Open in Colab

Maestría en Inteligencia Artificial Aplicada

TC 4033: Visión computacional para imágenes y video

Tecnológico de Monterrey

Dr. Gilberto Ochoa Ruiz

4. Image Convolution

Equipo # 16

Edwin David Hernández Alejandre A01794692

Miguel Guillermo Galindo Orozco A01793695

Jorge Pedroza Rivera A01319553

Juan Carlos Alvarado Carricarte A01793486

Gerardo Aaron Castañeda Jaramillo A01137646

Table of Contents

Importing Libraries

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

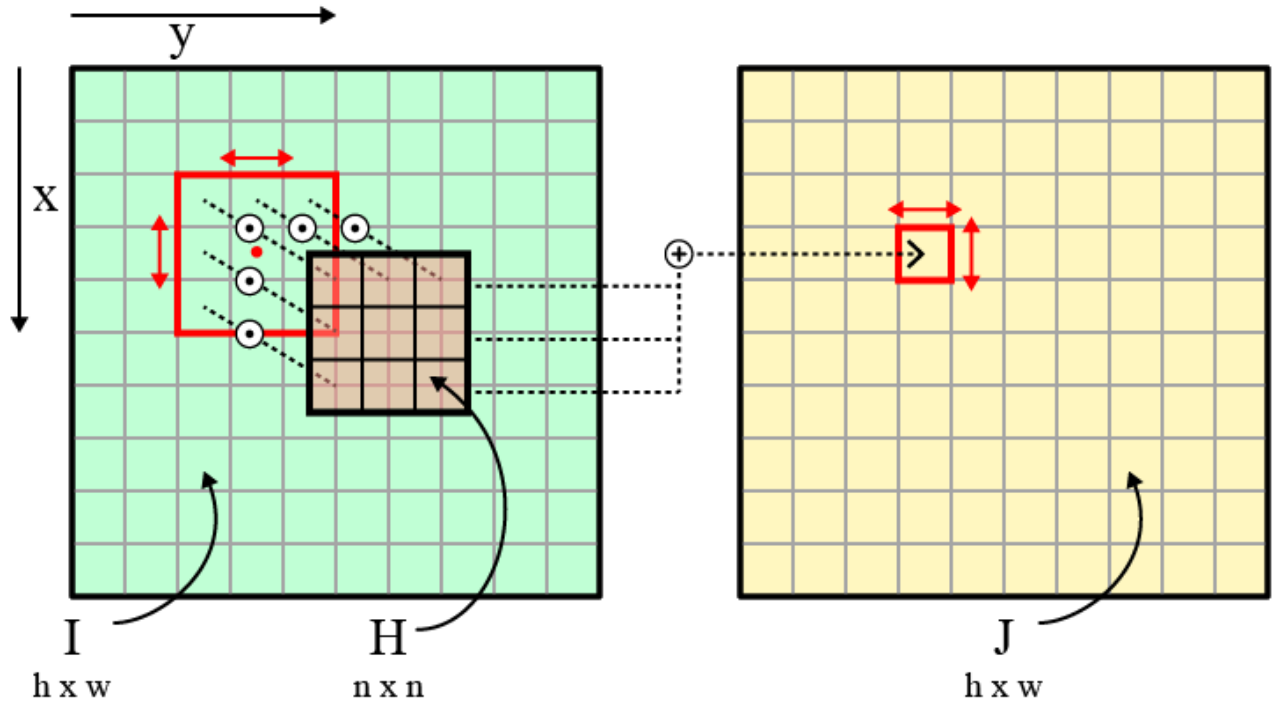
In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image, ImageFilter

import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt
import cv2

from scipy.ndimage import convolve
import time
```

Simple Convolution

Definition

- **I**: Image to convolve.
- **H**: filter matrix to convolve the image with.
- **J**: Result of the convolution.

The following graphics shows exemplary the mathematical operations of the convolution. The filter matrix **H** is shifted over the input image **I**. The values 'under' the filter matrix are multiplicated with the corresponding values in **H**, summed up and writen to the result **J**. The target position is usually the position under the center of **H**.



In order to implement the convolution with a block filter, we need two methods. The first one will create the block filter matrix **H** depending on the filter width/height **n**.

A block filter holds the value $\frac{1}{n \cdot n}$ at each position:

In [ ]:
```
def block_filter(n):
    H = np.ones((n, n)) / (n * n) # each element in H has the value 1/(n*n)
```

