SECCIÓN II

PROCESAMIENTO DIGITAL DE IMÁGENES E INTELIGENCIA ARTIFICIAL

Nueva sección, nuevo tipo de señal: en este caso, **señales bidimensionales**, específicamente imágenes. Y te preguntarás:

¿Por qué es necesaria una sección aparte para este tipo de señales?

La respuesta es sencilla. Aunque muchas de las ideas fundamentales del procesamiento digital siguen siendo las mismas, las técnicas utilizadas cambian considerablemente. El tipo de ruido ya no es el mismo que encontramos en las señales de audio, y también cambian los filtros y herramientas que nos permitirán mejorar la calidad de la imagen. Además, aparecen conceptos que no desempeñaban un papel relevante en el procesamiento de audio, pero que resultan fundamentales en el procesamiento digital de imágenes, como los histogramas, la ecualización, la detección de bordes, las operaciones morfológicas y la segmentación.

En algunos programas de ingeniería, los cursos de procesamiento de señales unidimensionales (1D) y bidimensionales (2D) se estudian por separado. Sin embargo, hacerlo de esta manera puede hacer que se pierda una visión más amplia del procesamiento digital de señales. Aunque las técnicas específicas sean diferentes, muchos de los conceptos estudiados en la sección anterior continúan presentes: representación digital, eliminación de ruido, filtrado,

evaluación de resultados e incluso la Inteligencia Artificial como una herramienta para resolver problemas reales.

En esta sección recorreremos tres etapas complementarias. En el **Capítulo 4** estudiaremos los fundamentos del procesamiento digital de imágenes, comprendiendo cómo se representan, mejoran y analizan las imágenes digitales.

En el **Capítulo 5** aplicaremos estos conceptos para resolver un problema real de segmentación mediante técnicas clásicas de procesamiento digital de imágenes.

Finalmente, en el **Capítulo 6** abordaremos el mismo problema utilizando una arquitectura **U-Net**, mostrando cómo la Inteligencia Artificial es capaz de aprender automáticamente la tarea de segmentación a partir de ejemplos previamente etiquetados.

Más que presentar dos soluciones diferentes, el objetivo será comprender dos formas de abordar un mismo problema. La primera, basada en reglas diseñadas explícitamente por el ingeniero; la segunda, basada en modelos de aprendizaje profundo capaces de aprender dichas reglas directamente a partir de los datos. Al finalizar esta sección no solo conocerás ambas estrategias, sino que también estarás en capacidad de comprender sus ventajas, limitaciones y los escenarios en los que resulta más conveniente utilizar cada una de ellas.

