



ugr | Universidad
de **Granada**

Grado en Ingeniería Informática. Cuarto.

Histograma de Gradientes aplicado a la detección de peatones.

Nombre de la asignatura:

Visión por Computador. Viernes de 9:30 a 11:30.

Realizado por:

David Lopez Pretel. 75573052C. derwey@correo.ugr.es
Néstor Rodríguez Vico. 75938615K. nrv23@correo.ugr.es



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS
INFORMÁTICA Y DE TELECOMUNICACIÓN.

Granada, 16 de enero de 2018.

Índice

1	Introducción.	3
2	HOG.	3
3	Pruebas.	8
3.1	Sin alisamiento Gaussiano.	10
3.2	Cambio en los tamaños de bloque y celdas.	11
4	Reconocimiento.	12
5	Bibliografía.	15

1. Introducción.

Nuestro trabajo consiste en la implementación del algoritmos del descriptor HOG (Histogram of Oriented Gradients) explicado en el paper *Histograms of Oriented Gradients for Human Detection* propuesto por *Navneet Dalal and Bill Triggs*. Hemos seguido los pasos que nos indica el propio paper y los vamos a ir explicando paso a paso. Para probar nuestro algoritmo hemos usado la base de datos que han usado los autores. Dicha base de datos la podemos encontrar en este enlace.

Veamos paso a paso el algoritmo.

2. HOG.

Lo primero que debemos hacer es realizar una corrección de Gamma. Este proceso nos permite codificar y decodificar la luminancia de la imagen. Tal y como podemos ver en Wikipedia, este proceso es simplemente aplicar la siguiente fórmula:

$$\text{salida} = A * \text{entrada}^\gamma$$

donde A es un valor constante (normalmente 1) y γ es un valor entre 0 y 1 (nosotros hemos usado 0.2 ya que hemos visto que aporta buenos resultados y a nosotros nos ha funcionado en nuestros experimentos). Para realizar este proceso, sólo debemos hacer lo siguiente:

```
1 img = np.power(img, 0.2, dtype=np.float32)
```

Como bien dice el propio nombre del algoritmo, éste trata con los gradientes. Vamos a calcular el gradiente para x y el gradiente para y. Para ello hacemos uso de la función *Sobel* de *OpenCV*:

```
1 gx = cv2.Sobel(img, cv2.CV_32F, 1, 0, ksize=1)
2 gy = cv2.Sobel(img, cv2.CV_32F, 0, 1, ksize=1)
```

Una vez tenemos los gradientes, debemos calcular la orientación y la magnitud de los mismos. Para ello vamos a usar la función *cartToPolar* de *OpenCV*:

```
1 magnitudes, angles = cv2.cartToPolar(gx, gy, angleInDegrees=True)
```

Con esta función obtenemos dos matrices de 3 dimensiones. Dichas matrices tendrán tantas filas y tantas columnas como la imagen original, ya que se calcula la magnitud y el ángulo del gradiente para cada píxel. Como tenemos una imagen en color, tendríamos, para cada píxel, 3 valores de magnitud y ángulo, uno para cada canal. Pero lo que vamos a hacer es, tal y como nos dice el paper, quedarnos en cada píxel con el mejor canal, es decir, el que tenga un mayor valor de magnitud:

```
1 max_norm = np.argmax(magnitudes, axis=2)
```

Con la llamada anterior, tenemos una matriz que nos permite saber en qué canal se encuentra el máximo para cada pixel. Ahora sólo debemos convertir nuestras matrices de magnitudes y ángulos de 3 dimensionen en matrices de 2 dimensiones, quedándonos el canal con mayor magnitud. Esto lo hacemos con las siguientes líneas de código:

```

1 m, n = np.shape(max_norm)
2 I, J = np.ogrid[:m, :n]
3 max_angles = angles[I, J, max_norm]
4 max_magnitudes = magnitudes[I, J, max_norm]

```

La idea de quedarnos el canal con mayor magnitud es porque en ese color el gradiente es mayor, y por lo tanto nos aporta más información de lo que está sucediendo. El siguiente paso es convertir los ángulos para que estén en el rango 0-180 en vez de en el rango 0-360. El paper nos dice que un ángulo de 270° es igual que un ángulo de 90° y que, empíricamente, han visto que se obtienen resultados similares. Para ello, simplemente debemos aplicar el módulo a nuestros ángulos:

```

1 max_angles = max_angles % 180

```

Una vez tenemos todos listo, vamos a pasar a calcular los histogramas propiamente dichos. Para ello, vamos a dividir la imagen en celdas y vamos a calcular un histograma para cada celda. Las celdas pueden ser de tamaño variable, pero en el paper muestran una imagen (figura 5 del paper) donde se ve que el valor óptimo es de 6x6. El proceso es simple:

```

1 def make_cells(angles, magnitudes, cell_size):
2     cells = []
3     for i in range(0, np.shape(angles)[0], cell_size):
4         row = []
5         for j in range(0, np.shape(angles)[1], cell_size):
6             row.append(np.array(histogram(angles[i:i + cell_size, j:j + cell_size],
7                                         magnitudes[i:i + cell_size, j:j + cell_size]), dtype=np.float32))
7     cells.append(row)
8
9 return np.array(cells, dtype=np.float32)