```
    return H
```

We will test the method by creating a filter with `n = 5`:

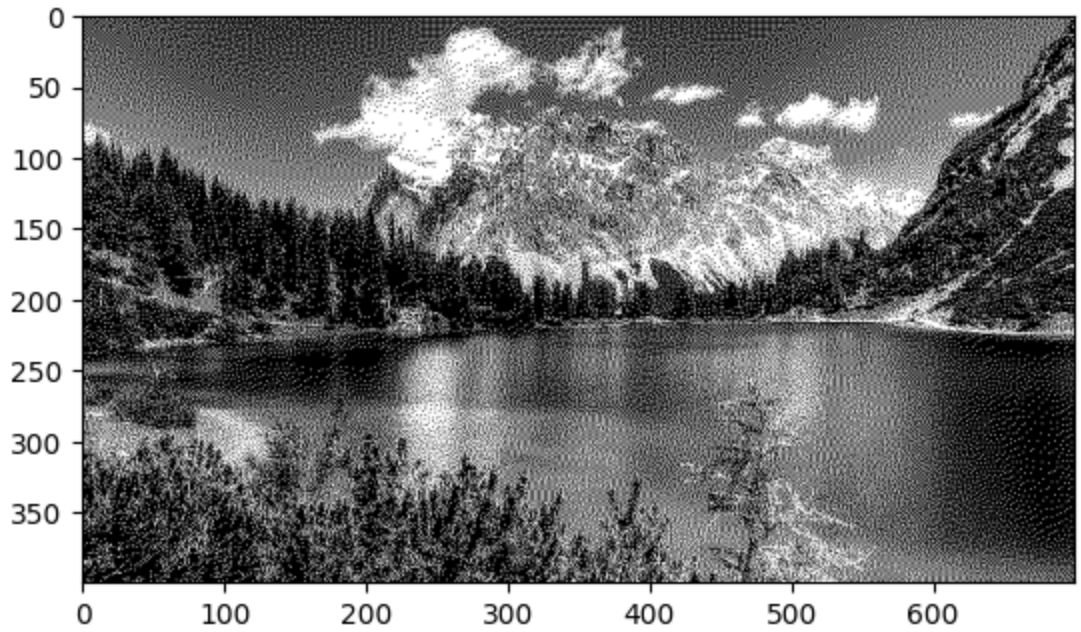
In [ ]:
```
H = block_filter(5)
print(H)
```

```
[[0.04 0.04 0.04 0.04 0.04]
 [0.04 0.04 0.04 0.04 0.04]
 [0.04 0.04 0.04 0.04 0.04]
 [0.04 0.04 0.04 0.04 0.04]
 [0.04 0.04 0.04 0.04 0.04]]
```

Next, we define the actual convolution operation. To prevent invalid indices at the border of the image, we introduce the padding **p**.

In [ ]:
```
def apply_filter(I, H):
    h, w = I.shape                          # image dimensions (height, width)
    n = H.shape[0]                          # filter size
    p = n // 2                              # padding size
    J = np.zeros_like(I)                    # output image, initialized with zeros

    for x in range(p, h-p):
        for y in range(p, w-p):
            J[x, y] = np.sum(I[x-p:x+n-p, y-p:y+n-p] * H)
    return J
```

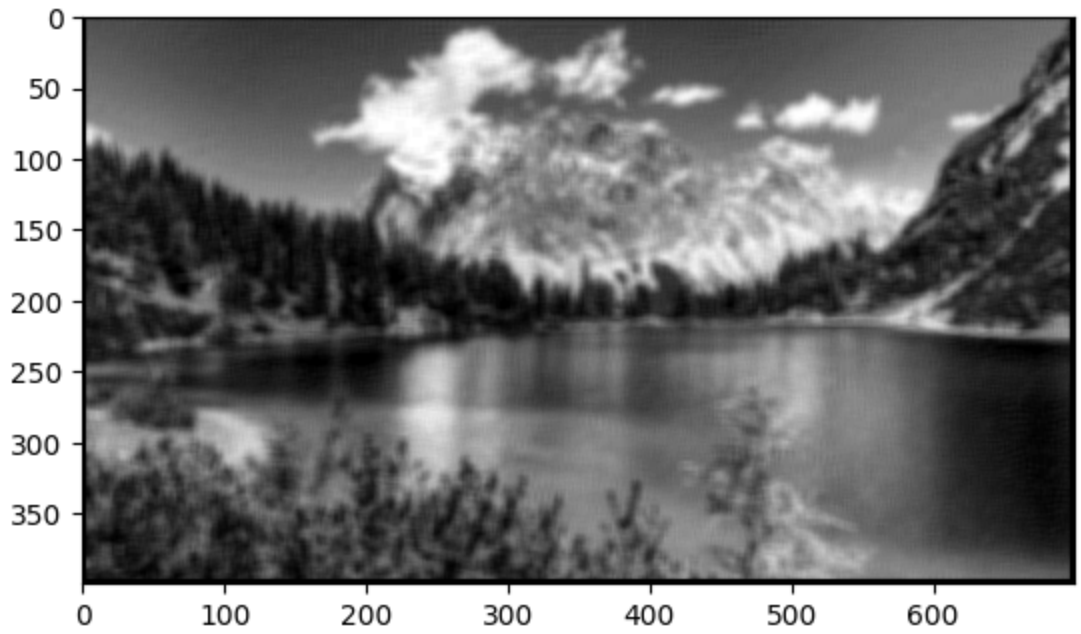
In [ ]:
```
image = Image.open('drive/MyDrive/Vision/data/image.jpg')
image = image.convert('1') # convert image to black and white

