Universidad Europea

Memoria Práctica 1

Martín de las Heras

Introducción

En esta práctica hemos aprendido a procesar imágenes a bajo nivel, con el objetivo de aprender cómo funciona la operación de convolución y los filtros. Concretamente, se divide en estos apartados:

- Generar una imagen de 100 x 100 pixeles con fondo negro sobre la que aleatoriamente se pinta una gaussiana centrada en un punto aleatorio y ancho (sigma) 10 pixeles.
- Leer la imagen y mediante la definición de un *kernel* ajustar a una Gaussiana 2D sobre la que se debe determinar como parámetro libre el factor de proporcionalidad entre la gaussiana del *kernel* y la gaussiana de la imagen (parámetro A) así como el centro de la gaussiana detectada en la imagen.
- Repetir los puntos anteriores pero generar al menos 4 niveles de ruido "*shot noise*" aditivos a la imagen y estudiar cómo el parámetro A y el nivel de ruido. Al menos uno de los niveles de ruido debe ser tan alto que no permita detectar nada en la imagen.

Generación de la imagen

En primer lugar, para generar la imagen, hemos definido los parámetros indicados (XSIZE x YSIZE con valores 100 x 100 y un *sigma* de 10) y definido un centro y un parámetro A aleatorios:

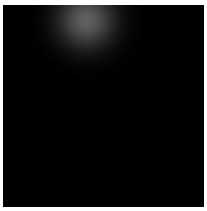
```
x_center_original = random.randrange(XSIZE)
y_center_original = random.randrange(YSIZE)
A = random.randint(1, 255)
```

Para ello definimos una función que devuelva el array de valores gaussianos en función de la distancia del punto en cuestión al centro definido, normalizando los valores para que el valor máximo sea A:

```
gaussian = np.exp(-((x - x_center)**2 + (y - y_center)**2) / (2 * sigma**2))
normalized_gaussian = gaussian / np.max(gaussian) * A
```

Por temas de eficiencia se le pasan todos los valores de golpe, de esta manera hace el cómputo directamente en vez de calcular dentro de una doble iteración.

También definimos otra función que toma ese valor y lo transforma en una imagen, redimensionando el array y apilándolo en los 3 canales de color de la imagen, para luego generar una imagen a partir de ese nuevo tensor. Obtenemos como resultado la siguiente imagen:



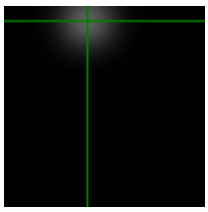
Determinación del centro y parámetro A

A continuación procedemos con la búsqueda del parámetro A y el centro mediante una función *kernel*. Para ello en primer lugar definimos el tensor de distancia y de soluciones, donde se van a almacenar los parámetros que determina la función *kernel* para cada una de las iteraciones que veremos a continuación. Para ello, iteramos por toda la imagen, creando una gaussiana de la misma manera que hemos visto antes, solo que sin definir el parámetro A y restándole la imagen original. Mediante una función de SciPy, buscamos el valor óptimo de ese parámetro A que minimice dicha resta para el píxel concreto sobre el que estamos iterando. Una vez lo tenemos, guardamos el resultado en los tensores que hemos definido antes:

```
for x_pixel in range(XSIZE):
    for y_pixel in range(YSIZE):
        def func(A):
            return np.sum(np.abs(gaussian_2d(A, x_pixel, y_pixel) - gaussian))
        opt = scipy.optimize.least_squares(func, [255])
        distance[x_pixel][y_pixel] = opt.cost
        solutions[x_pixel][y_pixel] = opt.x
```

Una vez hemos terminado, determinamos que los valores del centro y A vienen determinados por la matriz de distancias, donde el elemento con menor distancia es el que más se ha podido ajustar a los datos generados en el *kernel*. También generamos una mapa de calor para ilustrar las distancias recogidas por el proceso:

Para ilustrar el punto detectado por el proceso, hemos pintado una cruz verde sobre el centro original y una roja sobre el detectado. En este caso, al haber acertado se encuentran superpuestas, pero más adelante veremos que no es siempre este el caso. Cabe destacar que siempre es capaz de predecir el valor de A correctamente.



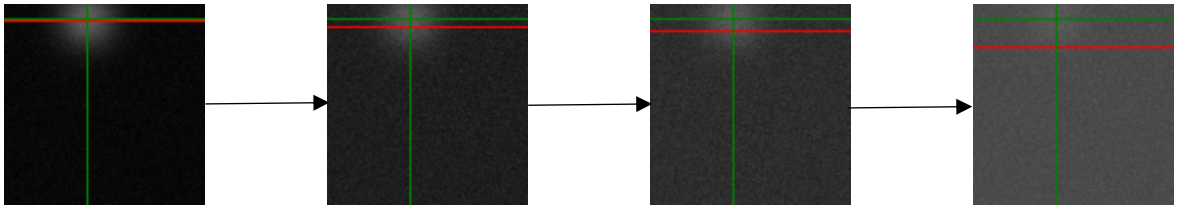
Detección con ruido

Por último, a la imagen original, le añadimos cuatro niveles de ruido de Poisson, para ver cómo empeora la detección. Para ello, simplemente generamos un ruido aleatorio con dicha distribución y se lo sumamos a la imagen. Respetando siempre la escala A:

```
noise_levels = [10, 50, 100, 500]
print(f'Now we add Poisson noise to the image in 4 levels: {noise_levels} and try again.')

for level in noise_levels:
    noise = np.random.poisson(level, XSIZE*YSIZE)
    noisy_gaussian = (gaussian + noise) / np.max(gaussian + noise) * A
```

A estas imágenes les aplicamos el mismo proceso de antes y vemos cómo van fallando cada vez más:



Conclusiones

En esta práctica hemos visto cómo hacer a bajo nivel operaciones con los *kernels*, así como ver el efecto que el ruido tiene frente a este tipo de problemas. Esto nos ha aportado una buena base de cara a entender los siguientes conceptos de la asignatura, ya que los conceptos generales se mantienen a lo largo del campo de la visión por ordenador.