Convolutional Neural Networks – Lab 2

RUBEN MARTINEZ GONZALEZ

February 8, 2024

1 Introducción

En este informe, se presenta el trabajo realizado en el laboratorio 2 de Redes Neuronales Convolucionales. Se implementaron varias funciones relacionadas con el procesamiento de imágenes, incluyendo la aplicación de filtros Sobel, Laplacianos y detección de bordes. La solución se implementó en Python utilizando la biblioteca NumPy para el procesamiento de matrices y la biblioteca Matplotlib para la visualización de imágenes. La implementación está desplegada en un notebook de Google Colab, el cual se puede acceder a través del siguiente enlace: `Colab`

2 Funciones Implementadas

2.1 Función sobelFiltering

Esta función convoluciona una imagen con un filtro Sobel, que permite calcular la derivada de la intensidad de la imagen, pixel a pixel, en una dirección específica. La función toma como parámetros la matriz que almacena los datos de la imagen y el filtro Sobel vertical u horizontal a utilizar.

```python
def sobelFiltering(image, sobel_filter):

    kernel = np.flip(sobel_filter)
    height, width = image.shape
    filter_size = kernel.shape[0]
    filter_radius = filter_size // 2

    # Padded version of the image with zero-padding
    padded_image = np.pad(image, ((filter_radius, filter_radius), (filter_radius, filter_radius)), mode='constant')

    # Initialize array to store the filtered image
    filtered_image = np.zeros_like(image)

    # Iterate over each pixel in the image
    for i in range(height):
        for j in range(width):
            # Extract the neighborhood of the pixel from the padded image
            neighborhood = padded_image[i:i+filter_size, j:j+filter_size]
            # Apply the Sobel filters
            filtered_image[i, j] = np.sum(neighborhood * kernel)

    return filtered_image

```

Listing 1: Implementación sobelfiltering

La implementación de esta función sigue la lógica descrita a continuación:

- Se inicializa el kernel invirtiendo el filtro Sobel que se recibe por parámetro y se obtienen las dimensiones de la imagen.
- Se crea una imagen con 'zero-padding' para poder aplicar el filtro en los bordes de la imagen.
- Se inicializa la imagen filtrada con ceros y las mismas dimensiones que la imagen original.
- Se itera sobre cada píxel de la imagen con padding y se extrae un vecindario de píxeles correspondiente.
- Se calcula un nuevo valor para cada pixel de la imagen filtrada aplicando el filtro Sobel al vecindario.
- Se retorna la imagen filtrada.

Todas las pruebas descritas en este documente se realizaron con la version a escalas de grises de la imagen a continuación. `Link de descarga:`

Figure 1: imagen_original

Se probó la función sobelFiltering con los siguientes filtros Sobel:

- Filtro Sobel en dirección 'X'

```
    Mx = np.array([
    [-1, 0, 1],
    [-2, 0, 2],
    [-1, 0, 1]
    ], dtype=float)

```

Listing 2: Filtro Sobel vertical

- Filtro Sobel en dirección 'Y'

```
    Mx = np.array([
    [-1, -2, -1],
    [ 0,  0,  0],
    [ 1,  2,  1]
    ], dtype=float)

```

Listing 3: Filtros Sobel horizontal

Para ejecutar la prueba solo fue necesario llamar a la función sobelFiltering con la imagen y el filtro correspondiente. A modo complementario se desarrolló la función compare_images para mostrar juntas la imagen original, la imagen filtrada y las diferencias entre ambas (resta de píxeles).

```
sobel_filtered_vertical_image = sobelFiltering(imagen_original, Mx)
compare_images(imagen_original,'Imagen Original', sobel_filtered_vertical_image, '
    Filtro de Sobel vertical')