image = np.array(image)

# image = np.zeros((200, 200), dtype=np.float)
# for x in range(200):
#     for y in range(200):
#         d = ((x-100)**2+(y-100)**2)**0.5
#         image[x, y] = d % 8 < 4

plt.imshow(image, cmap='gray',vmin=0.0, vmax=1.0)
plt.show()
```



In [ ]:
```
image = image.astype(float)
```

Next we test our implementation and apply a block filter with size 7

In [ ]:
```
n = 7
H = block_filter(n)
J = apply_filter(image, H)

plt.imshow(J, cmap='gray')
plt.show()
```



PyTorch Convolution

In [ ]:
```
from PIL import Image

img = Image.open('drive/MyDrive/Vision/data/image.jpg')
img.thumbnail((256,256), Image.ANTIALIAS) # Resize to half to reduce the size of this no
```

```
<ipython-input-9-603ddaee3bb5>:4: DeprecationWarning: ANTIALIAS is deprecated and will b
e removed in Pillow 10 (2023-07-01). Use LANCZOS or Resampling.LANCZOS instead.
  img.thumbnail((256,256), Image.ANTIALIAS) # Resize to half to reduce the size of this
notebook.
```

In [ ]:
```
img
```

Out[ ]:

In [ ]:
```
import torch, torchvision
from torchvision import transforms
from torch import nn
```

In [ ]:
```
to_tensor = transforms.Compose([
   transforms.Grayscale(),   # Convert image to grayscale.
   transforms.ToTensor()     # Converts a PIL Image in the range [0, 255] to a torch.Floa
```

```
])

