



UNIVERSIDAD DE GRANADA

Visión por Computador

Proyecto final

*Laura Calle Caraballo
Javier León Palomares*

12 de enero de 2018

Índice

1. Introducción	2
2. Descripción de la técnica e implementación	2
2.1. Cálculo de gradientes	2
2.2. Obtención de histogramas	3
2.3. Normalización de histogramas	4
3. Detección de peatones	5

Índice de figuras

1. Introducción

El objetivo de este proyecto es explorar la técnica de **Histograma de Gradientes (HoG)** aplicada a la detección de peatones, descrita en *Histograms of Oriented Gradients for Human Detection* (2005) de *Dalal y Triggs*.

Para ello, hemos implementado distintos aspectos del método realizando una serie de decisiones de diseño y hemos utilizado la base de datos original para comparar resultados. Las herramientas empleadas son *Python* y las librerías *OpenCV*, *NumPy* y *scikit-learn*.

2. Descripción de la técnica e implementación

El **Histograma de Gradientes** nos permitirá extraer características relevantes de una imagen para poder proceder a su clasificación (contiene un peatón o no). Consta de varias etapas que analizaremos a continuación.

2.1. Cálculo de gradientes

El primer paso del proceso es calcular los gradientes horizontales y verticales de cada imagen. Para ello, utilizaremos un *kernel* unidimensional $[-1, 0, 1]$ centrado en cada píxel a evaluar.

Utilizando los valores anteriores obtendremos las orientaciones de los cambios de intensidad y sus magnitudes. Posteriormente, ya que las imágenes son a color, elegiremos el canal con mayor magnitud de gradiente para cada píxel.

El código correspondiente a esta parte es:

```
def computeGradient(img, kx, ky):

    img_x = cv2.sepFilter2D(img, cv2.CV_32F, kx, ky,
                             borderType=cv2.BORDER_REPLICATE)
    img_y = cv2.sepFilter2D(img, cv2.CV_32F, ky, kx,
                             borderType=cv2.BORDER_REPLICATE)

    mag, angle = cv2.cartToPolar(img_x, img_y, angleInDegrees=True)
    angle = angle%180
    del img_x
    del img_y

    magnitudes = np.apply_along_axis(np.max, 2, mag)

    coord_0 = np.repeat(np.arange(img.shape[0]),img.shape[1])
    coord_1 = np.tile(np.arange(img.shape[1]),img.shape[0])

    indices = np.apply_along_axis(np.argmax, 2, mag)
    angles = angle[coord_0, coord_1, indices.flatten()]
    angles = angles.reshape(img.shape[:2])

    return magnitudes, angles
```

Para obtener las orientaciones de los gradientes (limitándolas al rango $[0, 180]$) y sus magnitudes hemos usado `cartToPolar`, que implementa las siguientes fórmulas:

$$\theta = \arctan \frac{g_y}{g_x}$$
$$g = \sqrt{g_x^2 + g_y^2}$$

Respecto a la selección del canal de color con mayor respuesta para cada píxel, la aproximación más directa sería utilizando dos bucles anidados para recorrer la imagen píxel por píxel, pero es posible ganar ligeramente en eficiencia aprovechando la capacidad de vectorización de NumPy: generamos todas las combinaciones de posiciones de la imagen, siendo cada una representada por `(coord_0[i], coord_1[i])`; obtenemos los índices del canal con mayor respuesta en cada píxel; finalmente, cada posición estará representada junto a su canal elegido por `(coord_0[i], coord_1[i], indices[i])`. De esta forma, podemos aprovechar la indexación por conjuntos de índices para resumir el proceso de selección en una única línea.

2.2. Obtención de histogramas

El uso de histogramas locales se basa en que la forma distintiva de un objeto se puede caracterizar muchas veces con suficiente calidad utilizando distribuciones y orientaciones de gradientes por áreas de una imagen. Además, al resumir implícitamente la información, la dimensionalidad del vector de características se reduce.

