Visión por Computador. Práctica 1.

Antonio Coín Castro

Curso 2019-20

Estructura del código

Se han desarrollado una serie de funciones genéricas de representación de imágenes, extráidas en su mayoría de la práctica 0 (aunque con ligeras modificaciones). Hay una serie de consideraciones a tener en cuenta:

- En general, todas las imágenes se convierten a números reales en el momento en que se leen, y solo se normalizan a $[0, 1]$ cuando se vayan a pintar.
- Las imágenes se pintan usando la librería `matplotlib`, empleando la técnica de *subplots* cuando sea necesario mostrar más de una imagen en la misma ventana.
- Tras mostrar una ventana de imagen, el programa quedará a la espera de una pulsación de tecla para continuar con su ejecución (incluso si se cierra manualmente la ventana del *plot*).
- Hay una serie de parámetros globales (editables) al inicio del programa para modificar ciertos comportamientos de algunas funciones.
- Se trabaja con una única imagen de ejemplo en todo el programa, la cual debe estar en la ruta relativa `imagenes/`.

El programa desarrollado va ejecutando desde una función `main` los distintos apartados de la práctica uno por uno, llamando en cada caso a las funciones que sean necesarias para mostrar ejemplos de funcionamiento. Los parámetros de ejecución se fijan en estas funciones, llamadas `exXY`, donde `X` es el número de ejercicio e `Y` es el apartado.

Bonus 1

Comentamos primero el bonus 1 porque se utiliza en el resto de apartados de la práctica. Se ha implementado una función `convolution2D` que recibe una imagen y un *kernel* bidimensional y se encarga de:

a) Decidir si es separable en dos vectores unidimensionales, y
b) en caso afirmativo, encontrar dichos vectores y aplicar la convolución a la imagen con ellos.

Para la primera parte tenemos en cuenta el siguiente resultado de álgebra lineal:

Proposición. Sea A una matriz real $n \times m$. Entonces, $\text{rango}(A) = 1$ sí y solo sí existen vectores columna no nulos $u, v \in \mathbb{R}^m$ tales que $A = uv^T$.

Demostración. Recordamos que el rango de A puede verse como la dimensión de la imagen de la aplicación lineal asociada $x \mapsto Ax$, para $x \in \mathbb{R}^m$.

Para ver la primera implicación, basta observar que si $A = uv^T$, entonces $Ax = (uv^T)x = u(v^Tx) = \langle v, x\rangle u$. Así, la imagen cae en una recta vectorial, luego tiene dimensión 1.

Recíprocamente, si A tiene rango 1 existe $u \in \mathbb{R}^m$ tal que $Ax = k(x)u$ para todo $x \in \mathbb{R}^m$, con $k(x) \in \mathbb{R}$. Aplicando esto a los vectores de la base usual de $\mathbb{R}^m$ obtenemos que cada columna de A es un múltiplo de u, digamos $A = u(k_1 \cdots k_m)$. Tomando $v = (k_1 \cdots k_m)^T$ tenemos que $A = uv^T$, como queríamos. □

Aprovecharemos también la descomposición en valores singulares de una matriz real [**1**]. Esta descomposición nos permite escribir $A = U\Sigma V^T$, donde U, V son matrices ortogonales y Σ es una matriz (rectangular) diagonal con valores no negativos. La propiedad fundamental que usaremos de esta descomposición es que el rango de A coincide con el número de elementos no nulos en la diagonal de Σ. En el caso en que A tenga rango 1, las primeras columnas de U y V (convenientemente ordenadas) serán los vectores que forman la descomposición de A como producto de dos vectores (el resto de columnas son nulas).

Utilizando esta descomposición, podemos a la vez comprobar que el *kernel* tiene rango 1 y encontrar los dos vectores por los que se factoriza. Empleamos la función `linalg.svd` de la librería `numpy` como sigue:

```
u, s, v = np.linalg.svd(kernel)
rank = np.sum(s > EPSILON)
if rank == 1:
  vx = u[:,0]
  vy = v[:,0]
```

Notamos que hemos considerado que los elementos de Σ son 0 si están por debajo de un cierto umbral, para subsanar posibles errores de redondeo a la hora de calcular la descomposición SVD.