```

Debemos tener cuidado con una cosa, las celdas no pueden sobreponerse, es decir, un píxel no puede influir en dos celdas distintas, por eso en los *range* usados para recorrer las matrices vamos incrementando en función del tamaño de la celda. Para cada celda, calculamos su histograma con la siguiente función:

```

1 def histogram(angles, magnitudes):
2     # [0, 20, 40, 60, 80, 100, 120, 140, 160]
3     h = np.zeros(10, dtype=np.float32)
4
5     for i in range(angles.shape[0]):
6         for j in range(angles.shape[1]):
7             angles[i, j] = 160
8             index_1 = int(angles[i, j] // 20)
9             index_2 = int(angles[i, j] // 20 + 1)
10
11            proportion = (index_2 * 20 - angles[i, j]) / 20
12
13            value_1 = proportion * magnitudes[i, j]
14            value_2 = (1 - proportion) * magnitudes[i, j]
15
16            h[index_1] += value_1
17            h[index_2] += value_2
18
19    h[0] += h[-1]
20    return h[0:9]

```

En el cálculo del histograma pretendemos repartir la magnitud de cada pixel en función de su ángulo. El histograma lo vamos a representar como un array de 9 casillas de la siguiente forma:

[0|20|40|60|80|100|120|140|160]

donde cada componente lo podemos ver como una cubeta donde vamos a ir incrementando su valor (todas se inicializan a cero) con la magnitud de cada pixel. Supongamos que tenemos un ángulo de 80 para un píxel, pues el valor de la magnitud de dicho píxel se incrementaría en la componente 5 (la asociada al ángulo 80) de nuestro histograma. Si nos llega un ángulo de 10, debemos repartir el valor de la magnitud en la mitad para la cubeta del 0 y la otra mitad para la cubeta del 20. Si nos llega un valor de 15, debemos asignar tres cuartos de la magnitud para la cubeta del 20 y un cuarto para la cubeta del 0. Si tenemos un valor que pasa de 160, echaremos la parte correspondiente en la cubeta del 160 y lo restante en la cubeta del 0. Por facilidad, nosotros tenemos 10 cubetas y al final de todo el proceso incrementamos la primera cubeta con el valor de la última (que sería la correspondiente al caso comentado anteriormente). Finalmente, devolvemos el histograma sin esa última cubeta ya que es una cubeta auxiliar para facilitar el calculo.

Una vez tenemos los histogramas de cada celda, debemos agrupar dichas celdas en bloques. Los bloques no es más que una agrupación cuadrada de celdas. Una vez más, estas agrupaciones pueden tener tamaño variable pero en la figura 5 del paper podemos ver que el tamaño óptimo es de 3x3. En este caso, al contrario que pasaba con los histogramas y las celdas, una celda si puede (y debe) contribuir en más de un bloque y, por lo tanto, la ventana que vamos moviendo para construir los bloques debe ir moviéndose de uno en uno. La función que calcula dichos bloques es la siguiente:

```

1 def make_blocks(block_size, cells):
2     before = int(block_size / 2)
3     after = int(block_size / 2)
4
5     if block_size % 2 != 0:
6         after = after + 1
7
8     first_stop = before
9     second_stop = before
10
11    if np.shape(cells)[0] % block_size == 0:
12        first_stop = first_stop - 1
13
14    if np.shape(cells)[1] % block_size == 0:
15        second_stop = second_stop - 1
16
17    blocks = []
18    for i in range(int(block_size / 2.0), np.shape(cells)[0] - first_stop):
19        for j in range(int(block_size / 2.0), np.shape(cells)[1] - second_stop):
20            blocks.append(np.array(cells[i - before:i + after, j - before:j + after].
21                               flatten()))
22
23    return blocks

```

Parece compleja por los condicionales, pero sólo están puestos para permitir que sea una función genérica y pueda aceptar cualquier tamaño de bloque. Pero si nos centramos en lo estrictamente relevante, son los dos for de las líneas 18 y 19, los cuales se van moviendo por las celdas (de uno en uno) y construyendo los bloques que necesitamos. Si tenemos un tamaño de bloque de 3x3 celdas, un bloque no es más que la concatenación

de los histogramas correspondientes a esas 9 celdas, es decir, un bloque es un vector de 9 histogramas concatenados.