```

Listing 4: Ejecutando sobelFiltering

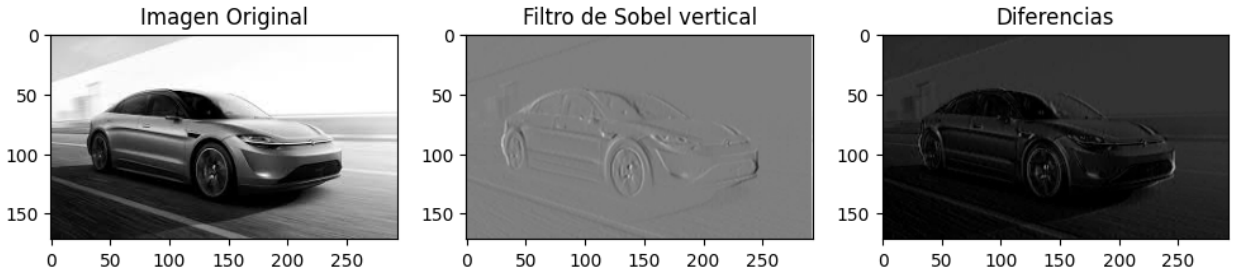

Figure 2: Resultado de aplicar el filtro Sobel en dirección 'X' a la imagen original.

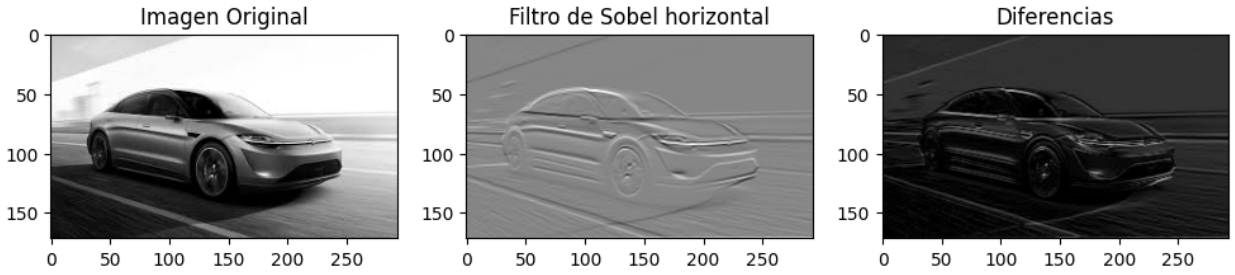

Figure 3: Resultado de aplicar el filtro Sobel en dirección 'Y' a la imagen original.

2.2 Función laplacianFiltering

Esta función convoluciona una imagen con un filtro Laplaciano, que se utiliza para detectar cambios abruptos en la intensidad de los píxeles de una imagen. La función toma como parámetro la matriz que almacena los datos de la imagen.

```
    def laplacianFiltering(image):

    laplacian_filter = np.array([
              [0,  1,  0],
              [1, -4,  1],
              [0,  1,  0]
             ], dtype=float)

    height, width = image.shape
    filter_size = laplacian_filter.shape[0]
    filter_radius = filter_size // 2

    # Padded version of the image with zero-padding
    padded_image = np.pad(image, ((filter_radius, filter_radius), (filter_radius,
    filter_radius)), mode='constant')

    # Initialize array to store the filtered image
    filtered_image = np.zeros_like(image)

    # Iterate over each pixel in the image
    for i in range(height):
        for j in range(width):
            # Extract the neighborhood of the pixel from the padded image
            neighborhood = padded_image[i:i+filter_size, j:j+filter_size]
            # Apply the Laplacian filter
            filtered_image[i, j] = np.sum(neighborhood * laplacian_filter)

    return filtered_image

```

Listing 5: Implementación laplacianFiltering

La implementación de esta función sigue la prácticamente la misma lógica descrita anteriormente para la función sobelFiltering, excepto que en este caso se utiliza un filtro Laplaciano para aplicar la convolución. Se probó la función laplacianFiltering con la imagen utilizada anteriormente y se muestran los resultados a continuación.

