UNIVERSIDAD
DE GRANADA

Práctica 1.
Filtrado y Detección de regiones

Lukas Häring García

October 22, 2019

Tabla de contenidos

1 Primer Apartado

En este apartado vamos a realizar unos ejercicios relacionados con las máscaras gaussianas y las laplacianas-gaussianas estudiadas en clase, con diferentes ejemplos de imágenes, bordes y sigmas.

1.1 Máscaras Gaussianas

Una máscara gaussiana es un operador de convolución que se utiliza para suavizar superficies, esta es obtenida a través del muestreo de una función gaussiana.

Vamos a convolucionar nuestra imagen con una máscara gaussiana 2D. Para ello, basta con utilizar el método **GaussianBlur** de la librería **cv2**.

Este necesita como parámetros, la imagen a aplicar; el segundo parámetro es un par de valores para el tamaño del kernel (El cual no utilizaremos ya que lo haremos dependiente del sigma); tercer y cuarto parámetro, el valor de los sigmas horizontal y vertical respectivamente y finalmente el tipo de borde que queremos aplicar, véase la referencia [1] para obtener más información sobre el tipo de bordes.

```
cv2.GaussianBlur(img,None,sigma,sigma,cv2.BORDER_CONSTANT)
```

Podemos obtener el mismo resultado a través de dos convoluciones 1D, para ellos, vamos a obtener el kernel 1D utilizando la función de cv2 **getGaussianKernel**, este nos pedirá como argumentos: El tamaño del kernel y el sigma. El tamaño del kernel es calculable a partir del sigma.

$$\#\text{Kernel} = 2 \cdot \lfloor 3\sigma \rfloor + 1$$

Esta ecuación es deducible a través del tamaño del intervalo y de la necesidad de muestrear un $\geq 95\%$ de la curva.

Para convolucionar ambos filtros 1D, utilizo el método diseñado por mí, que también es parte del **Bonus 1** y será explicado posteriormente. Este calcula eficientemente dicha convolución, recibe el nombre de **conv_1D_1D**. **conv_1D_1D** necesita como argumentos, la imagen y una matriz con las dos máscaras 1D. Además hace uso del principio de localidad, por lo que la primera convolución (horizontal) la realiza como tal y la convolución vertical, a la imagen transpuesta (y su posterior transposición). Esta función no devuelve el resultado normalizado, por lo que habrá que normalizarlo $[0, 255]$ **normalize** (Véase el pie de página).

```
gaussian_2_1d_cv2 = normalize(conv_1D_1D(img, kernel))
```

1.1.1 Resultados

A la izquierda el método **GaussianBlur** y a la derecha, mi función de convolución **conv_1D_1D**.

Fig. 1: **image** = "fish.bmp", **border** = "constant", $\boldsymbol{\sigma} = 2.0$

Fig. 2: **image** = "cat.bmp", **border** = "reflect", $\boldsymbol{\sigma} = 4.0$

Fig. 3: **image** = "bicycle.bmp", **border** = "replicate", $\boldsymbol{\sigma} = 6.0$

1.1.2 Valoración

Podemos observar una clara diferencia entre ambas imágenes, algo es extraño ya que **GaussianBlur** para "bicycle.bmp" y para "cat.bmp" parece no aplicar correctamente el filtro gaussiano y solo horizontalmente. Por otro lado, mi método consigue aplicar correctamente un difuminado homogéneo en la bicicleta. Además se observa que en la primera imagen, mi método "arrastra" el padding hacia el centro de la imagen, creando un difuminado obscuro al rededor de los bordes. Está claro que a medida que vamos subiendo $\boldsymbol{\sigma}$, este va difuminando cada vez más, por lo que los resultados, parecen ser coherentes.

1.2 Filtro Laplaciano-Gaussiano

Un filtro laplaciano-gaussiano es utilizada para marcar regiones con alto cambio de intensidad, por ello, es utilizado habitualmente para detectar bordes. Además es isotrópica, quiere decir que afecta por igual en todas direcciones.

