Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ingeniería Eléctrica

Tarea 3

Segmentación de imágenes

Alumno: Lucas Orellana Jara
Profesor: Claudio Pérez F.
Auxiliar: Diego Maureira
Ayudantes: Juan Pablo Pérez
Jhon Pilataxi
Jorge Zambrano

Fecha de entrega: 25 de octubre de 2024
Santiago de Chile

Índice de Contenidos

1. Mejora de algoritmo *handcrafted* de segmentación de grietas **1**

2. Entrenamiento de una CNN **6**

3. Comparación de rendimiento **10**

Anexo A. Mejora de algoritmo *handcrafted* de segmentación de grietas **12**

Índice de Figuras

1. Ejemplo de segmentación para una grieta cualquiera. 1
2. Segmentación de grieta usando parámetros que maximizan *IoU*. 4
3. Segmentación de grieta usando parámetros que maximizan *Accuracy*. 4
4. Segmentación de grieta usando parámetros que maximizan *IoU* y *Accuracy*. . . 4
5. Segmentación de grieta usando parámetros que maximizan *IoU*. 5
6. Segmentación de grieta usando parámetros que maximizan *Accuracy*. 5
7. Segmentación de grieta usando parámetros que maximizan *IoU* y *Accuracy*. . . 5
8. Salida de la consola al ingresar el comando `nvidia-smi` 6
9. Evolución *Accuracy* e *IoU* a lo largo de las épocas. 7
10. Evolución del *learning_rate* y *loss* a lo largo de las épocas. 7
11. Resultado de segmentación usando *CNN* 8
12. Segmentación de grieta ejemplo usando parámetros que maximizan *IoU*. 10
13. Segmentación de grieta ejemplo usando parámetros que maximizan *Accuracy*. . 11
14. Segmentación de grieta ejemplo usando parámetros que maximizan *IoU* y *Accuracy*. 11

Índice de Tablas

1. Métricas para todo el dataset aplicada esta segmentación. 1
2. Parámetros a optimizar para segmentación. 3
3. Resultados con parámetros optimizados. 3
4. Evolución de *Acuracy* e *IoU* para fondo y grieta. 8
5. Comparación de métricas para todos los enfoques usados. 10

Índice de Códigos

A.1. Algoritmo de segmentación de grieta preliminar. 12
A.2. Algoritmo de segmentación posterior optimización. 12

1. Mejora de algoritmo *handcrafted* de segmentación de grietas

Se parte desde la base que se tenía hecha en la tarea anterior (ver Código A.1) y es en base a este mismo que se obtienen los siguientes resultados:

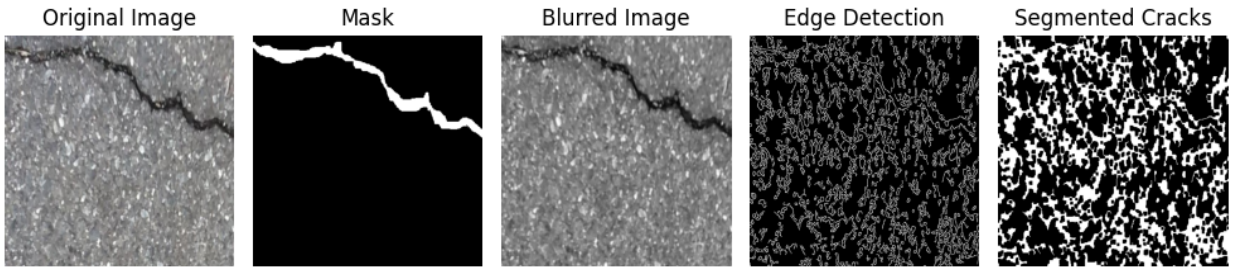

Figura 1: Ejemplo de segmentación para una grieta cualquiera.

Tabla 1: Métricas para todo el dataset aplicada esta segmentación.

Global Metrics	
Mean Accuracy (%)	Mean IoU (%)
75.69	8.21

Se puede ver en la Figura 1 que la segmentación no se está haciendo, llegando a detectar como grietas la propia textura del fondo. Y esta imagen no es la excepción, ya que la Tabla 1 refleja que este comportamiento se repite a lo largo de todo el dataset. Para ello nos fijamos en el valor $Mean\ IoU = 8.21$, el cual simboliza que se detecta menos del 10 % de la grieta. Es por esto que se hace importante modificar el algoritmo de segmentación para obtener mejores métricas.