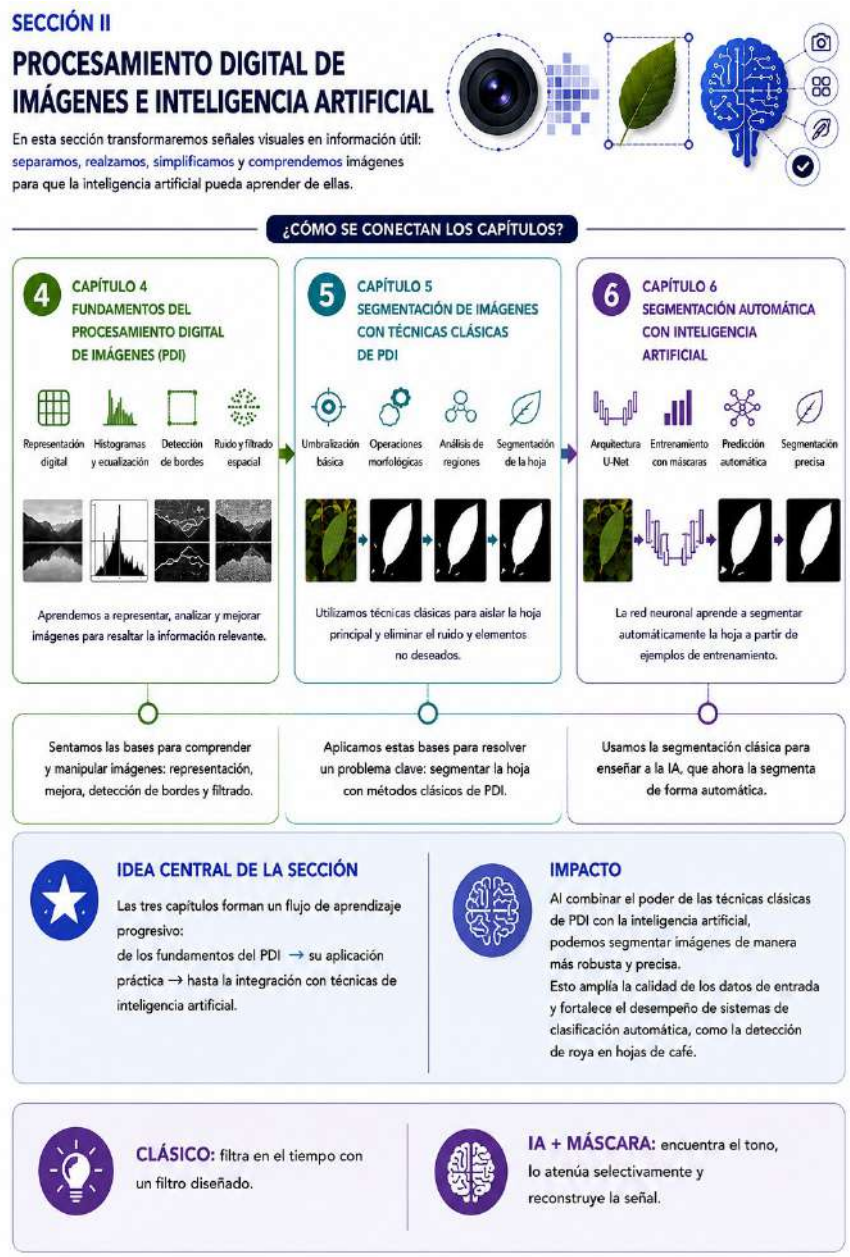


Figura 66. Sección II: desde los fundamentos del procesamiento digital de imágenes (PDI), pasando por la segmentación mediante técnicas clásicas, hasta la segmentación automática con Inteligencia Artificial mediante una arquitectura U-Net.

La **Figura 66** resume el recorrido que seguiremos durante esta sección. Comenzaremos comprendiendo los fundamentos del procesamiento digital de imágenes y las principales herramientas que nos permitirán mejorar, analizar y extraer información de una imagen. Posteriormente, aplicaremos estos conceptos para resolver un problema real de segmentación utilizando técnicas clásicas de procesamiento digital de imágenes. Finalmente, abordaremos exactamente el mismo problema desde una perspectiva completamente diferente, utilizando Inteligencia Artificial y una arquitectura U-Net capaz de aprender automáticamente a segmentar una imagen.

Espero que esta sección no solo te permita aprender nuevas técnicas. También deseo que, si ya estudiaste procesamiento digital de imágenes hace algunos años, tengas la oportunidad de **redescubrir** muchos de esos conceptos desde una perspectiva diferente, integrándolos con herramientas modernas como Python y la Inteligencia Artificial. A veces, volver sobre un tema no significa repetir lo aprendido, sino comprenderlo con una mayor profundidad.

Si al terminar estas páginas logras comprender que un mismo problema puede resolverse tanto mediante reglas diseñadas por un ingeniero como mediante modelos capaces de aprender directamente de los datos, y además disfrutas ese proceso de aprendizaje, entonces el objetivo de esta sección se habrá cumplido.

CAPÍTULO IV

FUNDAMENTOS DEL PROCESAMIENTO DIGITAL DE IMÁGENES

Los tres primeros capítulos del libro estuvieron enfocados en señales unidimensionales (1D), y específicamente los Capítulos 2 y 3 en señales de voz. Este tipo de señales dependen típicamente del tiempo y pueden representarse como una secuencia ordenada de muestras.