Disponemos también una función `separable_convolution2D` que realiza la convolución con una imagen de dos vectores `vx` y `vy`, que representan las máscaras que se pasan por filas y por columnas (deben ser siempre de longitud impar). A la hora de implementarla, se realiza primero la convolución por filas y luego por columnas usando la función `convolve`. Esta función implementa la fórmula de la convolución de dos vectores `a` y `v`, y devuelve un vector convolucionado de la misma longitud que `a`:

$$(a * v)[n] = \sum_{i=-M}^{M} a[i]v[n-i].$$

Para intentar que sea eficiente, se construye una matriz `A` que se multiplica por el vector de máscara `v` recorrido al revés, de acuerdo con la fórmula anterior. Tenemos en cuenta que para 'encajar' los dos vectores y multiplicar debemos rellenar con 0 a ambos lados del vector `a`, tantos como indique la mitad de la longitud de la máscara (división entera). La construcción de la matriz `A` es como sigue:

```
for i in range(len(a)):
    rows.append(a[i:i + len(v)])
A = np.array(rows, dtype = np.double)
```

El último detalle que queda por mencionar es la estrategia seguida en los bordes de la imagen. Se proporciona una función `make_border` que devuelve una imagen ampliada con tantas filas y columnas de borde como sean necesarias, según la longitud de la máscara. Para ello se emplean las funciones `vstack` y `hstack` de `numpy`. En principio se han programado dos estrategias: rellenar los bordes con un valor constante (`BORDER_CONSTANT`) o replicar el borde a partir del último píxel (`BORDER_REPLICATE`).

En el caso de que la imagen sea una matriz tribanda, se realiza la convolución por separado en cada banda siguiendo el procedimiento mencionado, mediante la función `channel_separable_convolution2D`.

Ejercicio 1

Apartado A

En este apartado mostramos el efecto de alisamiento con un *kernel* Gaussiano en una imagen. Como sabemos que la función Gaussiana es separable, basta considerar dos máscaras unidimensionales dadas por la función

$$G(x, \sigma) = exp\left\{\frac{-x^2}{2\sigma^2}\right\}.$$

Definimos entonces una función `gaussian_kernel1D` que nos devuelve una máscara Gaussiana unidimensional a partir de un parámetro σ (la desviación típica). Para ello, muestreamos los enteros del intervalo $[-3\sigma, 3\sigma]$, donde es conocido que se concentra casi toda la densidad de la función.

```
l = floor(3 * sigma)
gauss_ker = [gaussian(x, sigma) for x in range(-l, l + 1)]
gauss_ker = gauss_ker / np.sum(gauss_ker)
```

Es importante notar que devolvemos una máscara **normalizada**, en el sentido de que la suma de sus elementos es 1.

Mostramos ahora algunos ejemplos de convolución con máscaras Gaussianas (obtenidos a partir de la función `separable_convolution2D`), de forma que el alisado es isotrópico.

Figura 1: Ejemplos de convolución con máscaras Gaussianas.

Vemos que, como era de esperar, a mayor valor de σ mayor emborronamiento se produce, pues se eliminan más altas frecuencias de la imagen perdiendo detalles en el proceso. Además, observamos que el tipo de borde empleado influye en el resultado final, pues en la alternativa de borde constante se aprecia un borde negro en la imagen resultante que no aparece en la alternativa con borde replicado.

Pasamos ahora a realizar convoluciones con máscaras de derivada. Empleamos la aproximación por diferencias finitas para calcular las derivadas parciales de una imagen $f(x, y)$:

$$\frac{\partial f(x,y)}{\partial x} \approx f(x+1, y) - f(x, y).$$

Análogamente se puede aproximar la derivada con respecto a y, e iterando el proceso obtenemos derivadas de cualquier orden. Las máscaras de derivadas se obtienen mediante una llamada a la función de *OpenCV* `getDerivKernels`, que recibe como parámetros el orden de derivación en cada variable y el tamaño de las máscaras deseado. Internamente devuelve máscaras de Sobel para realizar la convolución **[2]**, que estarán normalizadas (mediante el parámentro `normalize = True`).

Obtenemos la convolución con máscaras de derivadas mediante una llamada a la función `derivatives2D`. Un detalle a tener en cuenta es que se devuelve el valor absoluto de la imagen resultante, pues lo que nos interesa son los valores altos, tanto negativos como positivos (no queremos perder los valores negativos en la normalización posterior).