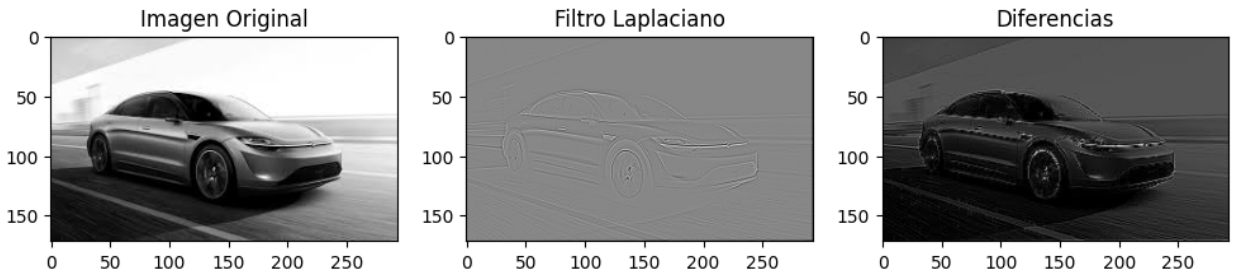

Figure 4: Resultado de aplicar el filtro Laplaciano a la imagen original.

2.3 Función edgeDetector

Esta función aplica el kernel de gradiente en dirección 'X', el kernel de gradiente en dirección 'Y', calcula la magnitud del gradiente y realiza una umbralización de la magnitud del gradiente. La función toma tres parámetros: la matriz que almacena los datos de la imagen, un número que indica el tipo de kernel de gradiente a utilizar (se probaron dos tipos diferentes Sobel y Prewitt) y el valor de umbral. La función muestra la derivada en dirección 'X', la derivada en dirección 'Y' y los bordes umbralizados. Se implementó también un extra para mostrar en la imagen resultante formas de flecha representando la magnitud y dirección de los bordes umbralizados.

La implementación de esta función se mostrará por partes para su mejor comprensión.

- Primero se aplica un suavizado a la imagen utilizando un filtro Gaussiano para reducir ruido y detalles innecesarios. (se reutiliza la función de suavizado averageFiltering implementada en el laboratorio 1)

```
# 1. Suavizado
gaussian_filter_matrix = np.array([
[1, 2, 1],
[2, 4, 2],
[1, 2, 1]
], dtype=float)
gaussian_filter_matrix /= np.sum(gaussian_filter_matrix)
smoothed_image = averageFiltering(image, gaussian_filter_matrix)

```

Listing 6: Implementación edgeDetector - Suavisado

- Luego se aplica el filtro de gradiente correspondiente (Sobel o Prewitt) a la imagen suavizada.

```
# 2. Realce
if type_gradient == 1:
    gradient_magnitude_image, filtered_vertical_image,
    filtered_horizontal_image = get_sobel_gradient_magnitude(smoothed_image)
elif type_gradient == 2:
    gradient_magnitude_image, filtered_vertical_image,
    filtered_horizontal_image = get_prewitt_gradient_magnitude(smoothed_image)
else:
    raise ValueError("Invalid type_gradient value. Choose 1 for Sobel or 2 for
     Prewitt.")

```

Listing 7: Implementación edgeDetector - Gradiente

  - Se definieron complementariamente las funciones 'get_sobel_gradient_magnitude' y 'get_prewitt_gradient_magnitude' para obtener la magnitud del gradiente.

```
def get_sobel_gradient_magnitude(image):
    Mx = np.array([
          [-1, 0, 1],
          [-2, 0, 2],
          [-1, 0, 1]
      ], dtype=float)

    My = np.array([
          [-1, -2, -1],
          [ 0,  0,  0],
          [ 1,  2,  1]
      ], dtype=float)

    sobel_filtered_vertical_image = sobelFiltering(imagen_original, Mx)
    sobel_filtered_horizontal_image = sobelFiltering(imagen_original, My)
```

```
    #                Magnitud =                                    (df/dx)^2
    + (df/dy)^2
    gradient_magnitude_image = np.sqrt(sobel_filtered_vertical_image**2 +
    sobel_filtered_horizontal_image**2)

    return gradient_magnitude_image, sobel_filtered_vertical_image,
    sobel_filtered_horizontal_image

```

Listing 8: Implementación get_sobel_gradient_magnitude

- Estas funciones aplican los filtros Sobel o Prewitt en dirección 'X' y 'Y' a la imagen original, calculan la magnitud del gradiente utilizando la fórmula $\sqrt{(df/dx)^2 + (df/dy)^2}$
- Se retorna la magnitud del gradiente y las derivadas en dirección 'X'(sobel_filtered_vertical_image) y 'Y' (sobel_filtered_horizontal_image).
- La función get_prewitt_gradient_magnitude es similar a la anterior, pero utiliza los filtros Prewitt en lugar de los filtros Sobel.

```
    Mx = np.array([
          [-1, 0, 1],
          [-1, 0, 1],
          [-1, 0, 1]
      ], dtype=float)

    My = np.array([
          [-1, -1, -1],
          [ 0,  0,  0],
          [ 1,  1,  1]
      ], dtype=float)