Ahora, en la Sección II trabajaremos con imágenes, las cuales corresponden a señales bidimensionales (2D). A diferencia de las señales de audio, las imágenes describen información espacial y, en la mayoría de las aplicaciones, no dependen explícitamente del tiempo.

Es decir, mientras una señal 1D puede interpretarse como una secuencia de muestras, una imagen puede representarse como una matriz cuyos elementos contienen información de intensidad o color.

Dependiendo del tipo de imagen, los valores almacenados en cada elemento de la matriz pueden tener diferentes formatos. En las imágenes más comunes, cada píxel se almacena utilizando 8 bits sin signo (*uint8*), permitiendo representar valores enteros en el rango de 0 a 255. Sin embargo, también existen imágenes binarias, imágenes de 16 bits e incluso imágenes en punto flotante utilizadas en aplicaciones científicas y médicas.

La Figura 67 presenta una comparación entre las principales características de las señales unidimensionales y las señales bidimensionales.

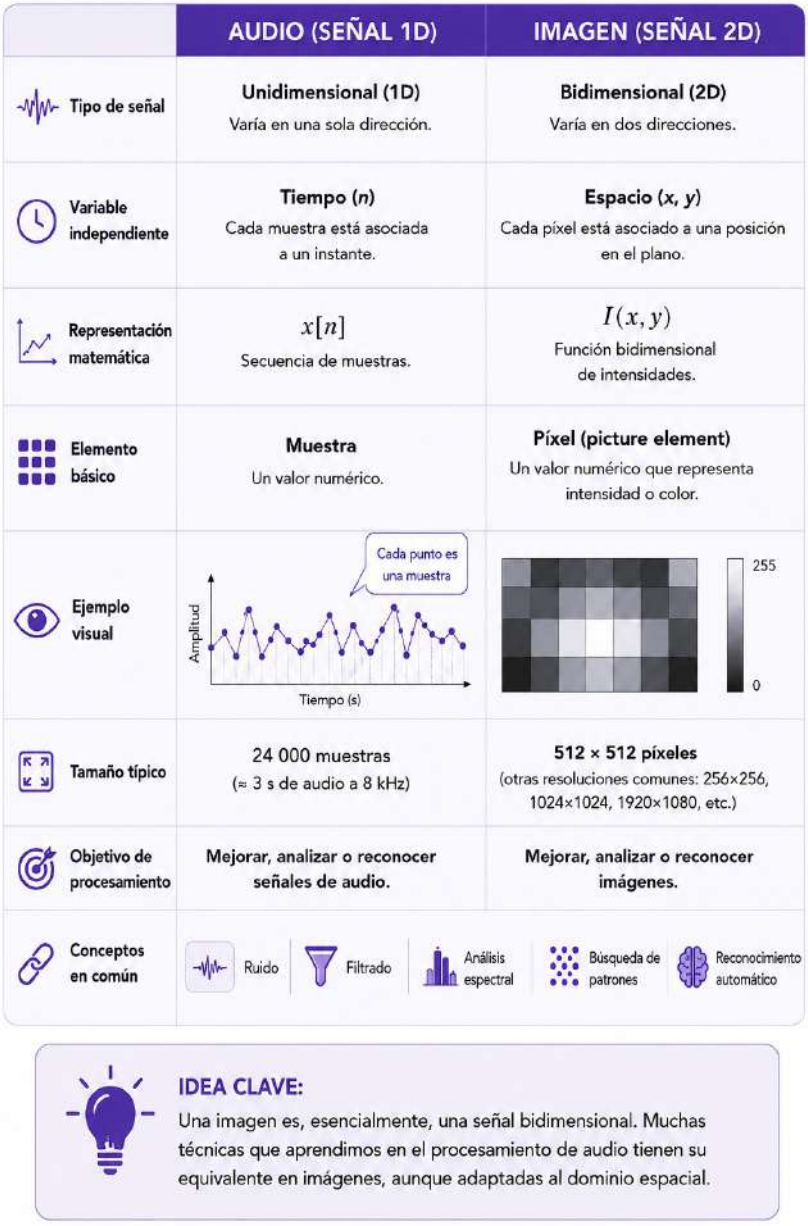


Figura 67. Comparación señal de audio vs. imagen: tipo de señal, variable independiente, representación matemática, elemento básico, ejemplo visual, tamaño típico, objetivo de procesamiento.