Para la actualización del algoritmo tendremos en cuenta los siguientes pasos a seguir:

1. Lectura de la imagen, tanto la original en escala de grises como de la máscara para realizar las comparaciones posteriores. Se trabaja con la escala de grises porque así se reduce la complejidad del tratamiento al no tener que lidiar con paletas de colores, enfocándose así el estudio en texturas, bordes y variaciones de la intensidad lumínica.

2. Se modifica el filtro gaussiano por un **filtro bilateral**. La razón es porque, por un lado, el filtro gaussiano (como puede ser también el filtro de mediana) elimina el ruido, pero puede llegar a difuminar los bordes de importancia de las grietas, mientras que el filtro bilateral también puede reducir el ruido a la vez que conserva los contornos de las grietas.

3. Aplicación de umbralización adaptativa para binarizar la imagen, usando la técnica de Otsu para calcular automáticamente el mejor umbral para separar el fondo de la grieta. Esto porque no necesariamente vamos a tener el mismo contraste entre fondo y grieta en toda la imagen, por lo que aplicar un filtro "personalizado" optimiza el proceso. Acá tenemos un primer acercamiento a la segmentación de la grieta.

4. Se sigue usando la detección de bordes con *Canny* para buscar los cambios bruscos de intensidad en la imagen, los cuales pueden ser indicativos de grietas, identificando así los bordes.

5. Se sigue usando operaciones morfológicas para cerrar las líneas detectadas por *Canny*, el cual se encarga de dilatar y erosionar las detecciones hechas por Canny, uniendo posibles discontinuidades en las grietas. obteniendo así un segundo acercamiento a la segmentación de la grieta.

6. Dado que obtenemos la umbralización por Otsu y la detección por Canny, hacemos una combinación de ambos resultados para tener una segmentación más robusta.

7. Finalmente se buscan los contornos de la imagen combinada y se filtran los que son probablemente no pertenecientes a una grieta.

Este enfoque tiene por finalidad poder detectar y segmentar grietas que pueden ser más complejas de detectar debido a las condiciones lumínicas de las imágenes, además de aquellas que presentan formas más bien irregulares. Entonces, con estas modificaciones en mente, se procedió a generar una búsqueda de los mejores parámetros, teniendo en cuenta que los parámetros a optimizar eran los siguientes:

1. `bilateral_d`: diámetro del área del píxel alrededor del cual se aplica el filtro bilateral. Dependiendo de su tamaño, podemos preservar más o menos detalles en la imagen, a costa de o mantener parte del ruido o perder los detalles finos.

2. `canny_low`: Umbral más bajo para detectar bordes. Si es bajo puede detectar bordes falsos y si es alto ignorar bordes sutiles.

3. `canny_high`: Umbral más alto para detectar bordes. Si es bajo puede detectar muchos bordes, incluso los que pueden ser falsos, y si es alto sólo detectará los bordes más significativos.

4. `iterations`: Indica la cantidad de veces a aplicar el cierre morfológico. Si son pocas conectará pequeños fragmentos de grietas, pero si es alto conectará regiones más grandes de grietas que posiblemente no tienen conexión.

5. `adaptive_blockSize`: Tamaño de la ventana para umbralización adaptativa. Si es pequeña puede introducir ruido, ya que sería más sensible a los cambios locales, y si es grande puede detectar patrones más grandes a costa de perder detalles finos.

6. `adaptive_C`: Constante de ajuste del umbral. Si es bajo genera menos pixeles blancos (detecciones más estrictas) y es alto aumenta la sensibilidad, aumentado la generación de pixeles blancos.

La Tabla 2 muestra los diferentes valores obtenidos como resultado de la optimización, teniendo en cuenta que se quería obtener diferentes resultados dependiendo si queríamos mejorar el *accuracy*, *IoU* o una combinación de ambas:

Tabla 2: Parámetros a optimizar para segmentación.

	Maximizar IoU	Maximizar Accuracy	Maximizar IoU + Accuracy
`bilateral_d`	5	15	15
`canny_low`	50	100	100
`canny_high`	150	200	200
`iterations`	2	1	1
`adaptive_blockSize`	15	11	15
`adaptive_C`	2	5	5

Con estas modificaciones obtenemos el nuevo algoritmo de segmentación que se puede revisar en Código A.2, el cual dio diferentes resultados dependiendo de la combinación de hiperparámetros usada. La Tabla 3 muestra estos diferentes resultados. Podemos ver así que los resultados difieren bastante de lo obtenido preliminarmente y entre sí dependiendo de lo que se quería maximizar, habiendo en los tres casos altos valores de *Accuracy*. Esto se debe principalmente a que los parámetros dependen mucho del tipo de imagen que se ingresaba, además de que las imágenes presentes en el dataset variaban desde simples grietas en el suelo a separaciones entre ladrillos montados.