Una vez tenemos los bloques listos, debemos aplicar un pequeño filtro de alisamiento Gaussiano. Consideramos que el histograma que está en el centro es el más importante y que los 8 restantes deben ser suavizados con el mismo valor, ya que están a la misma distancia del histograma central. Para ello vamos a aplicar el ejercicio del bonus de la práctica 1. Calculamos un núcleo Gaussiano simétrico (líneas 1 a 3) y multiplicamos nuestro bloque por los valores correspondientes (líneas 5 y 6).

```

1 kernel = [-1, 0, 1]
2 sigma = 0.5 * block_size
3 kernel = np.array([math.exp(-0.5 * (x ** 2 / sigma ** 2)) for x in kernel])
4
5 multiplier = np.append(np.append(np.ones(36) * kernel[0], np.ones(9) * kernel[1]), np.
6     ones(36) * kernel[2]))
6 blocks = [b * multiplier for b in blocks]
```

Finalmente, sólo debemos normalizar cada bloque. En el paper proponen 4 normalizaciones distintas. La primera de ellas es la norma L1 (línea 1), la siguiente es la norma L1 pero añadiendo la raíz cuadrada al resultado (línea 9), la tercera es la norma L2 (línea 17) y la última es la norma L2 pero aplicada dos veces (línea 25). Lo que hacen es aplicar la norma L2 y truncar todos los valores a 0.2 como máximo y luego aplicar de nuevo la norma L2. Las implementaciones son triviales pero las podemos ver a continuación. Debemos tener en cuenta los casos en los que la norma del vector sea cero (por tener un vector de todo ceros). También tenemos una función *normalize* que llama a cada normalización en función del parámetro recibido. Todas las normalizaciones tienen un parámetro de umbral. En nuestro caso, para todas las ejecuciones, dicho parámetro, por defecto, vale 0.

```

1 def normalize_L1(block, threshold):
2     norm = np.sum(block) + threshold
3     if norm != 0:
4         return block / norm
5     else:
6         return block
7
8
9 def normalize_L1_sqrt(block, threshold):
10    norm = np.sum(block) + threshold
11    if norm != 0:
12        return np.sqrt(block / norm)
13    else:
14        return block
15
16
17 def normalize_L2(block, threshold):
18    norm = np.sqrt(np.sum(block * block) + threshold * threshold)
19    if norm != 0:
20        return block / norm
21    else:
22        return block
23
24
25 def normalize_L2_Hys(block, threshold):
26    norm = np.sqrt(np.sum(block * block) + threshold * threshold)
27
28    if norm != 0:
```

```

29         block_aux = block / norm
30         block_aux[block_aux > 0.2] = 0.2
31         norm = np.sqrt(np.sum(block_aux * block_aux) + threshold * threshold)
32         if norm != 0:
33             return block_aux / norm
34         else:
35             return block_aux
36     else:
37         return block
38
39 def normalize(block, type_norm, threshold=0):
40     if type_norm == 0:
41         return normalize_L2(block, threshold)
42     elif type_norm == 1:
43         return normalize_L2_Hys(block, threshold)
44     elif type_norm == 2:
45         return normalize_L1(block, threshold)
46     elif type_norm == 3:
47         return normalize_L1_sqrt(block, threshold)

```

Con todo esto, el algopritmo obtenido es el siguiente:

```

1 def hog(img, cell_size=6, block_size=3, type_norm=0, all_norms=False):
2     # Gamma correction : gamma = 0.2
3     img = np.power(img, 0.2, dtype=np.float32)
4
5     # Calculate gradient
6     gx = cv2.Sobel(img, cv2.CV_32F, 1, 0, ksize=1)
7     gy = cv2.Sobel(img, cv2.CV_32F, 0, 1, ksize=1)
8
9     # Gradient magnitude and direction (in degrees)
10    magnitudes, angles = cv2.cartToPolar(gx, gy, angleInDegrees=True)
11    max_norm = np.argmax(magnitudes, axis=2)
12
13    # For each pixel, we store the angle with the biggest magnitude
14    m, n = np.shape(max_norm)
15    I, J = np.ogrid[:m, :n]
16    max_angles = angles[I, J, max_norm]
17    max_magnitudes = magnitudes[I, J, max_norm]
18
19    # Convert angles to be in range 0–180
20    max_angles = max_angles % 180
21
22    # Obtain the histogram for each region
23    cells = make_cells(max_angles, max_magnitudes, cell_size)
24
25    # Append the cells into blocks: 180 descriptors with 81 elements
26    blocks = make_blocks(block_size, cells)
27
28    # We need to apply a Gaussian kernel
29    kernel = [-1, 0, 1]
30    sigma = 0.5 * block_size
31    kernel = np.array([math.exp(-0.5 * (x ** 2 / sigma ** 2)) for x in kernel])
32
33    multiplier = np.append(np.append(np.ones(36) * kernel[0], np.ones(9) * kernel[1]),
34                           np.ones(36) * kernel[2]))
35    blocks = [b * multiplier for b in blocks]
36
37    # Now we need to normalize
38    if all_norms:
39        blocks_norm0 = np.concatenate([normalize(b, 0) for b in blocks])
40        blocks_norm1 = np.concatenate([normalize(b, 1) for b in blocks])
41        blocks_norm2 = np.concatenate([normalize(b, 2) for b in blocks])
42        blocks_norm3 = np.concatenate([normalize(b, 3) for b in blocks])
43
44        return blocks_norm0, blocks_norm1, blocks_norm2, blocks_norm3
45    else:

```

```

45     blocks = np.concatenate([normalize(b, type_norm) for b in blocks])
46     # Concatenate all the blocks and this is the final descriptor
47     return blocks

```

3. Pruebas.

Una vez tenemos nuestro algoritmo listo, vamos a probarlo. El algoritmo implementado, al fin y al cabo, no es más que un algoritmo que calcula un descriptor. Lo que vamos a hacer es calcular un descriptor para muchas imágenes y aprender un clasificador con dichos descriptores¹. En concreto, vamos a usar una máquina de soporte de vectores con un kernel lineal, ya que es lo recomendado por el paper. La SVM que vamos a usar es la de la biblioteca *sci-kit learn*. Como ya se comentó en la introducción, la base de datos de prueba es la proporcionada por el paper. El algoritmo está pensado para probar con ventanas de 64 píxeles de ancho y 128 píxeles de alto (64x128). En la página podemos ver que hay varias carpetas de imágenes y nos explican como usar cada una de ellas. En el caso de train y de las imágenes positivas, las imágenes son de tamaño 96x160 y nos dicen que usemos una ventana centrada en la imagen del tamaño deseado. En el caso de test y de las imágenes positivas, las imágenes son de tamaño 70x134, y la estrategia a seguir es la misma, coger una ventana centrada. En el caso de las imágenes negativas, la idea es coger de cada imagen 10 muestras aleatorias de tamaño 64x128. Una vez tenemos todos los descriptores HOGs calculados, los guardamos en un fichero *pickle* para poder usarlos en un futuro.

Con todos estos descriptores, la idea es aprender una SVM preliminar. Una vez tenemos dicha SVM, lo que nos propone el paper es coger más muestras de todas las imágenes negativas y ver donde nuestra SVM dice sí, es decir, está dando un falso positivo. Estos ejemplos también son denominados *ejemplos difíciles*. Lo ideal sería recoger todos los falsos positivos posibles y reentrenar la SVM con estos nuevos datos junto con los que ya teníamos. La idea de esto es aprender mejor la clase no peatón, ya que es una clase muy amplia y compleja y, probablemente, no se haya aprendido correctamente con los ejemplos negativos empleados. El problema que tiene esto es que consume muchísimo tiempo y muchísima memoria, ya que si tenemos 1218 imágenes negativas tendríamos del orden de $n_filas * n_columnas/2$ descriptores por cada imagen, que multiplicado por las 1218 imágenes que tenemos y todo esto multiplicado por el número de niveles que queramos hacer en la pirámide Gaussiana tendríamos muchísimas ventanas sobre las que aplicar nuestro algoritmo HOG. Debido al tiempo que nos llevaría todo este proceso, hemos obviado esta parte y hemos entrenado sólo la SVM que en el paper denominan preliminar, aún así los resultados son bastante buenos.

Una vez tenemos la SVM entrenada, sólo debemos pasarle los descriptores HOG de las imágenes de test junto con sus etiquetas al clasificador y ver los resultados. Para

¹Hay pequeños detalles de implementación, como el uso de bordes por temas de tamaño de máscara, que no están detallados en este documento pero si en el código.

decidir como de bueno son dichos resultados, hemos usado la curva ROC. Este el código que hace dicho proceso:

```
1 def plot_classifier(kernel, data_train, labels_train, data_test, labels_test, title,
2                     file):
3     clf = svm.SVC(kernel=kernel)
4
5     start_time = time.time()
6     clf.fit(data_train, labels_train)
7     elapsed_time = time.time() - start_time
8     print("{} seconds".format(elapsed_time))
9     pickle.dump(clf, open("./pickle/" + file + ".p", "wb"))
10    predict = clf.predict(data_test)
11
12    fpr, tpr, thresholds = roc_curve(labels_test, predict, pos_label=1)
13    roc_auc = auc(fpr, tpr)
14
15    plt.figure()
16    patch = mpatches.Patch(color='red', label='ROC_curve.area={},error={}'.format(
17        np.round(roc_auc, 4), np.round(1 - roc_auc, 4)))
18    plt.legend(handles=[patch], loc='lower_right')
19    plt.plot(fpr, tpr, color='red')
20    plt.xlim([0.0, 1.0])
21    plt.ylim([0.0, 1.05])
22    plt.xlabel('False_Positive_Rate')
23    plt.ylabel('True_Positive_Rate')
24    plt.title(title)
25    plt.savefig(file, dpi=700)
# plt.show()
plt.clf()
```

Dicho código aprende una SVM con kernel lineal con los datos de train y luego predice los datos de test. Una vez tenemos las etiquetas predichas, calculamos la curva ROC y la pintamos. Este proceso lo hemos hecho para los 4 tipos de normas que tenemos. Los resultados son los siguientes:

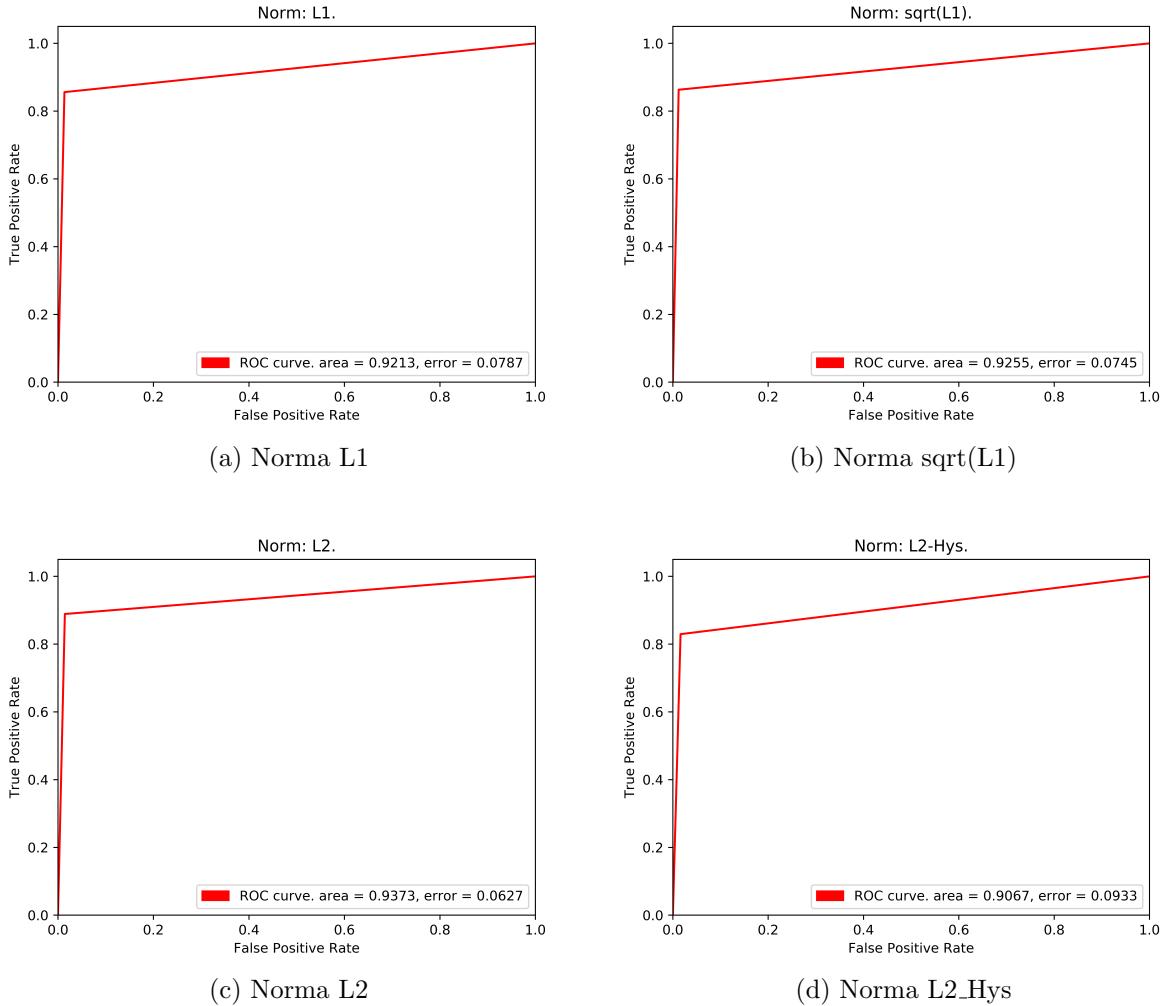


Figura 3.1: SVM lineal.

Como podemos ver los resultados son bastante buenos, obteniendo unas curvas ROC con un área en torno al 0.93, lo que supone un error del 0.07, es decir, un 7 %.

3.1. Sin alisamiento Gaussiano.

Hemos visto que, en un momento determinado del algoritmo, hemos hecho un alisamiento Gaussiano. Este alisamiento es bastante pequeño. Así que hemos probado a realizar los mismo experimentos sin realizar las líneas 28 a 34, es decir, la aplicación del filtro Gaussiano. Los resultados son los siguientes:

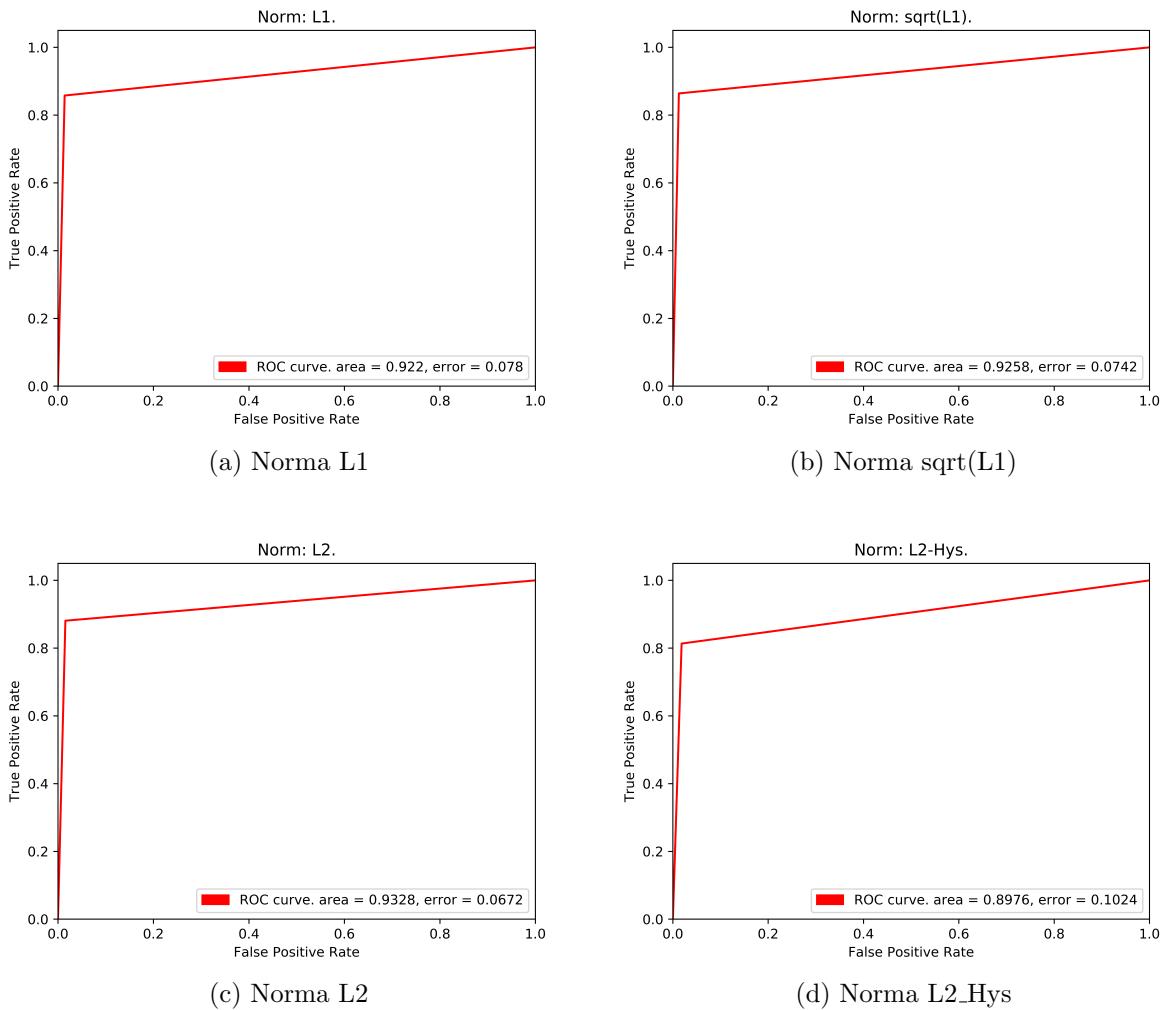


Figura 3.2: SVM lineal sin alisamiento.

Podemos ver que seguimos obteniendo unos resultados bastante buenos y similares a los obtenidos cuando no hacíamos el alisamiento pero algo inferiores. Así que podemos decir que la normalización ayuda algo pero no demasiado a obtener unos mejores resultados.

3.2. Cambio en los tamaños de bloque y celdas.

En los ejemplos mostrados hasta ahora y en los resultados mostrados por el paper, se ha usado un tamaño de celda de 6x6 y un tamaño de bloque de 3x3, pero, ¿es realmente bueno dicho tamaño? Para ello, vamos a ejecutar todo de nuevo pero usando unos tamaños que, según el paper, son los peores y vamos a ver si realmente hay diferencia. En la figura 5 del paper podemos ver que los peores resultados se obtienen con un tamaño de celda de 12x12 y un tamaño de bloque de 1x1. Veamos los resultados:

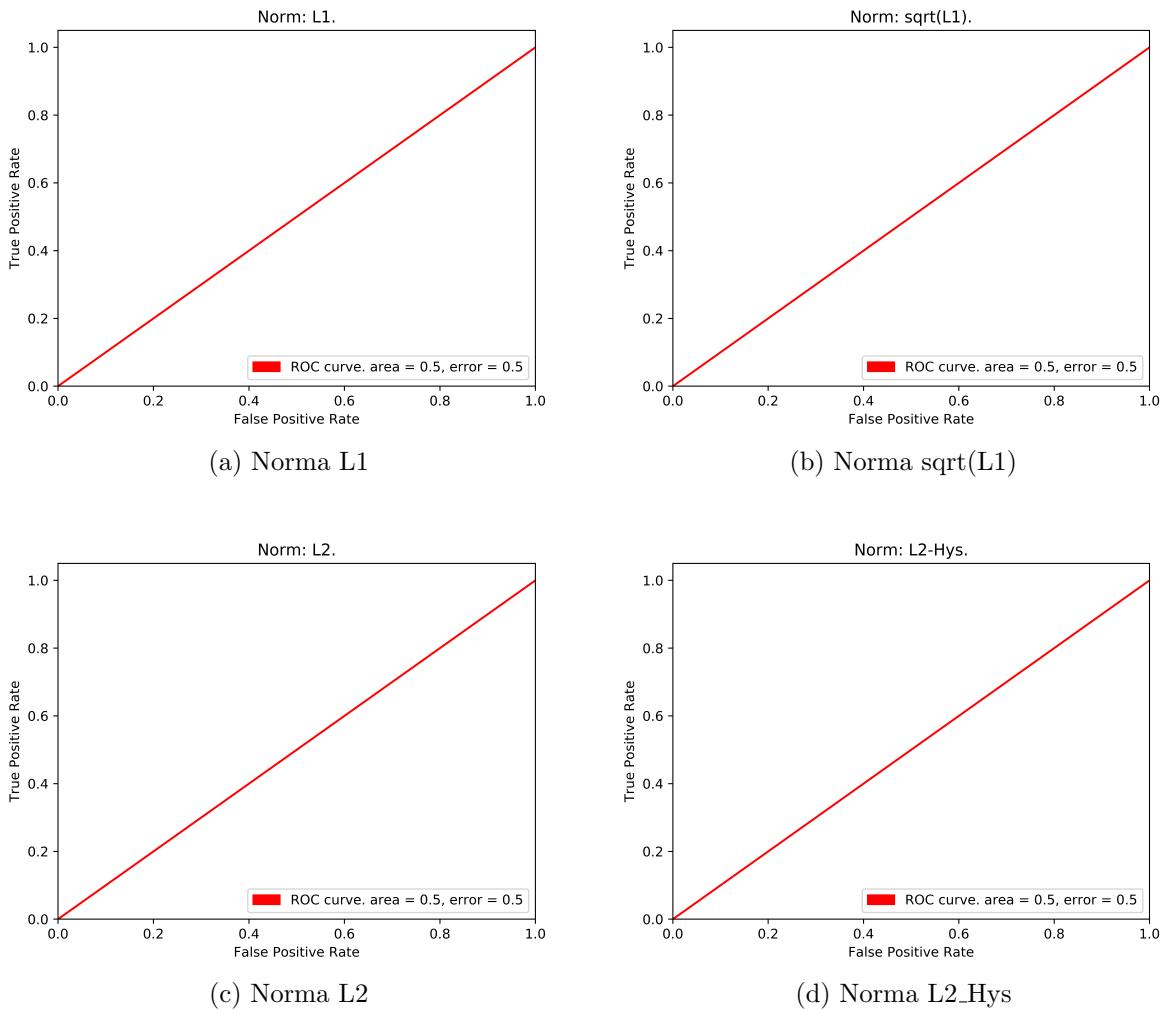


Figura 3.3: SVM lineal: tamaño de celda: 12x12 - tamaño de bloque: 1x1.

Como podemos ver, los resultados son bastante peores que los obtenidos con otros tamaño de bloque y celda.

4. Reconocimiento.

Hasta ahora sólo hemos hecho detección de peatones, es decir, dada una ventana de una imagen, determinar si hay un peatón en ella o no, pero se puede ir más allá. Hemos hecho unas pequeñas pruebas en las que hemos intentado indicar donde hay un peatón para una imagen dada. Para ello hemos hecho el siguiente código:

```

1 def recognition(img, classifier, type_norm, step=4):
2     positive_regions = []
3     pyramid = [img]
4     new_level = img
5

```

```

6   while np.shape(new_level)[0] >= 128 and np.shape(new_level)[1] >= 64:
7       new_level = cv2.GaussianBlur(src=new_level, ksize=(7, 7), sigmaX=1)
8       new_level = cv2.resize(new_level, dsize=(0, 0), fx=0.8333333, fy=0.8333333)
9       pyramid.append(new_level)
10
11      for level, img_pyramid in zip(range(len(pyramid)), pyramid):
12          for i in range(0, np.shape(img_pyramid)[0] - 128, step):
13              for j in range(0, np.shape(img_pyramid)[1] - 64, step*2):
14                  sub_img = cv2.copyMakeBorder(img_pyramid[i:i + 128, j:j + 64], 2, 2, 1,
15                                              cv2.BORDER_REFLECT)
16                  h = hog(sub_img, cell_size=12, block_size=1, type_norm=type_norm)
17                  prediction = classifier.predict(h.reshape(-1, h.shape[0]))
18                  if prediction[0] != 0.0:
19                      positive_regions.append([level, j, i])
20
21  return positive_regions

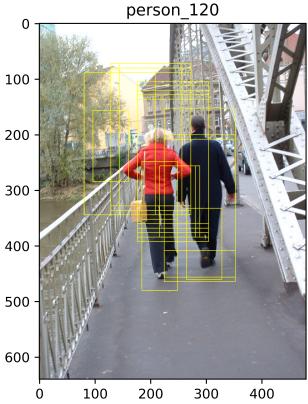
```