```

Listing 9: Filtros prewitt

- A continuación se realiza una umbralización de la magnitud del gradiente estableciendo valor 255 a los píxeles cuya magnitud sea mayor que el umbral y 0 en caso contrario. La representación resultante se almacena en la variable thresholded_edges.

```
# 3. Umbralizaci n
thresholded_edges = np.where(gradient_magnitude_image > threshold, 255, 0)

```

Listing 10: Implementación edgeDetector - Umbralización

- A continuación se dibujan flechas en la imagen resultante para representar la magnitud y dirección de los bordes umbralizados. Para ello realizaron los siguientes calculos adicionales:
  - se obtuvieron puntos aleatorios en la zona central de la imagen. La cantidad de puntos se estableció en num_arrows = 10. Para ello se Implementó la función get_edge_points.

```
     # 4. Dibujar flechas en los bordes umbralizados
    edge_points = get_edge_points(thresholded_edges, num_arrows = 10)

```

Listing 11: Implementación edgeDetector - edge_points

  - La función get_edge_points recibe como parámetro la imagen umbralizada y la cantidad de puntos a seleccionar. Se encarga de obtener los puntos en los que la imagen umbralizada tiene valor 255 y seleccionar solo los puntos en la zona central de la imagen. Se acotó la zona central de la imagen con un margen del 25% desde el borde de la imagen por conveniencia para disminuir la posibilidad de que al pintar las flechas estas se salgan de la imagen.

```
def get_edge_points(thresholded_edges, num_arrows = 10):
    edge_points = np.argwhere(thresholded_edges == 255)
     # Obtener l mites de la zona central
    h, w = thresholded_edges.shape
    margin = int(min(h, w) * 0.25)  # Margen del 25% desde el borde de la
    imagen
    min_x = margin
    max_x = w - margin
    min_y = margin
    max_y = h - margin

    # Seleccionar puntos aleatorios de la zona central
    edge_points = [point for point in edge_points if min_x < point[1] <
    max_x and min_y < point[0] < max_y]
    np.random.shuffle(edge_points)
    edge_points = edge_points[:num_arrows]  # Tomar solo algunos puntos
    para las flechas

    return edge_points

```

Listing 12: Implementación get_edge_points

– Se obtuvo la dirección del gradiente en cada punto de la imagen umbralizada utilizando la función np.arctan2. La función np.arctan2 recibe como parámetros las derivadas en dirección 'X' y 'Y' y retorna el ángulo correspondiente. Los ángulos se almacenaron en la variable gradient_direction_image.

```
gradient_direction_image = np.arctan2(filtered_horizontal_image,
    filtered_vertical_image)

```

Listing 13: Implementación edgeDetector - gradient_direction

–

– Finalmente se dibujaron las flechas en la imagen umbralizada recorriendo los puntos obtenidos aleatoriamente en el paso anterior, calculando las coordenadas finales de la flecha y utilizando la función plt.arrow de la biblioteca Matplotlib. Las coordenadas finales de la flecha se calcularon con las fórmulas:

  * dx = magnitud * np.cos(dirección)
  * dy = magnitud * np.sin(dirección)

  La magnitud se dividió por 10 a conveniencia para que las flechas se escalen adecuadamente en la imagen.

```
  for point in edge_points:
    y, x = point
    magnitude = gradient_magnitude_image[y, x]/10
    direction = gradient_direction_image[y, x]

    # Calcular las coordenadas finales de la flecha
    dx = magnitude * np.cos(direction)
    dy = magnitude * np.sin(direction)

    plt.arrow(x, y, dx, dy, color='red', head_width=4, head_length=8)