En el caso de las imágenes, la representación matemática más común es:

$$I(x,y)$$

donde:

- x corresponde a una coordenada espacial horizontal.
- y corresponde a una coordenada espacial vertical.

Cada par (x,y) identifica un píxel dentro de la imagen y el valor asociado representa su intensidad o color.

4.1 Resolución espacial y profundidad de bits

Existen dos conceptos fundamentales que permiten caracterizar una imagen digital: la **resolución espacial** y la **profundidad de bits**.

Comencemos con la **profundidad de bits**. Una imagen puede almacenar diferente cantidad de información en cada píxel dependiendo del número de bits utilizados para representarlo. Por ejemplo, podemos encontrar imágenes de 1 bit por píxel, 8 bits por píxel o 24 bits por píxel.

En el primer caso, tenemos imágenes binarias (*Black and White*, B/W), donde cada píxel puede tomar únicamente uno de dos valores posibles: 0 (negro) o 1 (blanco).

También, encontramos las imágenes en escala de grises, que habitualmente utilizan 8 bits por píxel. Esto significa que cada píxel puede tomar valores entre:

$$00000000_2 = 0$$

y

$$11111111_2 = 255$$

De esta forma, además del negro y el blanco, disponemos de 254 tonalidades intermedias de gris que permiten representar transiciones mucho más suaves en la imagen.

Finalmente, tenemos las imágenes a color en modelo RGB (*Red, Green, Blue*). En este caso, la imagen está compuesta por tres canales independientes:

- Canal rojo (*Red*).
- Canal verde (*Green*).
- Canal azul (*Blue*).

Cada canal utiliza típicamente 8 bits, para un total de:

$$8 + 8 + 8 = 24 \text{ bits por píxel}$$

La combinación de las intensidades de estos tres canales permite representar millones de colores diferentes (exactamente 2^{24} colores diferentes).

La Figura 68 presenta una comparación de los tres tipos de imágenes que abordaremos en esta sección. Cabe aclarar que existen otros tipos de imágenes, como las imágenes multiespectrales e hiperespectrales, que quedan por fuera del alcance de este libro.