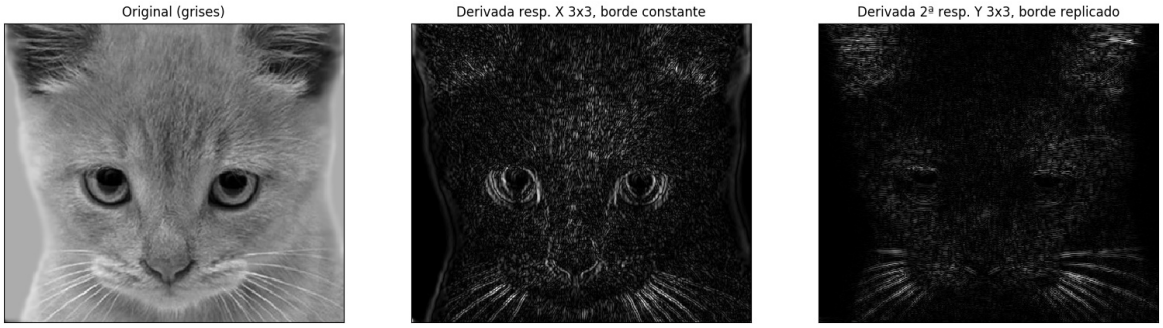

Figura 2: Ejemplos de convolución con máscaras de derivada.

Vemos que en la derivada respecto de x se aprecian los cambios en la dirección vertical, y que al aplicar la segunda derivada respecto de y se aprecian los cambios más bruscos de la imagen en la dirección horizontal. En este caso el efecto del borde apenas es visible sobre el fondo negro.

Apartado B

En este apartado perseguimos mostrar el resultado de aplicar un filtro Laplaciana-de-Gaussiana a una imagen. Este filtro consiste en aplicar primero un alisado Gaussiano para eliminar ruido y mejorar la visualización de la imagen, y después aplicar el operador Laplaciano. Si G denota el alisamiento Gaussiano, el resultado es:

$$\Delta(f) = G(f)_{xx} + G(f)_{yy}$$

La implementación dada esta fórmula es una consecuencia directa de lo que se vio en el apartado anterior:

```
im_smooth = gaussian_blur2D(im, sigma)
vxx, v = get_derivatives2D(2, 0, size)
u, vyy = get_derivatives2D(0, 2, size)
im1 = separable_convolution2D(im_smooth, vxx, v)
im2 = separable_convolution2D(im_smooth, u, vyy,)
laplacian = im1 + im2
```

Mediante este filtro podemos detectar algunos bordes, pues en los sitios donde haya un gran cambio de intensidad en una dirección, el operador será negativo a un lado y positivo al otro. De nuevo visualizamos el valor absoluto de la imagen resultante para no perder los valores negativos de cambio.

Notamos que este filtro ya está normalizado en escala, al estarlo el filtro Gaussiano que aplicamos inicialmente.

En este caso observamos que el efecto del borde constante es bastante visible. Una cosa curiosa es que en este caso la imagen se ve más oscurecida, dando la sensación de que se extiende el efecto del borde al resto de la imagen. En principio creo que esto no debería ocurrir, pero si utilizamos la función `Laplacian` de *OpenCV* vemos que el resultado es similar.

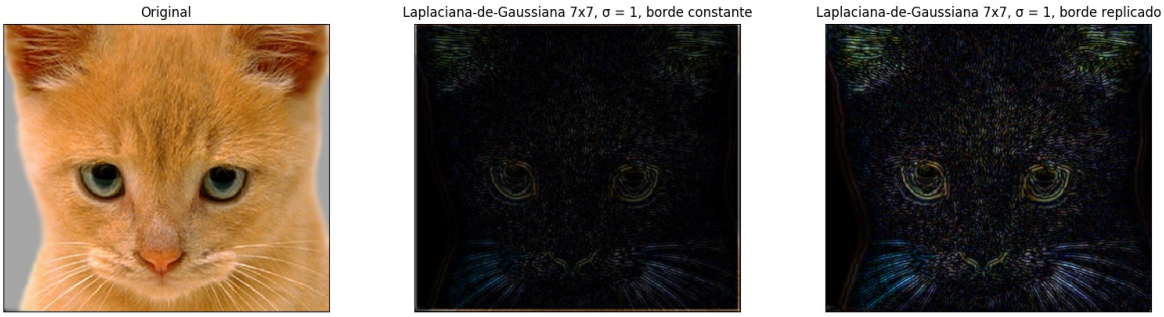

Figura 3: Ejemplos de filtrado con Laplaciana-de-Gaussiana.