to_pil = transforms.Compose([
    transforms.ToPILImage()
])
```

In [ ]:
```
input = to_tensor(img)
input.shape
```

Out[ ]:
```
torch.Size([1, 146, 256])
```

In [ ]:
```
to_pil(input)
```

Out[ ]:

2D convolution over an input image:

- `in_channels = 1` : an input is a grayscale image
- `out_channels = 1` : an output is a grayscale image
- `kernel_size = (3, 3)` : the kernel (filter) size is 3 x 3
- `stride = 1` : the stride for the cross-correlation is 1
- `padding = 1` : zero-paddings on both sides for 1 point for each dimension
- `bias = False` : no bias parameter (for simplicity)

In [ ]:
```
conv = nn.Conv2d(1, 1, (3, 3), stride=1, padding=1, bias=False)
```

In [ ]:
```
# The code below does not work because the convolution layer requires the dimension for
conv(input)
```

Out[ ]:
```
tensor([[[-0.1406, -0.1579, -0.1565,  ..., -0.2149, -0.2315, -0.1015],
         [-0.0314, -0.0900, -0.0907,  ..., -0.1352, -0.0842, -0.0258],
         [-0.0351, -0.0953, -0.0911,  ..., -0.1135, -0.0299, -0.0500],
         ...,
         [ 0.0576, -0.0106, -0.0642,  ..., -0.1113, -0.1021, -0.0866],
         [ 0.0129, -0.0860, -0.0225,  ..., -0.1065, -0.1076, -0.0935],
         [-0.0468,  0.0043,  0.0271,  ..., -0.0087, -0.0102,  0.0360]]],
       grad_fn=<SqueezeBackward1>)
```

We need to insert a dimension for a batch at dim=0.

In [ ]:
```
input = input.unsqueeze(0)
input.shape
```

Out[ ]:
```
torch.Size([1, 1, 146, 256])
```

In [ ]:
```
output = conv(input)
output.shape
```

Out[ ]:
```
torch.Size([1, 1, 146, 256])
```

Setting `padding=1` in the convolution layer, we obtain an image of the same size.

In [ ]:
```
output.shape
```

Out[ ]:
```
torch.Size([1, 1, 146, 256])
```

We need to remove the first dimension before converting to a PIL object.

In [ ]:
```
output.data.squeeze(dim=0).shape
```

Out[ ]:
```
torch.Size([1, 146, 256])
```

Display the output from the convolution layer by converting `output` to a PIL object.

In [ ]:
```
to_pil(output.data.squeeze(dim=0))
```

Out[ ]:

Clip every value in the output tensor within the range of [0, 1].

In [ ]:
```
to_pil(torch.clamp(output, 0, 1).data.squeeze(dim=0))
```

Out[ ]:

In [ ]:
```
def display2(img1, img2):
    im1 = to_pil(torch.clamp(img1, 0, 1).data.squeeze(dim=0))
    im2 = to_pil(torch.clamp(img2, 0, 1).data.squeeze(dim=0))
    dst = Image.new('RGB', (im1.width + im2.width, im1.height))
    dst.paste(im1, (0, 0))
    dst.paste(im2, (im1.width, 0))
    return dst
```

In [ ]:
```
display2(input, output)
```

Out[ ]:

Identity

In [ ]:
```
conv.weight.data = torch.tensor([[[
```

```
    [0., 0., 0.],
    [0., 1, 0.],
    [0., 0., 0.],
]]])

output = conv(input)
display2(input, output)
```

Out[ ]:

Brighten

In [ ]:

```
conv.weight.data = torch.tensor([[[
    [0., 0., 0.],
    [0., 1.5, 0.],
    [0., 0., 0.],
]]])
print(conv.weight.data)
output = conv(input)
display2(input, output)
```

```
tensor([[[[0.0000, 0.0000, 0.0000],
          [0.0000, 1.5000, 0.0000],
          [0.0000, 0.0000, 0.0000]]]])
```

Out[ ]:

Darken

In [ ]:

```
conv.weight.data = torch.tensor([[[
    [0., 0., 0.],
    [0., 0.5, 0.],
    [0., 0., 0.],
]]])
print(conv.weight.data)
output = conv(input)
display2(input, output)
```

```
tensor([[[[0.0000, 0.0000, 0.0000],
          [0.0000, 0.5000, 0.0000],
          [0.0000, 0.0000, 0.0000]]]])
```

Out[ ]:

Box blur

In [ ]:
```
conv.weight.data = torch.ones((1, 1, 3,3), dtype=torch.float) / 9.
print(conv.weight.data)
output = conv(input)
display2(input, output)
```

```
tensor([[[[0.1111, 0.1111, 0.1111],
          [0.1111, 0.1111, 0.1111],
          [0.1111, 0.1111, 0.1111]]]])
```

Out[ ]:

Gaussian blur

In [ ]:
```
conv.weight.data = torch.tensor([[[
    [1., 2., 1.],
    [2., 4., 2.],
    [1., 2., 1.],
]]])/16.
print(conv.weight.data)
output = conv(input)
display2(input, output)
```

```
tensor([[[[0.0625, 0.1250, 0.0625],
          [0.1250, 0.2500, 0.1250],
          [0.0625, 0.1250, 0.0625]]]])
```

Out[ ]:

Sharpen

In [ ]:
```
conv.weight.data = torch.tensor([[[
    [0., -1., 0.],
    [-1., 5., -1.],
    [0., -1., 0.],
]]])
print(conv.weight.data)
```

```
output = conv(input)
display2(input, output)
```

```
tensor([[[[ 0., -1.,  0.],
          [-1.,  5., -1.],
          [ 0., -1.,  0.]]]])
```

Out[ ]:

In [ ]:

```
conv.weight.data = torch.tensor([[[
    [0., -2., 0.],
    [-2., 10., -2.],
    [0., -2., 0.],
]]])
print(conv.weight.data)
output = conv(input)
display2(input, output)
```