La laplaciana-gaussiana, como su propio nombre indica, requiere de dos pasos. El primer paso, aplicar un **filtro Gaussiano** de suavizado (utilizado para eliminar ruido que pueda afectar). En segundo lugar, un **filtro Laplaciano**, que viene dado por la siguiente ecuación.

$$L(x, y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

Como podemos apreciar, la derivada lo que va a ver es el cambio en cada dirección (horizontal y vertical), de ahí su detección de bordes.

Una vez aplicado el suavizado, aplicaremos el Laplaciano en ambas direcciones a la imagen suavizada por separado y luego, sumaremos los resultados, obteniendo así el resultado final.

Para obtener las segundas derivadas espaciales, vamos a utilizar el método de cv2 **getDerivKernels**.

```
d2x = cv2.getDerivKernels(2, 0, tam)
d2y = cv2.getDerivKernels(0, 2, tam)
```

Este nos devolverá un filtro de Sobel[3] para cada derivada espacial, estos los aplicaremos con mi método **conv_1D_1D** y luego, sumaremos los resultados.

Cabe destacar que los filtros no están normalizados, por lo que deberemos multiplicar al resultado final por $\boldsymbol{\sigma}$. Además, el resultado no estará normalizado en 255, por lo que a la hora de mostrarlo, habrá que normalizar.

1.2.1 Resultados

A la izquierda el laplaciano-gaussiano sin valor absoluto. A la derecha, aplicando el valor absoluto.

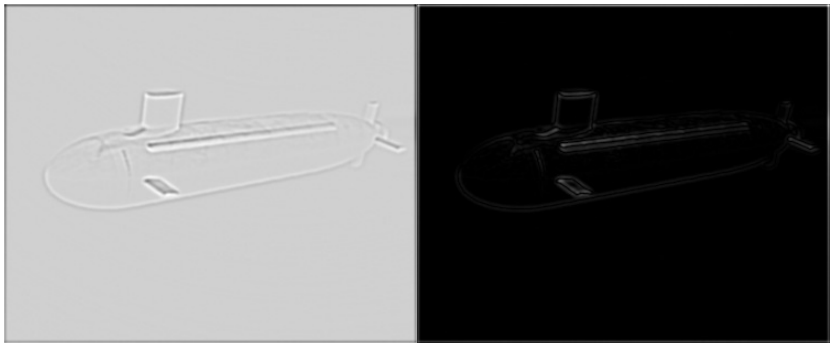

Fig. 4: **image** = "submarine.bmp", **border** = "constant", $\boldsymbol{\sigma} = 1.0$

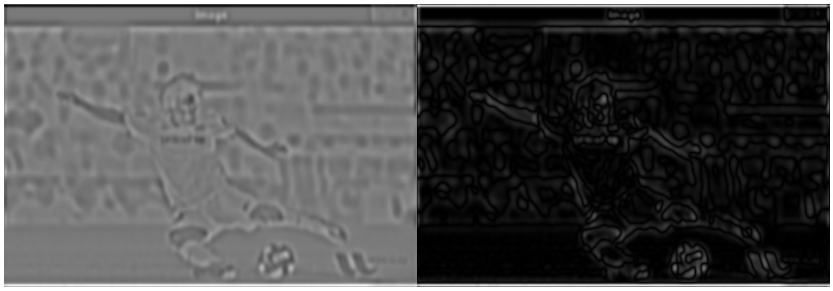

Fig. 5: **image** = "messi.jpg", **border** = "wrap", $\boldsymbol{\sigma} = 2.0$

