校正算法
def GammaCorrectiom(img1,gamma):
    mm=BuildTable(gamma)
    m, n = img1.shape
    for i in range(0, m):
        for j in range(0, n):
            img1[i][j] = mm[img1[i][j]]
    return img1
```

3.3、Hog 特征提取算法

3.3.1 计算图像每个像素的梯度（包括大小和方向）

计算图像横坐标和纵坐标方向的梯度，并据此计算每个像素位置的梯度方向值；求导操作不仅能够捕获轮廓，人影和一些纹理信息，还能进一步弱化光照的影响。

图像中像素点(x,y)的梯度为：

$$G_x(x,y) = I(x+1,y) - I(x-1,y)$$

$$G_y(x,y) = I(x,y+1) - I(x,y-1)$$

$$G(x,y) = \sqrt{G_x(x,y)^2 + G_y(x,y)^2}$$

$$\alpha(x,y) = \tan^{-1}(\frac{G_y(x,y)}{G_x(x,y)})$$

公式中 Gx(x,y),Gy(x,y)分别表示输入图像在像素点(x,y)处的水平方向梯度和垂直方向梯度。则 G(x,y),α(x,y)分别为像素点(x,y)的梯度幅值和梯度方向。

我们用 OpenCV 中的 sobel 算子计算图像每个像素的梯度，核心代码如下：

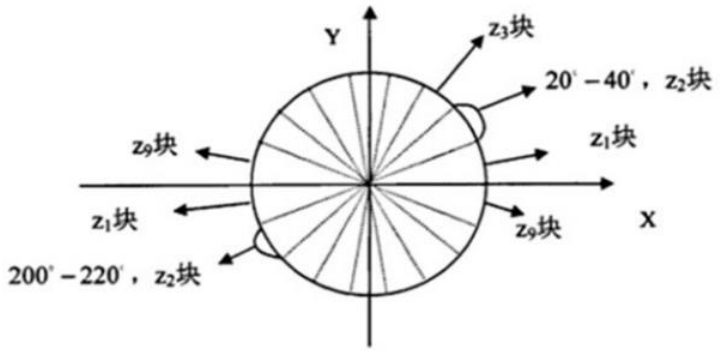
```
# calculate gradient magnitude and gradient angle for image using sobel

def calc_gradient(img):

    gradient_values_x = cv2.Sobel(img, cv2.CV_32F, 1, 0, ksize=5)

    cv2.convertScaleAbs(gradient_values_x)

    gradient_values_y = cv2.Sobel(img, cv2.CV_32F, 0, 1, ksize=5)

    cv2.convertScaleAbs(gradient_values_y)

    gradient_magnitude = cv2.addWeighted(gradient_values_x, 0.5, gradient_values_y, 0.5, 0)

    gradient_angle = cv2.phase(gradient_values_x, gradient_values_y, angleInDegrees=True)

    return gradient_magnitude, gradient_angle
```

3.3.2、为每个细胞单元构建梯度方向直方图

这一步骤的目的是为局部图像区域提供一个指示函数量化梯度方向的同时能够保持对图像中人体对象的姿势和外观的弱敏感性。

将图像分成若干个“单元格 cell”，例如每个 cell 为 8*8 的像素大小。假设采用 9 个 bin 的直方图来统计这 8*8 个像素的梯度信息，即将 cell 的梯度方向 0～180 度（或 0～360 度，考虑了正负，signed）分成 9 个方向块。如下图所示 :如果这个像素的梯度方向是 20-40 度 ,直方图第 2 个 bin 即的计数就加 1，

这样，对 cell 内每个像素用梯度方向在直方图中进行加权投影，将其映射到对应的角度范围块内，就可以得到这个 cell 的梯度方向直方图了，就是该 cell 对应的 9 维特征向量（因为有 9 个 bin）。这边的加权投影所用的权值为当前点的梯度幅值。例如说：某个像素的梯度方向是在，其梯度幅值是 4，那么直方图第 2 个 bin 的计数就不是加 1 了，而是加 4。这样就得到关于梯度方向的一个加权直方图。



核心代码如下：