Para calcular estos histogramas, vamos a dividir la imagen en celdas cuadradas de un cierto número de píxeles (en nuestro caso, 8×8). En cuanto a los histogramas, estarán formados por 9 secciones que corresponden a intervalos de 20 grados entre 0 y 180, puesto que tomaremos las orientaciones por dirección y no por sentido. Para conocer la aportación del ángulo en un píxel, haremos una interpolación lineal que reparta su influencia entre las dos secciones más cercanas; por ejemplo, si tenemos un ángulo de 35 grados, aportará el 25% de su valor de magnitud a la sección centrada en 20 y el 75% restante a la centrada en 40.

```
def cellHistogram(cell_m, cell_o, bins = 9, max_angle = 180):
    bin_size = max_angle//bins
    histogram = np.zeros(bins)

    lower_bin = (cell_o//bin_size).astype(np.uint8)
    upper_bin = (lower_bin+1)%bins

    value_to_upper = (cell_o%bin_size) / bin_size
    value_to_lower = (1-value_to_upper) * cell_m
    value_to_upper *= cell_m

    for i in range(8):
        for j in range(8):
            histogram[lower_bin[i,j]] += value_to_lower[i,j]
            histogram[upper_bin[i,j]] += value_to_upper[i,j]

    return histogram
```

Como se puede apreciar, hacemos uso de vectorización para calcular en pocas líneas las aportaciones de todos los píxeles de la celda a las secciones del histograma.

Cabe mencionar que tanto el tamaño de las celdas como la disposición de los histogramas en 9 categorías cubriendo el rango $[0, 180]$ se han elegido así porque producen resultados de mayor calidad según el artículo. Asimismo, en el caso de las celdas un tamaño de 8×8 permite que haya un número exacto de éstas en las ventanas de 64×128 píxeles que luego usaremos para detectar peatones.

Finalmente, veamos cómo se usa la función a la hora de calcular todos los histogramas de una imagen:

```
def computeCellHistograms(border_size, magnitudes, angles, cell_size = 8,
                           bins = 9, max_angle = 180):

    end_r = magnitudes.shape[0] - border_size
    end_c = magnitudes.shape[1] - border_size

    rows = magnitudes.shape[0] - (border_size*2)
    cols = magnitudes.shape[1] - (border_size*2)

    histograms = np.zeros((rows//cell_size, cols//cell_size, bins), np.float)

    for i in range(border_size, end_r, cell_size):
        for j in range(border_size, end_c, cell_size):
            histograms[(i-border_size)//cell_size, (j-border_size)//cell_size] = \
                cellHistogram(magnitudes[i:i+cell_size, j:j+cell_size],
                              angles[i:i+cell_size, j:j+cell_size], bins, max_angle)

    return histograms
```

Lo primero que hacemos es eliminar el borde que pueda tener la imagen hasta que sus dimensiones sean múltiplos del tamaño de celda. Esto se hace principalmente porque los ejemplos positivos de peatones en los conjuntos de entrenamiento y test tienen un tamaño ligeramente mayor al de 64×128 que vamos a necesitar.

Posteriormente, recorreremos la imagen en saltos de 8 píxeles de izquierda a derecha y de arriba abajo para ir obteniendo los histogramas de toda la malla de celdas.

2.3. Normalización de histogramas

La alta variabilidad de contextos y grados de iluminación que existe en la realidad provoca que los gradientes que calculamos se muevan en rangos muy amplios. Puesto que esto puede añadir una complejidad significativa a la ya difícil tarea de distinguir entre una clase y todo lo demás, es necesario el uso de normalización para tener una escala unificada de magnitudes entre 0 y 1.

En nuestro caso, esta normalización se realiza agrupando las celdas en bloques cuadrados (por defecto, de 2×2 celdas) y aplicando el proceso con una cierta superposición entre dichos bloques (por defecto, 50%). La superposición, aunque pueda introducir redundancia, aporta calidad al proceso y mejora los resultados, guiándonos por las conclusiones de los autores de la técnica.