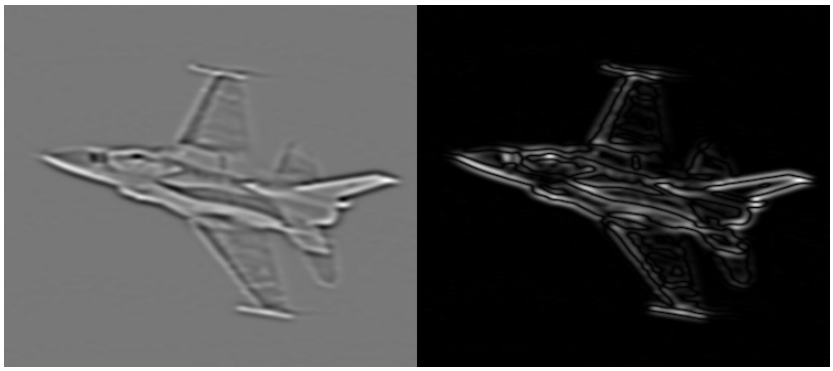

Fig. 6: **image** = "plane.bmp", **border** = "replicate", $\boldsymbol{\sigma} = 3.0$

1.2.2 Valoración

El filtro Laplaciano-Gaussiano, como ya hemos comentado, suaviza por lo que vamos a eliminar información útil o no (ruido).

Es curioso observar que al aplicar el borde "contant", este tome una intensidad baja, esto ocurre ya que al aplicar las derivadas, estos puntos se detecten como bordes ya que su exterior es cero. Podríamos decir que se obtienen "mejores" resultados con cualquier otro borde.

Además, el sigma vemos que afecta a la intensidad de los bordes así como al difuminado de estos, por lo que, podríamos afirmar que debemos utilizar un valor intermedio, para no generar bordes muy borrosos pero tampoco demasiado claros.

Algo a comentar es que el filtro gaussiano utiliza el mismo sigma que el filtro Laplaciano, algo que se podría evitar añadiendo un argumento más, pero es solo un detalle de implementación.

2 Segundo Apartado

En esta sección vamos a ver las pirámides gaussianas y laplacianas. También el llamado **blob detection**, detección de zonas interesantes.

2.1 Pirámide Gaussiana

La pirámide gaussiana es utilizada para escalar imágenes sin que pierda información vital de la imagen, ya que utilizando solo interpolación, suele perder, puntos clave, dirección de lineas, etc.

Además la pirámide gaussiana es utilizada por la GPU para generar texturas con distinto nivel de detalle, conocido como *LOD Textures*[4] o Mipmaps.

Mi método **gauss_pyramid** hace uso de cuatros parámetros: el primero, la imagen; el número de escalas (Se cuentan n + 1 escalas) como segundo argumento; El valor de sigma y por último, el tipo de borde (por defecto, borde constante). Este insertará la primera imagen como guía para las demás.

```
gauss_pyramid(img, 4, sigmam, cv2.BORDER_CONSTANT)
```

Además, hace uso de los métodos **gauss_pyramid_helper** y **GSharp**, que se explicarán a continuación. En primer lugar, el método genera una image con la misma altura que la imagen que pasamos y un ancho de 1.5 veces más.

GSharp un método que devuelve la imagen suavizada y luego re-escalada al tamaño (width, height). El suavizado es realizado a través del método **GaussianBlur** y el re-escalado a través de la función de cv2, **resize** que utiliza como tipo de interpolación **INTER_LINEAR** de **cv2**.

```
filter = GSharp(sourc, height, width, sigma);
```

gauss_pyramid_helper Es un método recursivo que requiere la imagen resultante creada por **gauss_pyramid**, la imagen a re-escalar recursivamente; el desplazamiento horizontal y vertical, utilizado para posicionar correctamente cada imagen; el nivel actual de profundidad, el máximo nivel y el sigma que se va a aplicar.

2.1.1 Resultados

Fig. 7: **image** = "motorcycle.bmp", **border** = "reflect", $\boldsymbol{\sigma} = 1.0$

Fig. 8: **image** = "messi.jpg", **border** = "constant", $\boldsymbol{\sigma} = 2.0$

2.1.2 Valoración

Para la *Fig. 7* se ha utilizado un sigma pequeño $\boldsymbol{\sigma} = 1.0$, ya que este contiene bastantes detalles que no queremos perder a la hora de suavizar mucho, vemos que para cada escala obtenemos una imagen con detalles suficientes, incluso la rueda y los cables se pueden notar suficientemente.