```
#     calculate gradient for cells into a vector of 9 values
def calc_cell_gradient(cell_magnitude, cell_angle):
    orientation_centers = dict([(20, 0), (60, 0),  (100, 0),  (140, 0), (180, 0),  (220, 0), (260, 0), (300, 0), (340, 0)])
    x_left = 0
    y_left = 0
    cell_magnitude = abs(cell_magnitude)
    for i in range(x_left, x_left+8):
        for j in range(y_left, y_left+8):
            gradient_strength = cell_magnitude[i][j]
            gradient_angle = cell_angle[i][j]
            min_angle, max_angle = get_closest_bins(gradient_angle, orientation_centers)
            #print gradient_angle, min_angle, max_angle
            if min_angle == max_angle:
                orientation_centers[min_angle] += gradient_strength
            else:
                orientation_centers[min_angle] += gradient_strength * (abs(gradient_angle - max_angle)/40)
                orientation_centers[max_angle] += gradient_strength * (abs(gradient_angle - min_angle)/40)
    cell_gradient = []
    for key in orientation_centers:
        cell_gradient.append(orientation_centers[key])
    return cell_gradient
```

```
def get_closest_bins(gradient_angle, orientation_centers):
    angles = []
    #print math.degrees(gradient_angle)
    for angle in orientation_centers:
        if abs(gradient_angle - angle) < 40:
            angles.append(angle)
    angles.sort()
    if len(angles) == 1:
        #print gradient_angle, angles[0]
        return angles[0], angles[0]

    return angles[0], angles[1]
```

3.3.3、把细胞单元组合成大的块（block），块内归一化梯度直方图

由于局部光照的变化以及前景-背景对比度的变化，使得梯度强度的变化范围非常大。这就需要对梯度强度做归一化。归一化能够进一步地对光照、阴影和边缘进行压缩。

把各个细胞单元组合成大的、空间上连通的区间（blocks）。这样，一个 block 内所有 cell 的特征向量串联起来便得到该 block 的 HOG 特征。这些区间是互有重叠的，这就意味着：每一个单元格的特征会以不同的结果多次出现在最后的特征向量中。我们将归一化之后的块描述符（向量）就称之为 HOG 描述符。

核心代码：

```
#     render gradient
def render_gradient(image, cell_gradient):

    height, width = image.shape
    height = height - (height%8) - 1
    width = width - (width%8) - 1
    x_start = 4
    y_start = 4
    #x_start = height - 8
    #y_start = width - 8
    cell_width = 4
    for x in range(x_start, height, 8):
        for y in range(y_start, width, 8):
            cell_x =int(x/8)
            cell_y = int(y/8)
            cell_grad = cell_gradient[cell_x][cell_y]
            mag = lambda vector: math.sqrt(sum(i**2 for i in vector))
            normalize = lambda vector, magnitude : [element/magnitude for element in vector] if magnitude > 0 else  vector
            cell_grad = normalize(cell_grad, mag(cell_grad))
            angle = 0
```

```
                angle_gap = 40
                # print x, y, cell_grad
                for magnitude in cell_grad:
                    angle_radian = math.radians(angle)
                    x1 = int(x + magnitude*cell_width*math.cos(angle_radian))
                    y1 = int(y + magnitude*cell_width*math.sin(angle_radian))
                    x2 = int(x - magnitude*cell_width*math.cos(angle_radian))
                    y2 = int(y - magnitude*cell_width*math.sin(angle_radian))
                    test_a = (x1, y1)
                    test_b = (x2, y2)
                    # print   test_a
                    # print test_b
                    cv2.line(image, (y1, x1), (y2, x2), 255)
                    angle += angle_gap
        h, w = image.shape
```

3.3.4、收集 HOG 特征

最后一步就是将检测窗口中所有重叠的块进行 HOG 特征的收集，并将它们结合成最终的特征向量供分类使用。

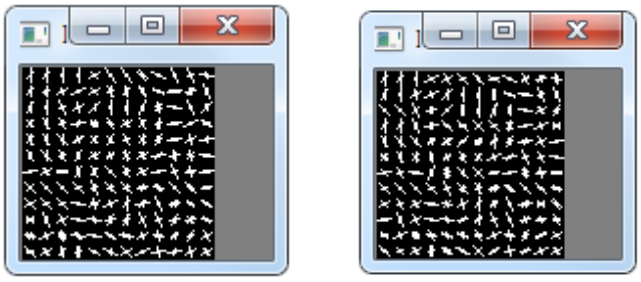
核心代码如下：