Según el cálculo realizado para normalizar, podemos distinguir entre las tres variantes probadas:

- División entre la norma L1:

```
def norm_1(x):  
    norm = np.linalg.norm(x,0)  
    return x/norm if norm != 0 else x
```

- División entre la norma L2:

```
def norm_2(x):  
    norm = np.linalg.norm(x)  
    return x/norm if norm != 0 else x
```

- División preliminar entre la norma L2, recorte de valores superiores a 0.2 y división entre su nueva norma L2:

```
def norm_2_hys(x):  
    x = norm_2(x)  
    x[x > 0.2] = 0.2  
    return norm_2(x)
```

La forma de aplicar lo anterior se traduce en la siguiente función:

```
def normalizeHistograms(histograms, norm_f = norm_2, block_size = 2,  
                        overlapping = 0.5):  
    step = int(block_size*(1-overlapping))  
    normalized = []  
    for i in range(0,histograms.shape[0]-block_size+1, step):  
        for j in range(0,histograms.shape[1]-block_size+1, step):  
            normalized.extend((norm_f(histograms[i:i+block_size, j:j+block_size].flatten()))  
                                )  
    return np.asarray(normalized)
```

La función recibe como parámetro el conjunto de histogramas, así como la normalización que aplicará, el tamaño de los bloques y el solapamiento entre los mismos. Tras calcular el intervalo entre cada par de bloques, los recorremos y añadimos cada conjunto resultante de histogramas normalizados al vector de características de la imagen.

3. Detección de peatones

Para la detección se ha utilizado un tamaño de ventana de 64×128 píxeles, cuya forma permite abarcar personas aproximadamente erguidas. Asimismo, dentro de las ventanas con ejemplos positivos de entrenamiento y test, parte de este tamaño es un borde alrededor del peatón que añade contexto útil; los autores comprobaron que reducir este borde causaba una pérdida notable de precisión.

Con los ejemplos positivos ya proporcionados, nuestra próxima tarea será conseguir ejemplos negativos del mismo tamaño a partir de las fotografías del conjunto de datos que no contienen personas. Siguiendo el procedimiento original, vamos a extraer 10 ventanas aleatorias de cada una de esas imágenes. Para no tener que generar miles cada vez que queramos entrenar un clasificador, las guardamos una única vez:

```
def extractNegativeWindows(path, dst_path):
    for name in os.listdir(path):
        img = cv2.imread(os.path.join(path, name))
        for i in range(10):
            f = random.randint(0, img.shape[0]-128)
            c = random.randint(0, img.shape[1]-64)
            cv2.imwrite(dst_path+"/"+str(i)+"_"+name, img[f:f+128, c:c+64])
```

Una vez tenemos todas las imágenes necesarias, es el momento de aplicar el proceso descrito en el apartado anterior a cada una de ellas para obtener las características con las que trabajarán los clasificadores de peatones. En código esto se traduce en recorrer los directorios de ejemplos positivos y negativos y utilizar las distintas funciones que calculan un **Histograma de Gradientes**:

```
def obtainDataFeatures(deriv_kernel, pos_path, neg_path, norm_f = norm_2):
    kx = deriv_kernel[0]
    ky = deriv_kernel[1]

    pos_directory = os.listdir(pos_path)
    neg_directory = os.listdir(neg_path)

    descriptor_length = (15*7)*(4*9)
    features = np.zeros((len(pos_directory) + len(neg_directory), descriptor_length))
    labels = np.repeat(np.asarray([1,0]), [len(pos_directory), len(neg_directory)])
    i = 0

    for name in pos_directory:
        img = cv2.imread(os.path.join(pos_path, name))

        magnitudes, angles = computeGradient(img, kx, ky)

        border_size = (img.shape[0]-128) // 2
        histograms = computeCellHistograms(border_size, magnitudes, angles, 8)
        descriptor = normalizeHistograms(histograms, norm_f=norm_f)
        features[i,:] = descriptor
        h.printProgressBar(i, features.shape[0])
        i+=1

    for name in neg_directory:
        img = cv2.imread(os.path.join(neg_path, name))
```

```
magnitudes, angles = computeGradient(img, kx, ky)

border_size = (img.shape[0]-128) // 2
histograms = computeCellHistograms(border_size, magnitudes, angles, 8)
descriptor = normalizeHistograms(histograms, norm_f=norm_f)
features[i,:] = descriptor
i+=1
h.printProgressBar(i, features.shape[0])

return features, labels
```

Tal como se mencionó anteriormente, las imágenes positivas ya incluidas en el conjunto de datos tienen un borde que incrementa un poco el tamaño, por lo que antes de procesar cada una hay que conocer cuánto vale ese borde para que `computeCellHistograms` trabaje sobre 64×128 .

Llamaremos a esta función dos veces: una para crear los datos de entrenamiento y otra para crear los datos de test.