```
tensor([[[[ 0., -2.,  0.],
          [-2., 10., -2.],
          [ 0., -2.,  0.]]]])
```

Out[ ]:

Edge detection

In [ ]:

```
conv.weight.data = torch.tensor([[[
    [0., 1., 0.],
    [1., -4., 1.],
    [0., 1., 0.],
]]])
print(conv.weight.data)
output = conv(input)
display2(input, output)
```

```
tensor([[[[ 0.,  1.,  0.],
          [ 1., -4.,  1.],
          [ 0.,  1.,  0.]]]])
```

Out[ ]:

In [ ]:

```
conv.weight.data = torch.tensor([[[
    [-1., -1., -1.],
```

```
    [-1., 8., -1.],
    [-1., -1., -1.],
]]])
output = conv(input)
display2(input, output)
```

Out[ ]:

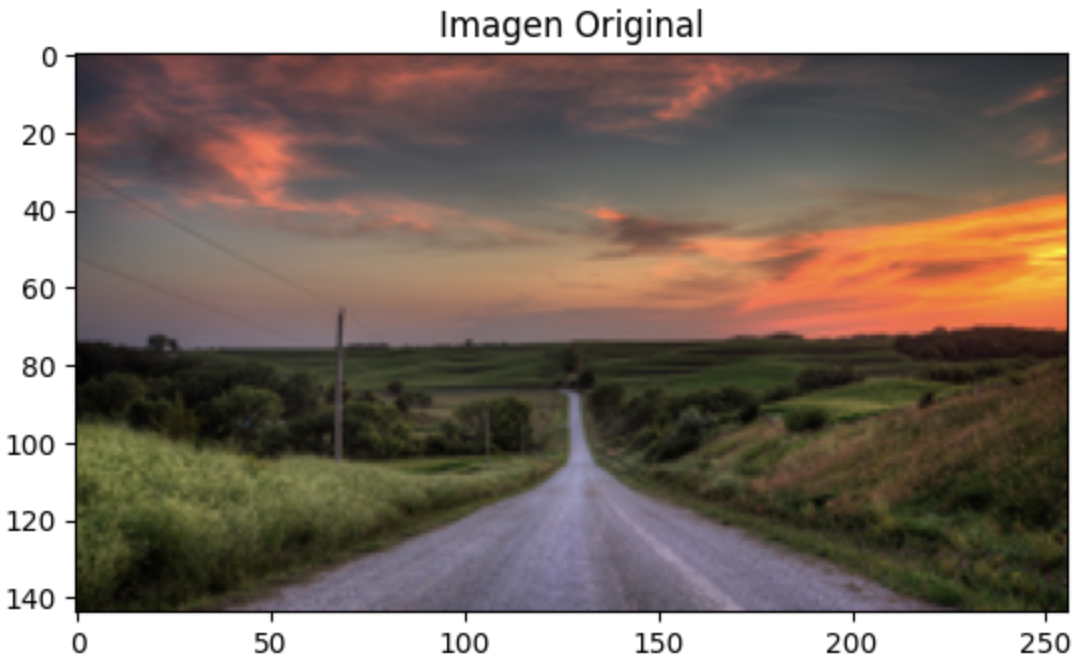
2. Ejercicios

In [ ]:
```
img = Image.open('drive/MyDrive/Vision/4_convolution/data/image2.jpeg')
img.thumbnail((256, 256), Image.ANTIALIAS)
img_tensor = torch.tensor(np.array(img) / 255.0, dtype=torch.float32).permute(2, 0, 1)
```

```
<ipython-input-34-8c67cd84b00c>:2: DeprecationWarning: ANTIALIAS is deprecated and will be removed in Pillow 10 (2023-07-01). Use LANCZOS or Resampling.LANCZOS instead.
  img.thumbnail((256, 256), Image.ANTIALIAS)
```

Comenzamos cargando nueva imagen, cambiando su tamaño con la función de thumbnail(), y convirtiéndola a un tensor para trabajarla con convolucionales.

In [ ]:
```
plt.imshow(img)
plt.title('Imagen Original')
plt.show()
```



Ejercicio 1

Implementa los detectores de línea siguientes usando código en Python (es decir, sin usar librerías de

OpenCV): Prewitt, Sobel y Laplaciano. Investiga la complejidad algoritmica de estos, ¿cual es más eficiente?

Comenzamos cargando nueva imagen, cambiando su tamaño con la función de thumbnail().

In [ ]:
```
img = Image.open('drive/MyDrive/Vision/4_convolution/data/image2.jpeg')
img.thumbnail((512, 512), Image.ANTIALIAS)
```