Tabla 3: Resultados con parámetros optimizados.

	Mean Accuracy (%)	Mean IoU (%)	Inference Time ([s])
Maximizar Accuracy	90.04	2.51	0.5786
Maximizar IoU	63.81	11.59	0.6461
Maximizar Accuracy + IoU	89.58	3.47	0.5738

Si aplicamos este nuevo algoritmo sobre una misma imagen usando los diferentes parámetros obtenidos, obtenemos los siguientes resultados:

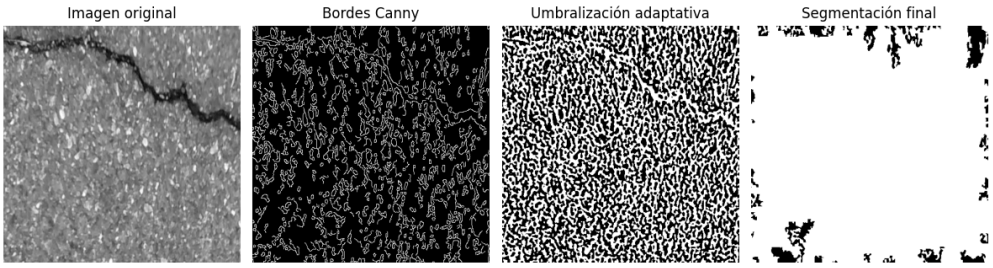

Figura 2: Segmentación de grieta usando parámetros que maximizan *IoU*.

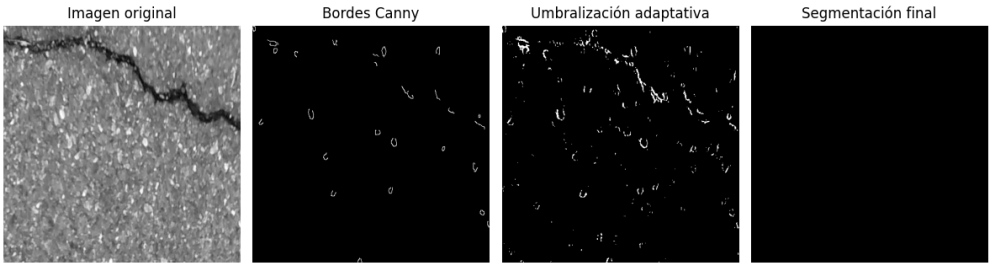

Figura 3: Segmentación de grieta usando parámetros que maximizan *Accuracy*.

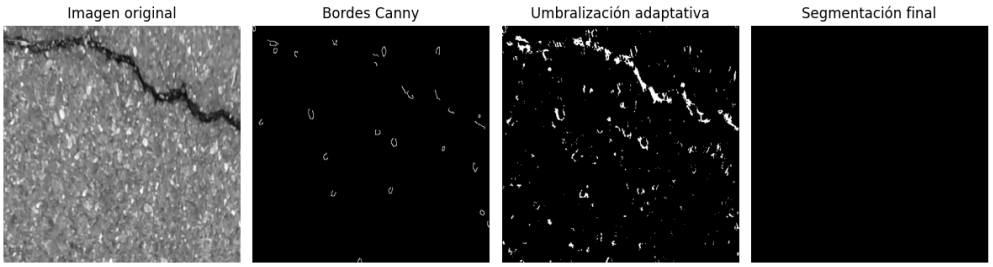

Figura 4: Segmentación de grieta usando parámetros que maximizan *IoU* y *Accuracy*.

En la Figura 2 podemos ver que la tarea de maximizar el *IoU* se cumple, pero se cumple logrando marcar de pixeles plancos casi toda la imagen. En la Figura 3 vemos que en la segmentación final no se logra obtener una máscara (recordar que esto es la combinación entre el filtro adaptativo y el filtro de contornos), pero algo se puede vislumbrar en la umbralización adaptativa. Caso similar se da en la Figura 4, en donde la segmentación adaptativa logra vislumbrar bastante la grieta, pero al combinarla con *Canny*, se pierde la grieta. Esto es aplicado a esta imagen, porque si vemos los resultados de las Figura 5, Figura 6 y Figura 7, podemos ver que para el caso de la maximización de *IoU*, logramos segmentar casi en su totalidad la grieta.