```
def hog_detector(img):
    height, width = img.shape
    gradient_magnitude, gradient_angle = calc_gradient(img)
    #showImage(gradient_magnitude)
    hog_vector = []
    num_horizontal_blocks = int(width/8)
    num_vertical_blocks = int(height/8)
    cell_gradients = np.zeros([int(height/8), int(width/8)])
    cell_gradient_vector = []
    #   calculating cell gradient vector
    for i in range(0, (height - (height%8)-1), 8):
        horizontal_vector = []
        for j in range(0, width - (width%8) -1, 8):
            cell_magnitude = gradient_magnitude[i:i+8, j:j+8]
            cell_angle = gradient_angle[i:i+8, j:j+8]
            horizontal_vector.append(calc_cell_gradient(cell_magnitude, cell_angle))
        cell_gradient_vector.append(horizontal_vector)

    # print "rendering height, width", height, width
    render_gradient(np.zeros([height, width]), cell_gradient_vector)
    height = len(cell_gradient_vector)
    width = len(cell_gradient_vector[0])
```

```
#   calculating final gradient hog vector
for i in range(height-1):
    for j in range(width - 1):
        vector = []
        vector.extend(cell_gradient_vector[i][j])
        vector.extend(cell_gradient_vector[i][j+1])
        vector.extend(cell_gradient_vector[i+1][j])
        vector.extend(cell_gradient_vector[i+1][j+1])
        mag = lambda vector: math.sqrt(sum(i**2 for i in vector))
        magnitude = mag(vector)
        if magnitude != 0:
            normalize = lambda vector, magnitude : [element/magnitude for element in
vector]
            vector = normalize(vector, magnitude)
        hog_vector.append(vector)


return hog_vector
```

3.3.5、Hog 特征可视化

利用 OpenCV 实现的 hog 特征可视化效果如下：



3.4、余弦夹角相似度计算

余弦相似度，又称为余弦相似性。通过计算两个向量的夹角余弦值来评估他们的相似度。余弦相似度用向量空间中两个向量夹角的余弦值作为衡量两个个体间差异的大小。余弦值越接近 1，就表明夹角越接近 0 度，也就是两个向量越相似，这就叫"余弦相似性"。

对于二维空间，根据向量点积公式 ，显然可以得知：

$$\cos\theta = \frac{A \cdot B}{\|A\|\|B\|}$$

假设向量 a、b 的坐标分别为(x1,y1)、(x2,y2) 。则：

$$\cos\theta = \frac{x_1x_{2+}y_1y_2}{\sqrt{{x_1}^2+{y_1}^2}\times\sqrt{{x_2}^2+{y_2}^2}}$$

设向量 A = (A1,A2,...,An)，B = (B1,B2,...,Bn) 。推广到多维:

$$\cos\theta = \frac{\sum_1^n(A_i\times B_i)}{\sqrt{\sum_1^n {A_i}^2}\times\sqrt{\sum_1^n {B_i}^2}}$$

核心代码如下：

```
def cos_dist(a, b):
    if len(a) != len(b):
        return None
    part_up = 0.0
    a_sq = 0.0
    b_sq = 0.0
    for a1, b1 in zip(a,b):
        part_up += a1*b1
        a_sq += a1**2
        b_sq += b1**2
    part_down = math.sqrt(a_sq*b_sq)
    if part_down == 0.0:
        return None
    else:
        return   abs((part_up / part_down))
```

效果如下：


Hog算法
"D:\Program Files\Python27\python.exe" F:/LBP人脸识别/Hog算法.py
(96L, 96L, 3L)
(96L, 96L, 3L)
121
121
the Image similary is 0.863076861937


Hog算法
"D:\Program Files\Python27\python.exe" F:/LBP人脸识别/Hog算法.py
(96L, 96L, 3L)
(96L, 96L, 3L)
121
121
the Image similary is 0.914840125291


Hog算法
"D:\Program Files\Python27\python.exe" F:/LBP人脸识别/Hog算法.py
(96L, 96L, 3L)
(96L, 96L, 3L)
121
121
the Image similary is 0.487383201843

Hog算法