Ejercicio 2

Apartado A

Queremos implementar ahora la visualización de la pirámide Gaussiana de una imagen, simulando cómo veríamos la imagen a distancia. Para implementarla, reducimos el tamaño de la imagen una octava cada vez hasta llegar al número de niveles deseado, alisando previamente con un filtro Gaussiano que elimine el ruido progresivamente.

Esta funcionalidad se implementa en la función `gaussian_pyramid`, que se apoya en la función `blur_and_downsample` para ir reduciendo el tamaño de la imagen. La reducción se consigue eliminando de la imagen las filas y las columnas pares:

```
nrows, ncols = im.shape[:2]
im_downsampled = np.array([im[i] for i in range(1, nrows, 2)])
im_downsampled = np.transpose([im_downsampled[:, j] for j in range(1, ncols, 2)])
```

Para la visualización de la pirámide se utiliza la función `format_pyramid`, inspirada en un ejemplo de la documentación de la librería `scikit-image` [**3**]. La estrategia consiste en construir una matriz que actúe como marco donde iremos incrustrando cada una de las imágenes que forman la pirámide. En principio se permite que el factor de reducción de una imagen a la siguiente sea un número real `k`, por lo que debemos adaptar el marco para que pueda almacenar todas las imágenes de la pirámide (añadiendo filas y columnas extra):

```
diff = np.sum([im.shape[0] for im in vim[1:]]) - nrows
extra_rows = diff if diff > 0 else 0
extra_cols = int(ncols / k)
```

Como podemos apreciar en la Figura 4, la primera imagen es la imagen original, y cada nueva imagen en la pirámide se ve más emborronada, reteniendo efectivamente las frecuencias bajas en cada paso como se esperaría. El efecto del borde constante también es visible, como ya lo era al aplicar un solo alisado Gaussiano a la imagen. En el último nivel ya apenas podemos apreciar detalles, debido también al tamaño tan pequeño que tiene la imagen.

Apartado B

Ejercicio 3

Bibliografía

[**1**] Singular value decomposition

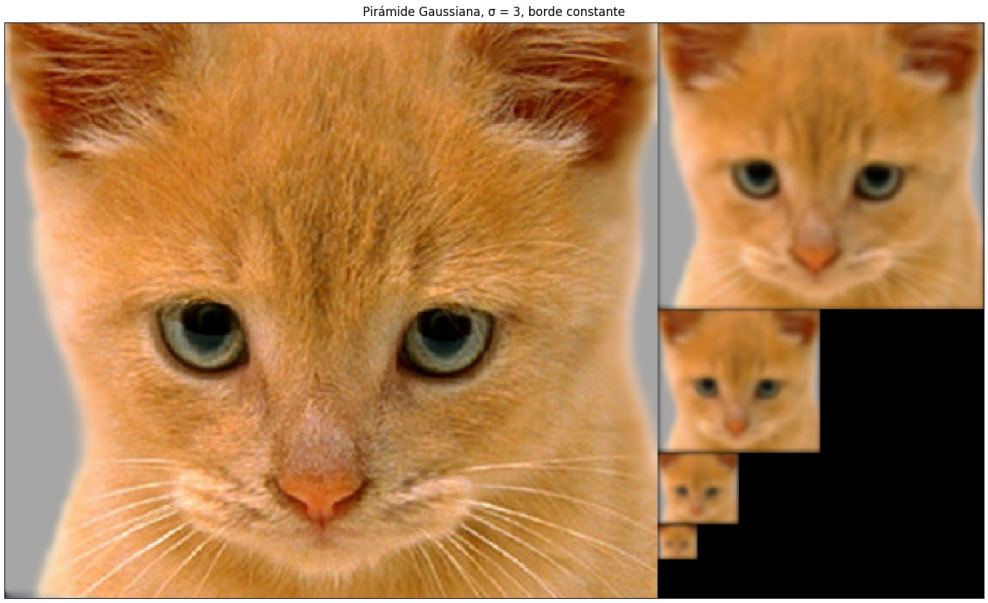

Figura 4: el pspspsps

[**2**] Sobel filter

[**3**] Ejemplo de pirámide de scikit-image