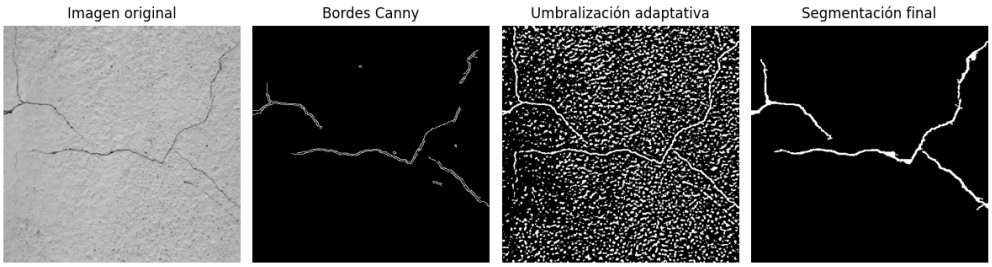

Figura 5: Segmentación de grieta usando parámetros que maximizan *IoU*.

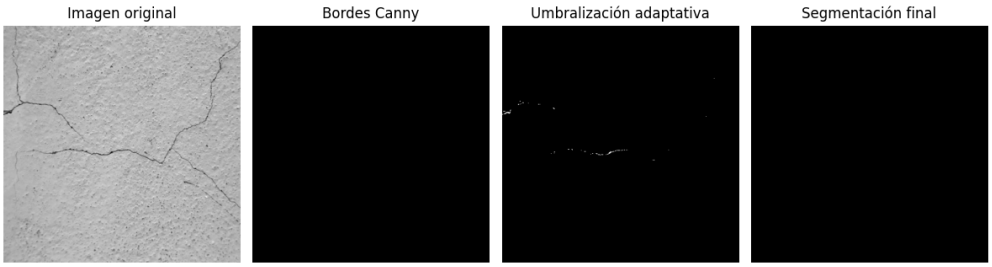

Figura 6: Segmentación de grieta usando parámetros que maximizan *Accuracy*.

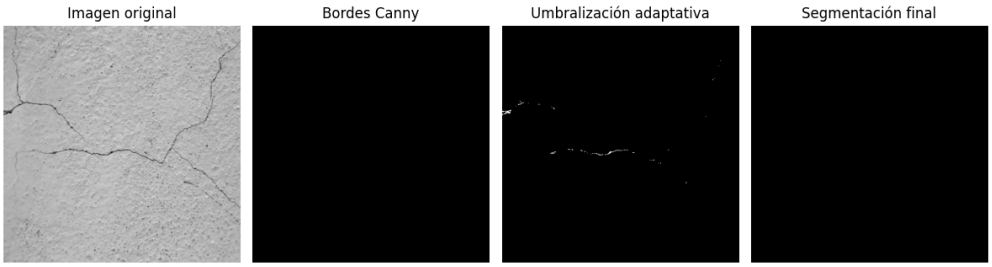

Figura 7: Segmentación de grieta usando parámetros que maximizan *IoU* y *Accuracy*.

2. Entrenamiento de una CNN

Ahora se usará una *CNN* para resolver el mismo problema de segmentación de grietas pero sobre un conjunto de imágenes más amplio y usando el *framework* de segmentación *MMSegmentation*, usando además un *hardware* más potente para el entrenamiento, usando como espacio de trabajo Google Colaboratory.

Cuando dentro de un *notebook* en Google Colab se ejecuta el comando `!nvidia-smi`, el resultado es el que aparecen la Figura 8, el cual indica lo siguiente:

- Información sobre versiones y sistema: se indica la versión de `nvisia-smi`, del controlador de la GPU y de CUDA. Este último es la plataforma de NVIDIA para realizar computación paralela, la cual permite usar la GPU para ejecutar tareas pesadas computacionalmente, tal como es el caso del entrenamiento de redes neuronales.
- Estado de al GPU: Acá muestra diversa información acerca de la GPU usada y sus características. Podemos ver que se trata de una Tesla T4, que se apaga luego de estar inactiva, espacio disponible para uso, temperatura y consumo de potencia.
- También tiene un apartado de procesos en ejecución. Según lo que se muestra en la imagen, en ese moento no habían procesos ejecutándose.

```
Fri Oct 25 15:40:34 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   39C    P8              9W /  70W |      3MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

Figura 8: Salida de la consola al ingresar el comando `nvidia-smi`

Para el modelo *CNN* entrenado se usaron 6 épocas de entrenamiento debido al límite de tiempo de uso de la GPU en Google Colab. Al finalizar, la evolución de las métricas se ve de la siguiente manera:

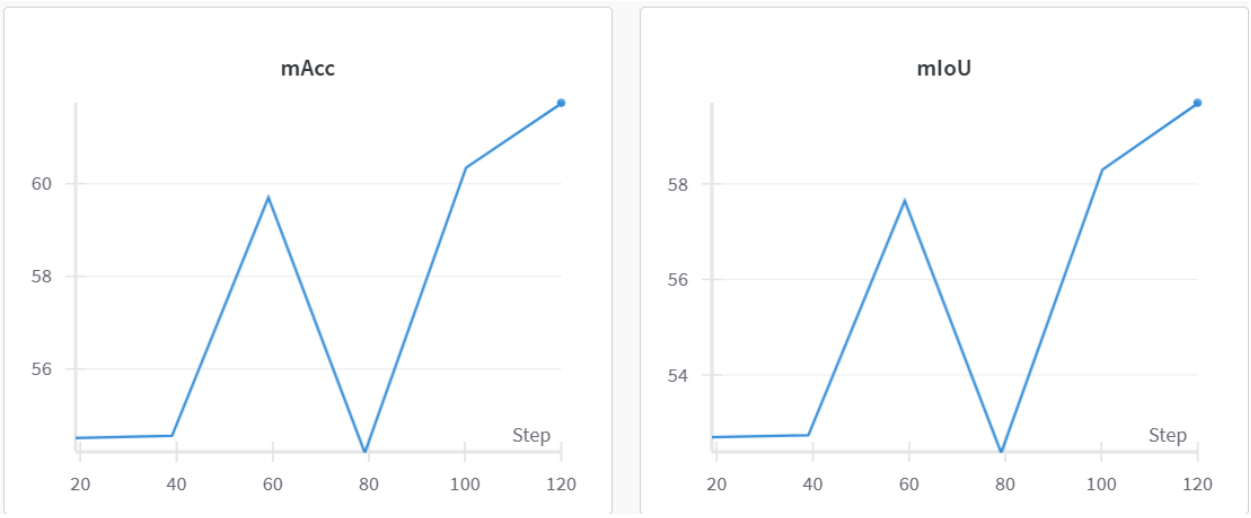

Figura 9: Evolución *Accuracy* e *IoU* a lo largo de las épocas.

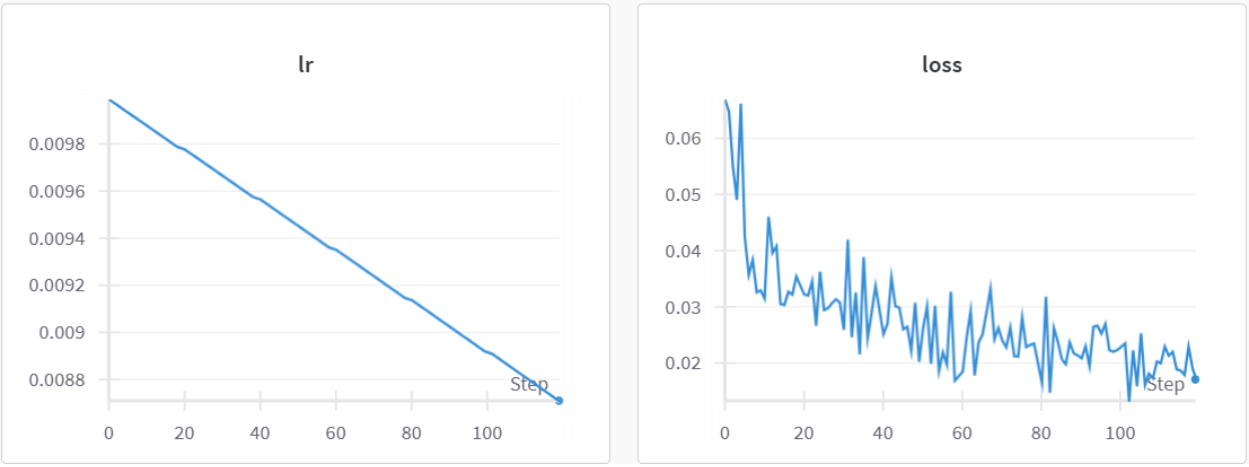

Figura 10: Evolución del *learning_rate* y *loss* a lo largo de las épocas.

Si nos concentramos en la Figura 9, vemos que ha un alza desde la primera época a la tercera época de entrenamiento, llegando a valores cercanos $mAcc \approx 60$ y $mIoU \approx 58$, pero tiene un decrecimiento bastante brusco en la cuarta época y llegando a sus máximos respectivos al final de la sexta época de entrenamiento.

Ahora, viendo los gráficos de la Figura 10, vemos que hay una baja sostenida en el *learning_rate* implementado, el cual partió en $lr \approx 0.01$ y terminó en $lr \approx 0.009$, mientras que el *loss* mostró un tendencia a la baja a lo largo de todo el entrenamiento, partiendo con un $loss \approx 0.07$ y terminando con $loss \approx 0.02$.

Si nos centramos en las métricas de *Accuracy* e *IoU* durante el entrenamiento para el fondo y la grieta, los resultados aparecen en la Tabla 4