```
"D:\Program Files\Python27\python.exe" F:/LBP人脸识别/Hog算法.py
(96L, 96L, 3L)
(96L, 96L, 3L)
121
121
the Image similary is 0.966501357369
```

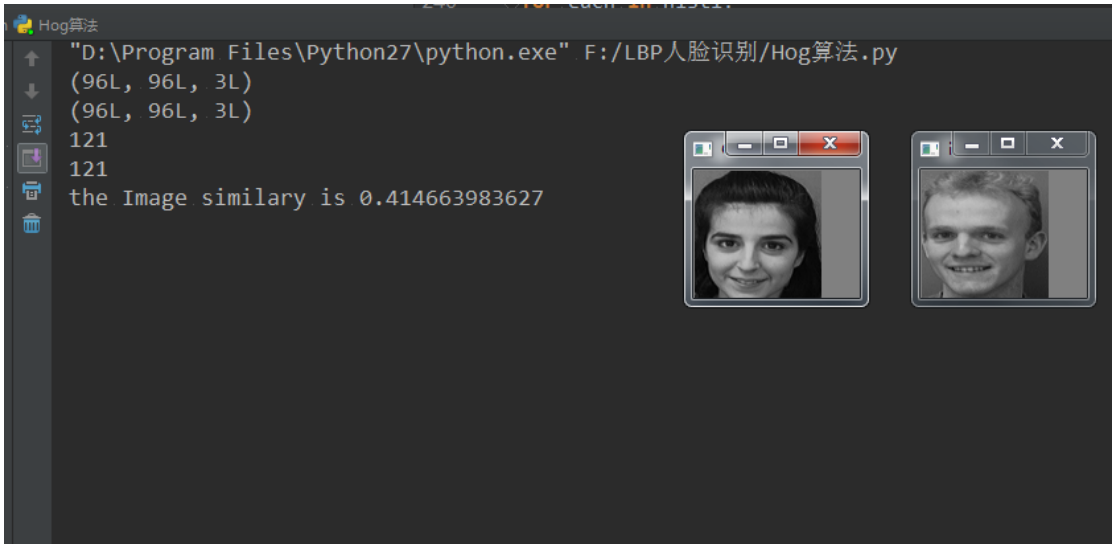
```
240     for each in hist1:
```

Hog算法

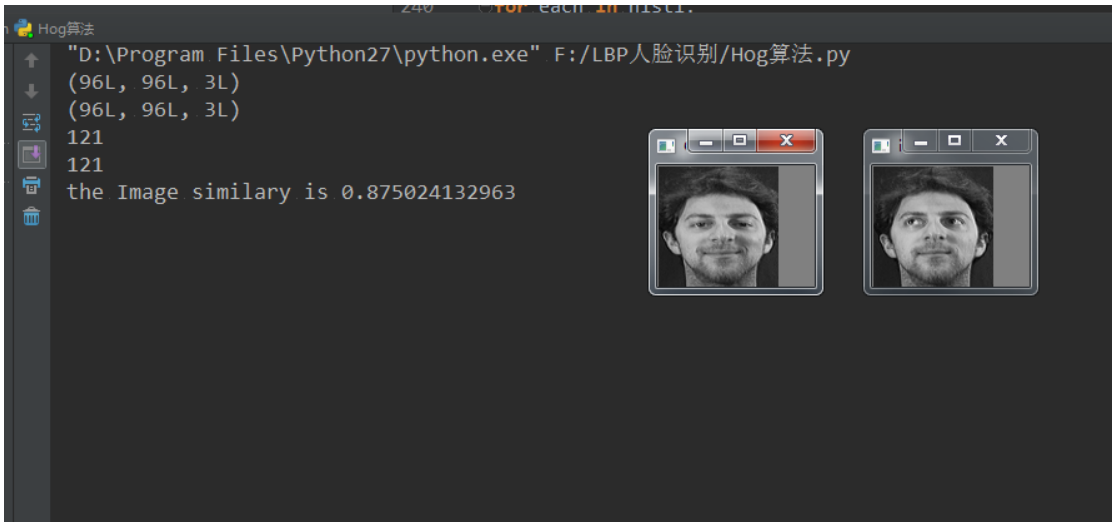
```
"D:\Program Files\Python27\python.exe" F:/LBP人脸识别/Hog算法.py
(96L, 96L, 3L)
(96L, 96L, 3L)
121
121
the Image similary is 0.80727688609
```

Hog算法

```
"D:\Program Files\Python27\python.exe" F:/LBP人脸识别/Hog算法.py
(96L, 96L, 3L)
(96L, 96L, 3L)
121
121
the Image similary is 0.448232311887
```


Hog算法
"D:\Program Files\Python27\python.exe" F:/LBP人脸识别/Hog算法.py
(96L, 96L, 3L)
(96L, 96L, 3L)
121
121
the Image similary is 0.414663983627


Hog算法
"D:\Program Files\Python27\python.exe" F:/LBP人脸识别/Hog算法.py
(96L, 96L, 3L)
(96L, 96L, 3L)
121
121
the Image similary is 0.437142753693


Hog算法
"D:\Program Files\Python27\python.exe" F:/LBP人脸识别/Hog算法.py
(96L, 96L, 3L)
(96L, 96L, 3L)
121
121
the Image similary is 0.875024132963