```
<ipython-input-36-c10562dd2eb8>:2: DeprecationWarning: ANTIALIAS is deprecated and will
be removed in Pillow 10 (2023-07-01). Use LANCZOS or Resampling.LANCZOS instead.
  img.thumbnail((512, 512), Image.ANTIALIAS)
```

Haciendo uso de la lógica aprendida al inicio del Notebook utilizaremos tensores para realizar el ejercicio. Generamos método para cambiar a escala de grises, así como convertir a tensor para trabajarla. De igual manera, to_pil() para convertir el tensor en imagen.

In [ ]:
```
to_tensor = transforms.Compose([
   transforms.Grayscale(),  # Convert image to grayscale.
   transforms.ToTensor()    # Converts a PIL Image in the range [0, 255] to a torch.Floa
])

to_pil = transforms.Compose([
    transforms.ToPILImage()
])
```

Observamos el tensor:

In [ ]:
```
input = to_tensor(img)
input.shape
```

Out[ ]:
```
torch.Size([1, 288, 512])
```

Y el tensor en imagen:

In [ ]:
```
to_pil(input)
```

Out[ ]:

Importamos display de IPython para poder desplegar las imágenes dentro de nuestro próximo for loop:

In [ ]:
```
from IPython.display import display
```

Ocupamos implementar los detectores de línea Prewitt, Sobel, y Laplacian, de acuerdo a las instrucciones del ejercicio. Por lo que generamos los kernels con sus respectivos valores concorde a sus métodos. Los

almacenamos en una lista, y los aplicamos dentro de un for loop.

Prewit

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \quad G_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

Sobel

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Laplaciano

0	−1	0
−1	4	−1
0	−1	0

−1	−1	−1
−1	8	−1
−1	−1	−1

In [ ]:
```
# definimos valores de acuerdo a los métodos
tensor_prewittX = [[1,0,-1],[1,0,-1],[1,0,-1]]
tensor_prewittY = [[1,1,1],[0,0,0],[-1,-1,-1]]
tensor_sobelX = [[-1, 0, 1],[-2, 0, 2],[-1, 0, 1]]
tensor_sobelY = [[-1, -2, -1],[0, 0, 0],[1, 2, 1]]
tensor_laplaciano = [[0, 1, 0],[1, -4, 1],[0, 1, 0]]

# generamos listas con lo previo
filters_list = [tensor_prewittX,tensor_prewittY,tensor_sobelX,tensor_sobelY,tensor_lapla
filters_titles = ['PrewittX','PrewittY','SobelX','SobelY',"Laplaciano"]

images_filtered = []  # lista para almacenar las imagenes despues de los métodos
for i in range(len(filters_titles)):
  start = time.time()     # We store the time to calculate the total processing time
  conv.weight.data = torch.tensor([[filters_list[i]]], dtype=torch.float32)  # asignamos
  end = time.time()      # Now, we store the time when it finishes processing
  images_filtered.append(conv(input))    # aplicamos métodos
  print(f'{filters_titles[i]}: Total processing time: {(end - start):.4f} seconds') # An
  display(display2(input,images_filtered[i]))
```