Tabla 4: Evolución de *Acuracy* e *IoU* para fondo y grieta.

	Background		**Crack**	
	IoU	Acc	IoU	Acc
Epoch 1	96.44	99.95	8.99	9.11
Epoch 2	96.44	99.94	9.08	9.23
Epoch 3	96.68	99.79	18.63	19.62
Epoch 4	96.42	99.94	8.36	8.48
Epoch 5	96.73	99.79	19.86	20.91
Epoch 6	96.84	99.8	22.55	23.7

De acá podemos obtener que las mejores métricas de validación se obtuvieron al finalizar la última época de entrenamiento, en la iteración 5760 del total de 5760 que se ejecutaron. Podemos ver claramente, comparando los resultados de la Tabla 4 que estos resultados no se condicen con los obtenidos en la iteración anterior y que, aunque sean similares, son bastante más altos en escala. Esto también se ve apoyado por el gráfico en la evolución del *mAcc* y *mIoU* de la Figura 9.

Ahora, ¿cómo se ven estos resultados? En la Figura 11 podemos ver el resultado de segmentar la grieta de una imagen:

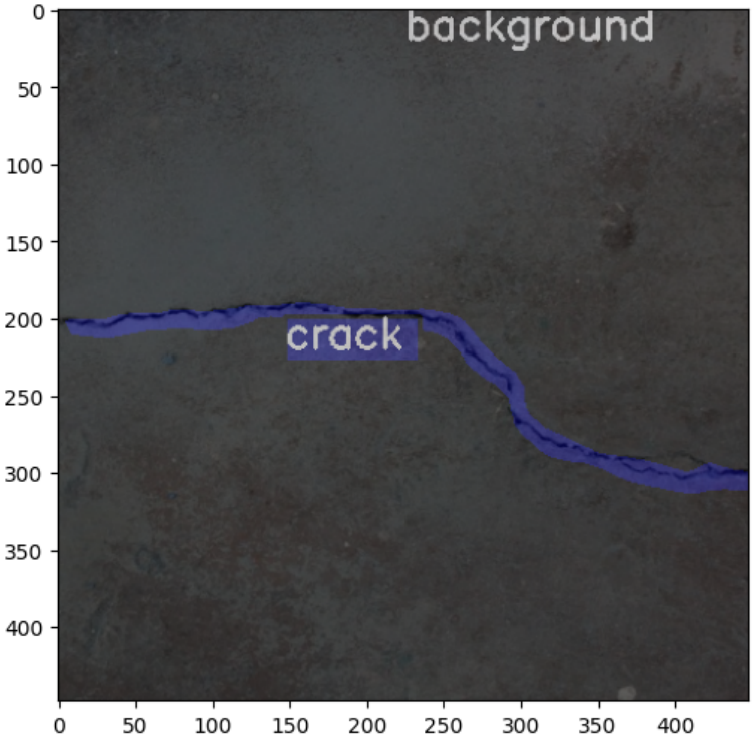

Figura 11: Resultado de segmentación usando *CNN*

Se puede ver que la segmentación se hizo de manera muy fina, siguiendo incluso las irregularidades que presentaba la imagen. Pero, si fijamos nuestra atención al final de la grieta en el lado derecho, hay una bifurcación que el modelo no logra detectar correctamente, omitiéndola.

Debido a un error interno en el código, no se pudo acceder a las visualizaciones que se pedían en el punto 11 de esta sección.

3. Comparación de rendimiento

A continuación se presenta una tabla comparativa con las métricas y tiempos:

Tabla 5: Comparación de métricas para todos los enfoques usados.

	Mean Accuracy (%)	Mean IoU (%)	Inference Time ([s])
Primer Algoritmo	75.69	8.21	0.5103
Maximizar Accuracy	90.04	2.51	0.5786
Maximizar IoU	63.81	11.59	0.6461
Maximizar Accuracy + IoU	89.58	3.47	0.5738
Uso de CNN	61.75	59.70	0.0768

Existen bastantes diferencias entre los modelos obtenidos de manera manual con el modelo *CNN* entrenado. De por sí la *CNN* obtuvo un resultado casi 5 veces mejor en la métrica *IoU* en comparación al mejor modelo manual, lo cual representa una clara ventaja en términos de segmentación correcta. El *Accuracy* no presenta resultados tan buenos