La segunda imagen *Fig. 8*, tenemos que el sigma utilizado es demasiado alto, ya que el jugador en cada escala parece acoplarse aún más al fondo, por lo que, en mi opinión el sigma debería ser más bajo, que a diferencia del anterior, hasta la última imagen puede diferenciarse qué objeto es.

Podemos ver que el borde no afecta al resultado, por lo que cojamos cual cojamos, no va a implicar un gran cambio entre las escalas.

2.2 Pirámide Laplaciana

La pirámide laplaciana es una técnica utilizada para comprimir imágenes, reconocer patrones, etc. Como hemos visto en las diapositivas, la imagen es escalada la mitad (1 octava), previamente se suavizada. Ahora se aplica la operación inversa, es decir, se suaviza y se escala hacia arriba. Una vez tenemos las dos imágenes, se aplica el operador de sustracción sobre la original, obteniendo así, los contornos de la imagen. Este proceso es realizado para cada imagen que hemos re-dimensionado.

De manera similar a la pirámide gaussiana, necesitamos de dos funciones, la especificada anteriormente, **GSharp** y una de ayuda, **laplacian_pyramid_helper**. Almacenamos en una variable la imagen re-escalada a la mitad, para así poder enviarla para la siguiente iteración.

```
width = np.uint(img.shape[1] * 0.5)
heihgt = np.uint(img.shape[0] * 0.5)
filtering = GSharp(img, width, height, sigma, border)
```

Ahora, obtenemos el resultado de la iteración (contornos), aplicando el proceso inverso sobre la imagen original.

```
width = img.shape[1]
heihgt = img.shape[0]
img -= GSharp(filtering, width, height, sigma, border)
img = normalize(abs(img) if absolute else img, True)
```

Como vemos, hemos normalizado la imagen, podemos darle valor absoluto para así tener una vista más clara de los bordes.

Cabe destacar que se han omitido muchos pasos ya que son equivalentes a la pirámide gaussiana que ha sido explicada en el apartado anterior.

2.2.1 Resultados

A la izquierda, la pirámide laplaciana con valor absoluto. A la derecha, sin el valor absoluto. Ambos interpolan como está especificado en el método **GSharp**.

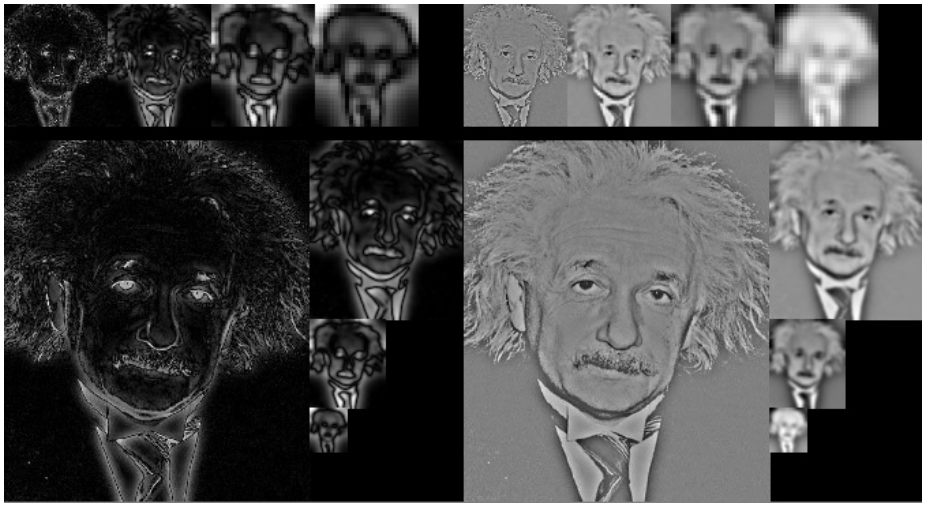

Fig. 9: **image** = "einstein.bmp", **border** = "constant", $\boldsymbol{\sigma} = 2.0$