```
PrewittX: Total processing time: 0.0014 seconds
```

PrewittY: Total processing time: 0.0002 seconds

SobelX: Total processing time: 0.0001 seconds

SobelY: Total processing time: 0.0002 seconds

Laplaciano: Total processing time: 0.0001 seconds

Los métodos de Prewitt y Sobel tienen sus respectivos valores para X y Y. Por lo que es necesario juntarlos para obtener sus respectivas imágenes:

Prewitt

In [ ]:
```
display2(input,images_filtered[0] + images_filtered[1])
```

Out[ ]:

Sobel

In [ ]:
```
display2(input,images_filtered[2] + images_filtered[3])
```

Out[ ]:

Laplaciano

In [ ]:
```
display2(input,images_filtered[4])
```

Out[ ]:

Conclusión 1: Tanto el filtro de Sobel o Prewit funcionan mejor en términos de calidad para definir líneas, al realizar un ajuste tanto vertical como Horizontal sobre la imagen. Sin embargo, el filtro laplaciano es más eficiente computacionalmente al sólo ser una operación matricial, en lugar de 2 (vertical y horizontal) como se realiza en los otros filtros.

Ejercicio 2

Implementa un algoritmos de realce o mejoramiento de imágenes mediante un algoritmo de en el cual se extraen las líneas de la imagen y después se aplica la diferencia con la imagen original, multiplicando los píxeles de la imagen "máscara" (las líneas encontradas) por un factor alfa mayor a 1.

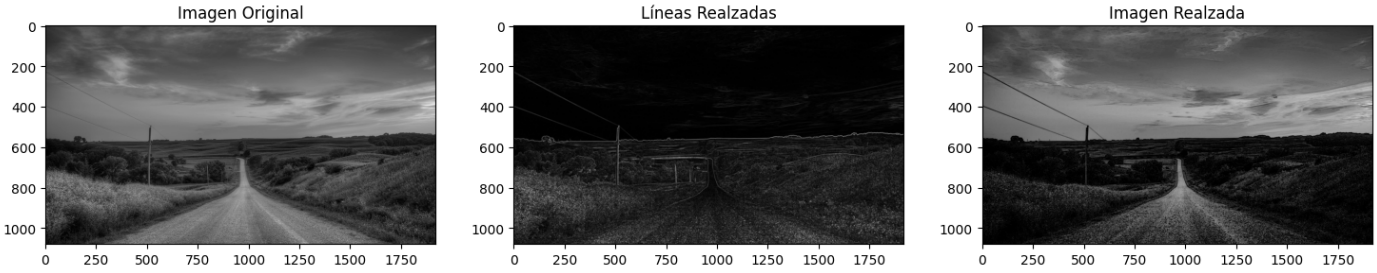
Generamos una función para realzar la imagen mediante líneas. Primero aplicamos un blur para suavizar la imagen y que nuestro método de detección de líneas no encuentre tanto ruido. Seguido, utilizamos Sobel para detección de imágenes. Calculamos la magnitud del gradiente y la normalizamos entre 0 y 255. Convertimos a integer, y multiplicamos la magnitud por nuestro factor alfa (mayor a 1). Finalmente, extraemos las líneas realzadas de la imagen original.

In [ ]:
```
def line_enchance(imagen_path, alfa):
    imagen = cv2.imread(imagen_path, cv2.IMREAD_GRAYSCALE)

    # Aplicar un filtro de media para suavizar la imagen
    imagen_suavizada = cv2.blur(imagen, (5, 5))

    # Aplicar el filtro Sobel para detectar bordes en X y Y
    sobel_x = cv2.Sobel(imagen, cv2.CV_64F, 1, 0, ksize=5)
    sobel_y = cv2.Sobel(imagen, cv2.CV_64F, 0, 1, ksize=5)

    # Calcular la magnitud del gradiente
    magnitud = np.sqrt(sobel_x**2 + sobel_y**2)

    # Normalizar la magnitud del gradiente a valores entre 0 y 255
    magnitud = cv2.normalize(magnitud, None, 0, 255, cv2.NORM_MINMAX)

    # Convertir la magnitud a tipo de datos uint8
    magnitud = np.uint8(magnitud)

    # Multiplicar los píxeles de la imagen mask por alfa
    lineas_realzadas = alfa * magnitud.astype(np.float32)

    # Restar las líneas realzadas de la imagen original
    imagen_realzada = (imagen.astype(np.float32) - lineas_realzadas).clip(0, 255).astype

    plt.figure(figsize=(18, 10))
    plt.subplot(131), plt.imshow(imagen, cmap='gray'), plt.title('Imagen Original')
    plt.subplot(132), plt.imshow(lineas_realzadas, cmap='gray'), plt.title('Líneas Realz
```

```
    plt.subplot(133), plt.imshow(imagen_realzada, cmap='gray'), plt.title('Imagen Realza
    plt.show()
```

Aplicamos nuestra función, y desplegamos las imágenes:

In [ ]:
```
ruta_imagen = 'drive/MyDrive/Vision/4_convolution/data/image2.jpeg'
alfa = 1.5