debido posiblemente a que el entrenamiento estuvo mucho más enfocado a la detección de las grietas que en la concordancia de toda la imagen, pudiendo haber detectado más grietas en falsas en el fondo en comparación a los modelos manuales. También el tiempo de inferencia es considerablemente menor en la *CNN* que en el resto de modelos.

Pongamos de ejemplo la imagen que se ve en la Figura 11 y comparémosla con lo obtenido por los modelos manuales. En las Figura 12, Figura 13 y Figura 14 podemos ver los diferentes resultados para esta misma segmentación dependiendo de lo que se maximizó:

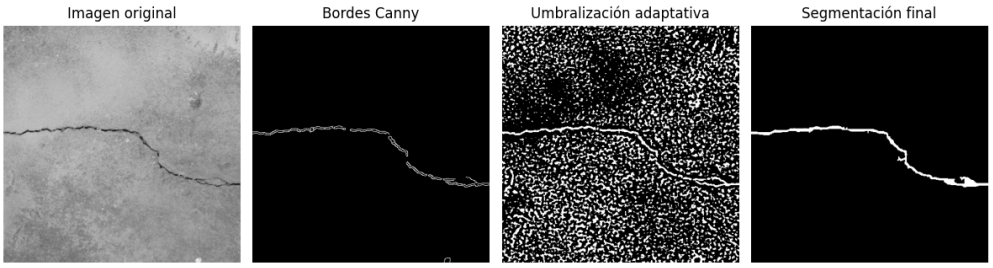

Figura 12: Segmentación de grieta ejemplo usando parámetros que maximizan *IoU*.

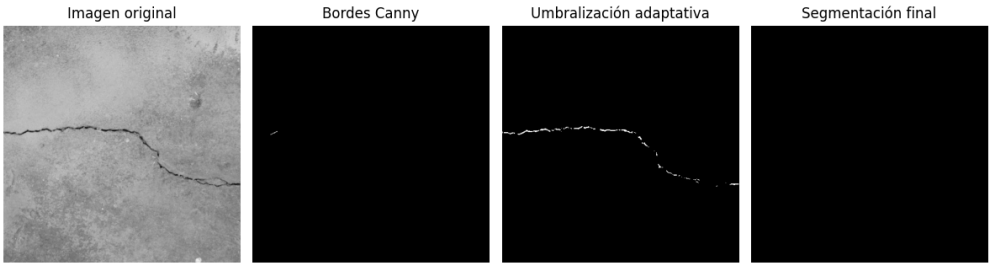

Figura 13: Segmentación de grieta ejemplo usando parámetros que maximizan *Accuracy*.

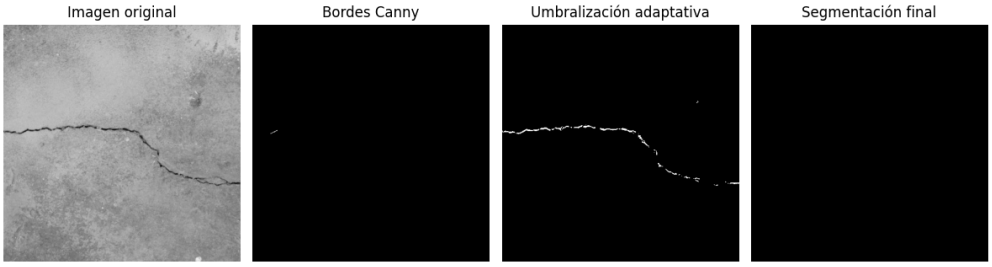

Figura 14: Segmentación de grieta ejemplo usando parámetros que maximizan *IoU* y *Accuracy*.

El patrón se repite. Mientras la umbralización adaptativa da resultados aceptables (independiente de lo que se maximice), el filtro *Canny* muestra ser más inflexible y menos efectivo cuando se busca maximizar el *Accuracy* o una mezcla de ambas métricas. Además, en el caso de la Figura 12, se puede ver que, a diferencia de la *CNN*, se logra segmentar algo de la bifurcación de la grieta.

En general, el modelo que tuvo un comportamiento mucho más sólido y menos cambiante es el *CNN*, tanto por la complejidad de su estructura como por la cantidad de entrenamientos necesitados para alcanzar estos resultados. Si hubiese sido entrenada con menos épocas, su rendimiento no habría sido muy diferente al de los modelos manuales, por lo que es importante destacar el tiempo de entrenamiento que este tipo de modelos necesitan. Además, si nos centramos en los modelos manuales, el uso de un umbral adaptativo resultó (a lo menos cualitativamente en los ejemplos) bastante más efectivo que el uso de *Canny* como se venía haciendo desde la actividad pasada.

Anexo A. Mejora de algoritmo *handcrafted* de segmentación de grietas

Código A.1: Algoritmo de segmentación de grieta preliminar.