La idea es bien sencilla. Cogemos una ventana del tamaño que nuestro algoritmo espera, 128 filas por 64 columnas, y la vamos paseando por toda la imagen preguntando si en dicha ventana hay un peatón o no. Este proceso de preguntar supone extraer el descriptor HOG para dicha ventana y pasarlo al clasificador ya entrenado. Si el clasificador nos dice que sí es peatón, simplemente guardamos el resultado para luego poder pintar los cuadros que indican donde está el peatón. Pero esto tiene un problema, y es que mover la ventana píxel a píxel es muy lento, así que hemos decidido introducir un argumento, *step*, que nos permita indicar cada cuantos píxeles queremos realizar el calculo del descriptor. Dado que nuestra ventana tiene el doble de filas que columnas, dicho incremento es el doble para las columnas. Para los resultados mostrados a continuación hemos usado un *step* de 4, es decir, nos movíamos de 4 en 4 en las filas y de 8 en 8 en las columnas.

Otro detalle a tener en cuenta es que puede ser que el peatón sea más grande que la ventana que estamos usando, y por lo tanto este no quepa en ella y dificulte las tareas de reconocimiento, por lo tanto, el proceso comentado debemos hacerlo en distintos niveles de una pirámide Gaussiana. El paper indica que el factor de escalado para dicha pirámide sea de 1.2. La función *pyrDown* de *OpenCV* no admite un valor menor que 2, así que lo hemos hecho de forma manual, aplicando primero un filtro Gaussiano (bastante suave) y posteriormente un escalado, usando la función *resize* de *OpenCV*, la cuál nos permite indicar el factor de escalado. Dicho factor lo hemos puesto a 0.8333333, es decir, 1/1,2. Finalmente, debemos saber que tenemos que tener tantos niveles de pirámide Gaussiana como sea posible, para asegurarnos de que, en caso de que haya peatón, este quepa en el tamaño de una ventana en alguno de los niveles de la pirámide. Por lo tanto, debemos reducir la imagen hasta que esta sea menor que el tamaño de la ventana.

Una vez tenemos las zonas en las que el clasificador ha dicho que sí hay peatón, sólo debemos pintar un rectángulo en esas coordenadas usando la función *rectangle* de *OpenCV* y teniendo cuidado con dos cosas. La primera de ella es que debemos usar las coordenadas al revés, es decir, el valor de las columnas como *x* y el valor de las filas como *y*. También debemos tener cuidado con el nivel de la pirámide en la que se han obtenidos las coordenadas, ya que, en función de ello, deberemos decidir donde pintar el

rectángulo. Los resultados los podemos ver a continuación:



(a) recog_person_120



(b) recog_person_138



(c) recog_person_198

Figura 4.1: SVM lineal: tamaño de celda: 6x6 - tamaño de bloque: 3x3.

En las figuras mostradas, el tamaño del rectángulo dibujado viene dado por el nivel de la pirámide Gaussiana en el que se ha encontrado, pintando un rectángulo más grande cuanto más alto es el nivel de la pirámide en el que se ha encontrado (es decir, más pequeña es la imagen). El tamaño del cuadrado debe ir variando ya que, si encontramos un peatón en un nivel n , quiere decir que en los niveles superiores el peatón no cabía en la ventana y por lo tanto no se ha reconocido correctamente. Por lo tanto, cuando pintemos el rectángulo, debemos pintarlo de tamaño $64*128$ pero proporcionado al nivel de la pirámide en el que se ha encontrado, para así permitir que dicho rectángulo abarque al peatón entero en el nivel original de la pirámide. Como podemos ver, el resultado es bastante bueno, ya que podemos ver que detecta bastante bien a los peatones. Pero si nos fijamos bien, hay cuadrados que sólo pillan parte del peatón y otros cuadrados

que si pillan al peatón entero. Este proceso se podría mejorar usando la supresión de no máximos, es decir, encontrar una forma de elegir que rectángulo es mejor que otro y, una vez seleccionado, no pintar ningún rectángulo que haga intersección con este. En las imágenes no se ha mostrado este proceso puesto que queremos mostrar todas las regiones encontradas por nuestro algoritmo.

También debemos tener en cuenta que pueden aparecer falsos positivos, es decir, regiones en las que el algoritmo diga que hay peatón cuando realmente no lo hay. Esto se debe a lo que comentamos cuando explicamos las pruebas que íbamos a realizar, y es que no hemos podido hacer (por temas de tiempo) el reentrenamiento de la SVM y, por lo tanto, la clase *no peatón* no se ha aprendido correctamente y puede que, por lo tanto, en zonas donde no hay un peatón, puede ser que nuestro algoritmo diga que sí lo hay.

5. Bibliografía.

- *Histograms of Oriented Gradients for Human Detection - Navneet Dalal and Bill Triggs*
- Documentación de OpenCV.
- INRIA Dataset: <http://pascal.inrialpes.fr/data/human/>