line_enchance(ruta_imagen, alfa)
```



Observamos la imagen original, las líneas, y el resultado a un alfa de 1.5.

Ejercicio 3

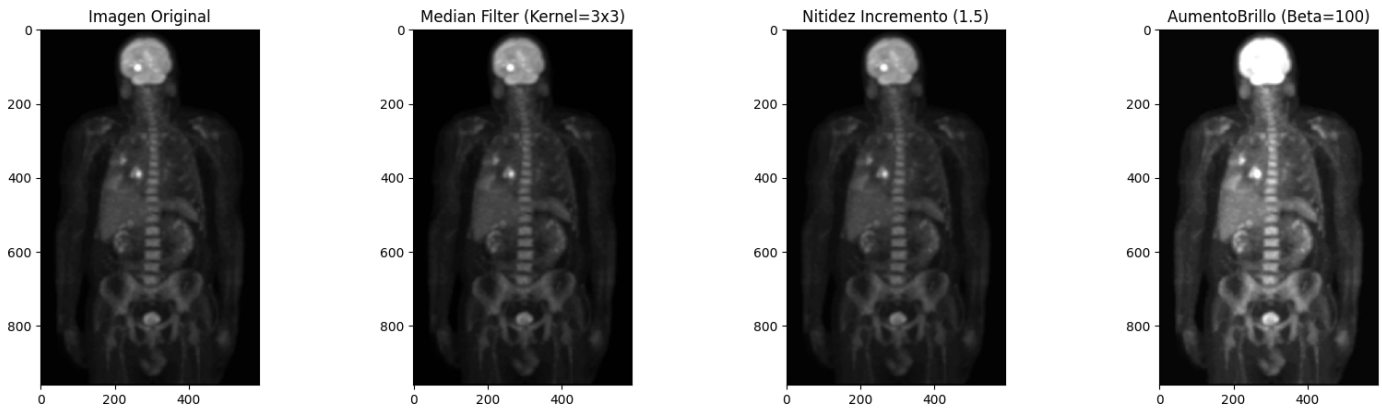
Buscar una aplicación médica (en el libro de Gonzalez viene varios ejemplos de imágenes PET) y hacer mejoramiento usando la técnica de de la Figura 3.43 combinando diferentes etapas de procesamiento de imagenes

Generamos una función para aplicar nuestro mejoramiento de imagen. La figura 3.43 del libro muestra reducción de ruido mediante un filtro de paso bajo Gaussiano con s = 3. Así como reducción de ruido usando un median filter de 7x7. Nosotros comenzamos aplicando un median filter, seguido de un aumento de sharpness de la imagen, y finalmente incremento de brillo.

In [ ]:
```
def mejoramiento_imagen(image_path, median_kernel_size=7, sharpening_strength=1.5, brigh
    original_image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

    # Reducimos ruido con un median filter
    median_filtered_image = cv2.medianBlur(original_image, median_kernel_size)

    # Mejoramos nitidez
    sharpened_image = cv2.addWeighted(median_filtered_image, 1.0 + sharpening_strength,

    # Aumentamos brillo
    brightness_image = cv2.convertScaleAbs(sharpened_image, alpha=1, beta=brightness)

    # Desplegamos imagenes
    plt.figure(figsize=(20, 5))
    plt.subplot(1, 4, 1), plt.imshow(original_image, cmap='gray'), plt.title('Imagen Ori
    plt.subplot(1, 4, 2), plt.imshow(median_filtered_image, cmap='gray'), plt.title(f'Me
    plt.subplot(1, 4, 3), plt.imshow(sharpened_image, cmap='gray'), plt.title(f'Nitidez
    plt.subplot(1, 4, 4), plt.imshow(brightness_image, cmap='gray'), plt.title(f'Aumento

    plt.show()
```

Aplicamos nuestra función, y desplegamos las imágenes:

In [ ]:
```
image_path = 'drive/MyDrive/Vision/4_convolution/data/image3.jpg'

mejoramiento_imagen(image_path, median_kernel_size=3, sharpening_strength=1.5, brightnes
```



Se procuro intentar una combinación de métodos diferentes para mejorar la visualización de la imagen original. La combinación de median filter (3x3), sharpness y brightness increase, fue la que encontramos que mejor nos permitió observar la imagen original.

3. Referencias

- Bradski, G. (2000). The OpenCV Library. Dr. Dobb's Journal of Software Tools.
- Clark, A. (2015). Pillow (PIL Fork) Documentation. readthedocs. Retrieved from https://buildmedia.readthedocs.org/media/pdf/pillow/latest/pillow.pdf
- Gonzalez, R., & Woods, R. (2018). Digital Image Processing. Pearson.
- Tsankashvili, N. (2018). Comparing Edge Detection Methods. Retrieved from Medium: https://medium.com/@nikatsanka/comparing-edge-detection-methods-638a2919476e