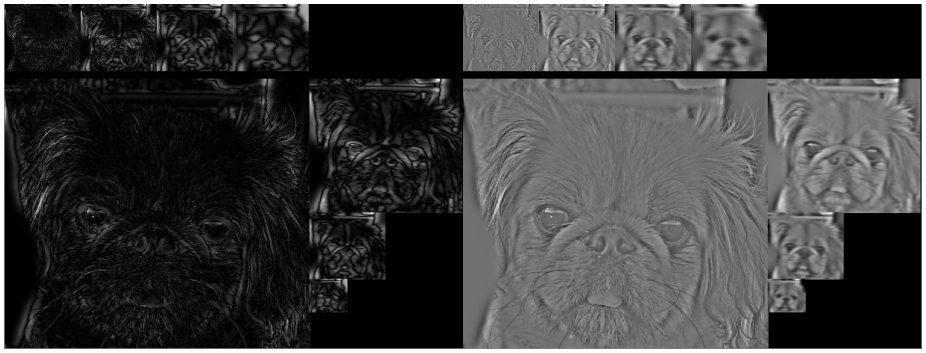

Fig. 10: **image** = "dog.bmp, **border** = "replicate", $\boldsymbol{\sigma} = 3.0$

2.2.2 Valoración

Podemos observar un resultado similar a la laplaciana-gaussiana, como para cada escala, vamos suavizando, cada vez perdemos más información (útil o no). Además, para cada escala, se acentúan unos bordes y en otras escalas, otros.

Para la primera imagen, observamos que con un $\boldsymbol{\sigma} = 2.0$, la primera iteración da mucho ruido al rededor del pelo, pero a medida que iteramos, ese ruido desaparece. Vemos que la corbata, los ojos y el bigote están siempre presentes, eso quiere decir que tienen una alta intensidad por lo que son los elementos que más destacan.

La imagen del perro tiene un efecto parecido a la anterior, la imagen tiene mucho ruido debido al pelo, pero de la misma forma, cada octava reduce aún más ese ruido. Además el elemento que más podemos ver que es destacado son los ojos y el hocico en más iteraciones.

Cabe destacar que el tipo de borde no afecta al resultado, por lo que da igual el borde que apliquemos, podría omitirse o utilizar el borde constante.

2.3 Detección de regiones

3 Pié de página

3.1 Normalización a 255

El método **normalize** recibe una matriz como argumento, esta devuelve una matriz con valores entre 0 y 255, además tiene un parámetro extra (**False** si no está definido) que identifica dos casos:

1. **Normalización completa** (Segundo argumento vale **True**).
   Esto quiere decir que, normaliza tanto valores fuera como dentro del intervalo.

2. **Normalización de fronteras** (Segundo argumento vale **False**).
   Normalizará únicamente valores exteriores al intervalo, tanto por encima como por debajo de este.

Por lo que calcula el máximo (max) y el mínimo (min) de la región (matriz) y aplica la siguiente función $f : \mathbb{R} \to \mathbb{N}$ que viene a continuación.

$$f(x) = \left\lfloor \frac{x - min}{max - min} \cdot 255 \right\rfloor$$

Este mapeo, nos devolverá una matriz cuyos valores están siempre en el intervalo $[0, 255]$ en valores enteros. Cabe destacar que también funciona para imágenes de tres canales, aplicando la misma estrategia para cada canal.

Referencias

[1] cv2 Gaussian Blur Function in Python
https://www.tutorialkart.com/opencv/python/opencv-python-gaussian-image-smoothing/

[2] Laplaciana Gaussiana
https://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm

[3] Deriv Kernel
https://docs.opencv.org/2.4/modules/imgproc/doc/filtering.html#getderivkernels

[4] Level of detail Textures
https://realazthat.neocities.org/demos/2016-08-17/glsl-gaussian/glsl-gaussian-suite/glsl-gaussian-suite.html

[5] Laplacian Pyramid Compression
https://www.sciencedirect.com/topics/engineering/laplacian-pyramid