```

Listing 14: Implementación edgeDetector - Dibujar flechas

- Finalmente se muestran los resultados obtenidos en 4 subfiguras que contienen:
  - La derivada en dirección 'X'
  - La derivada en dirección 'Y'
  - Los bordes umbralizados
  - Las flechas que representan la magnitud y dirección de los bordes umbralizados.

```
plt.subplot(2, 2, 1)
plt.imshow(filtered_vertical_image, cmap='gray')
plt.title('Derivative in x direction')

plt.subplot(2, 2, 2)
plt.imshow(filtered_horizontal_image, cmap='gray')
plt.title('Derivative in y direction')


plt.subplot(2, 2, 3)
plt.imshow(thresholded_edges, cmap='gray')
plt.title(f'Umbralizaci n (Threshold = {threshold})')

plt.subplot(2, 2, 4)
plt.imshow(thresholded_edges, cmap='gray')
plt.title(f'Arrow shapes magnitude and directions')


```

Listing 15: Implementación edgeDetector - Mostrar resultados

Se probó la función edgeDetector con la imagen utilizada anteriormente y se muestran los resultados a continuación.

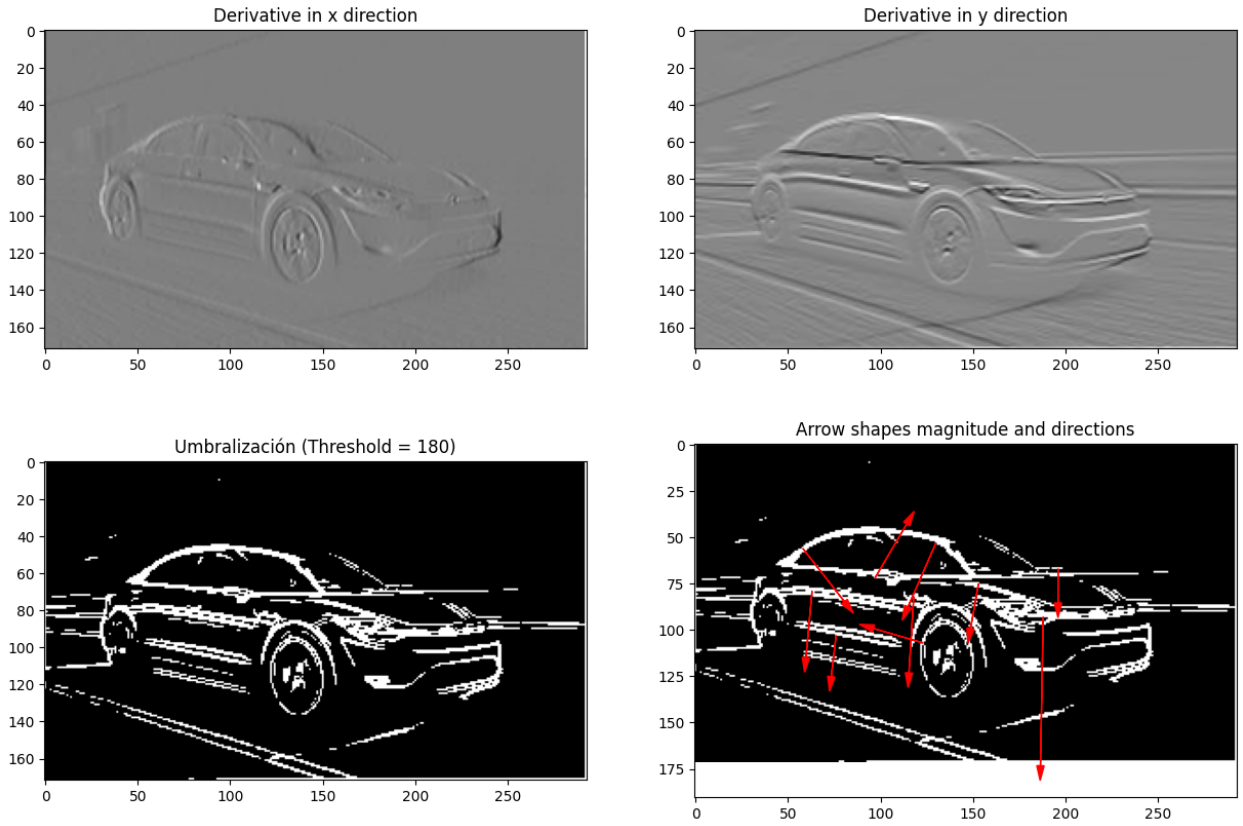

Figure 5: Resultado de aplicar el detector de bordes con filtro Sobel y umbral 180.