```
def my_segmentation_cracks(image_path):
    '''
    Recibe de entrada el PATH a una imagen de grieta y desarrolla
    cuatro procesos para la segmentación de la grieta:
    1. Conversión a escala de grises.
    2. Aplicación de filtro gaussiano.
    3. Detección de bordes usando Canny.
    4. Operación morfológica de cierre.

    Args
        image_path (str): PATH en donde se tiene almacenada la imagen.

    Return
        closed (): La imagen segmentada.
        mask (): La máscara de la imagen de entrada.
    '''
    # Lectura de la máscara
    mask_path = image_path.replace('images', 'masks')
    mask = cv2.imread(mask_path, cv2.IMREAD_GRAYSCALE)

    # Lectura y conversión a escala de grises
    imagen_rgb = cv2.cvtColor(cv2.imread(image_path), cv2.COLOR_BGR2RGB)
    imagen_gray = cv2.cvtColor(imagen_rgb, cv2.COLOR_RGB2GRAY)

    # Aplicación de filtro gaussiano
    imagen_blur = cv2.GaussianBlur(imagen_gray, (3, 3), 0)

    # Detección de bordes usando Canny
    imagen_canny = cv2.Canny(imagen_blur, 50, 200)

    # Operación morfológica de cierre
    imagen_cierre = cv2.morphologyEx(imagen_canny, cv2.MORPH_CLOSE,np.ones((3,3),
     ↪ np.uint8), iterations=2)

    return imagen_cierre, mask
```

Código A.2: Algoritmo de segmentación posterior optimización.

```
def better_segmentation(image_path, params):
    '''
```

```
    Segmentación de grietas.

    Args:
        image_path (str): PATH de la imagen de entrada.
        params (dict): Diccionario con los parámetros de los filtros.

    Return:
        prediction (np.array): Imagen binaria con la predicción de grietas.
        mask (np.array): Máscara de referencia para comparación.
    '''
    # Leer la imagen en escala de grises
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    mask_path = image_path.replace('images', 'masks')
    mask = cv2.imread(mask_path, cv2.IMREAD_GRAYSCALE)

    # 1. Suavizado con filtro bilateral
    blurred = cv2.bilateralFilter(
        image,
        d=params["bilateral_d"],
        sigmaColor=75,
        sigmaSpace=75
    )

    # 2. Detección de bordes usando Canny
    edges = cv2.Canny(blurred, params["canny_low"], params["canny_high"])

    # 3. Operación morfológica para cerrar huecos en los bordes detectados
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (7, 7))
    closed = cv2.morphologyEx(edges, cv2.MORPH_CLOSE, kernel, iterations=params["
     ↪ iterations"])

    # 4. Umbralización adaptativa para refinar la segmentación
    adaptive_thresh = cv2.adaptiveThreshold(
        blurred, 255,
        cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
        cv2.THRESH_BINARY_INV,
        blockSize=params["adaptive_blockSize"],
        C=params["adaptive_C"]
    )

    # 5. Combinar los resultados de Canny y umbralización
    combined = cv2.bitwise_or(closed, adaptive_thresh)

    # 6. Encontrar contornos y dibujar las grietas detectadas
    contours, _ = cv2.findContours(combined, cv2.RETR_EXTERNAL, cv2.
     ↪ CHAIN_APPROX_SIMPLE)
    prediction = np.zeros_like(image)
```

```
48
49      for contour in contours:
50          if cv2.contourArea(contour) > 500:  # Filtrar áreas pequeñas
51              cv2.drawContours(prediction, [contour], -1, 255, thickness=cv2.FILLED)
52
53      return prediction, mask
```