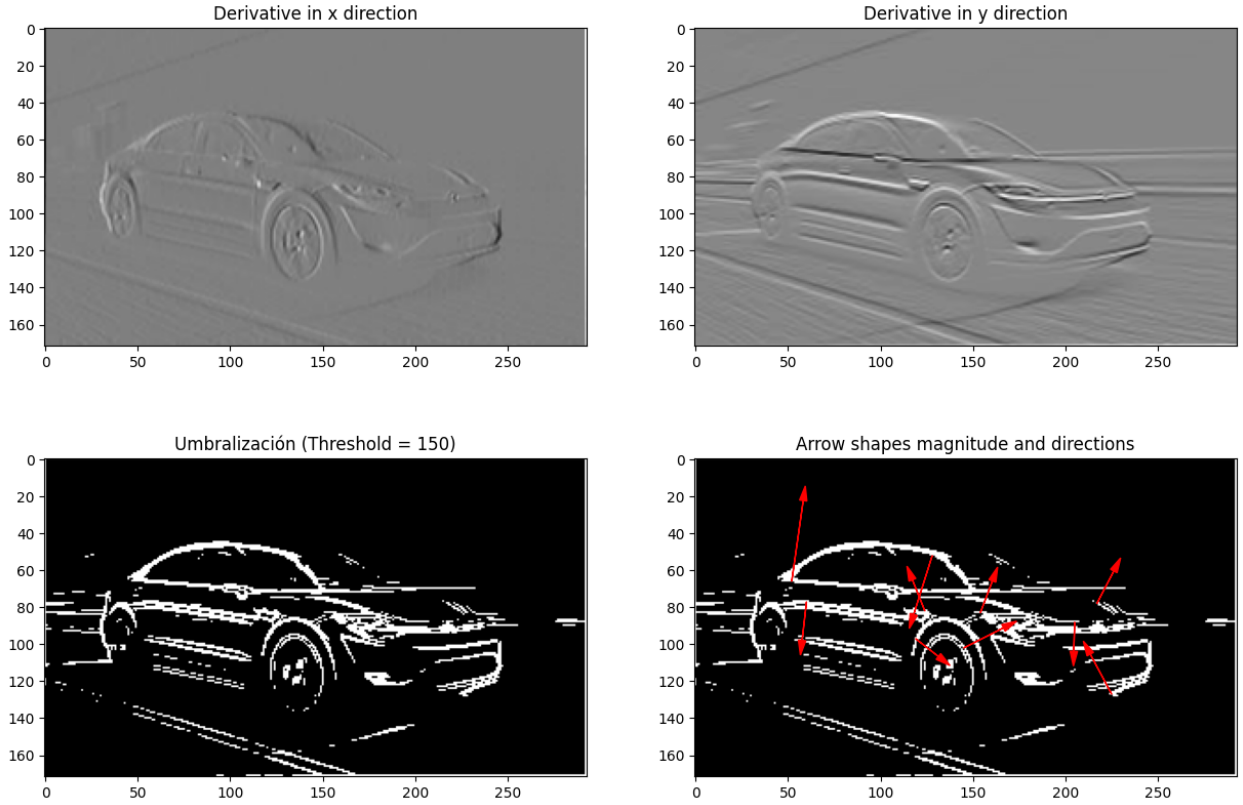

Figure 6: Resultado de aplicar el detector de bordes con filtro Prewitt y umbral 180.

3 Conclusiones

En esta práctica se implementaron varias funciones para el procesamiento de imágenes, incluyendo filtros Sobel y Laplacianos, así como la detección de bordes utilizando el kernel de gradiente en dirección 'X' y 'Y'. Se logró aplicar estos filtros y detectores de bordes a una imagen a escala de grises y se obtuvieron resultados satisfactorios. Estas técnicas son fundamentales en el campo de la visión por computadora y son ampliamente utilizadas en aplicaciones de reconocimiento de objetos y segmentación de imágenes.