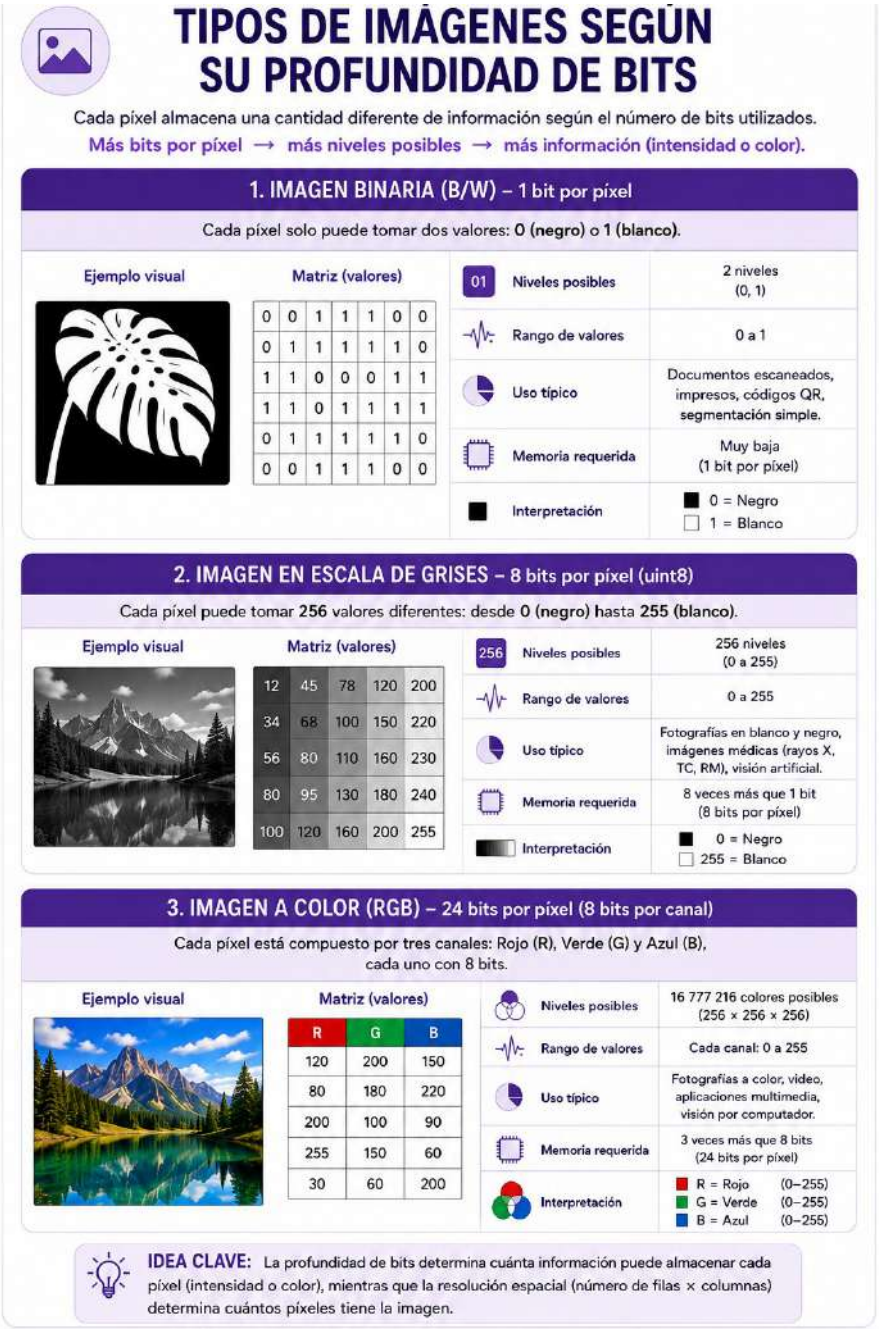


Figura 68. Tipos de imágenes según su profundidad de bits: binaria (B/W), en escala de grises y a color (RGB).

Un aspecto importante que quiero resaltar corresponde al aumento en los requerimientos de almacenamiento a medida que se incrementa la profundidad de bits. Por ejemplo, si una imagen RGB posee las mismas dimensiones que una imagen en escala de grises, su “peso” será aproximadamente tres veces mayor, dado que contiene tres canales (rojo, verde y azul) en lugar de uno solo. De forma similar, una imagen RGB requerirá aproximadamente veinticuatro veces más memoria que una imagen binaria, ya que cada píxel se representa mediante 24 bits en lugar de 1 bit (y digo *aproximadamente* porque, además de los píxeles, también se almacena información adicional como encabezados y metadatos).

En consecuencia, una mejor representación visual y una mayor cantidad de información por píxel traen consigo un costo adicional en términos de almacenamiento, transmisión y procesamiento. Como veremos más adelante, este compromiso entre calidad visual y recursos computacionales es una consideración frecuente en aplicaciones de procesamiento digital de imágenes. En ingeniería, como ocurre en muchos otros problemas, obtener más información suele implicar consumir más recursos.

Y ahora, pasemos al concepto de **resolución espacial**, el cual nos indica cuántos píxeles componen una imagen.

La resolución espacial suele expresarse mediante el número de filas y columnas de la matriz que representa la imagen. Por ejemplo, una imagen de:

512×512

posee 512 filas y 512 columnas, para un total de:

$512 \times 512 = 262{,}144$ píxeles.

De manera similar, una imagen Full HD tiene una resolución de:

1920×1080 lo que equivale a más de dos millones de píxeles.

Cuantos más píxeles tengamos disponibles, más pequeños serán los detalles que podremos observar. Sin embargo, como ya vimos al hablar de la profundidad de bits, este aumento en la calidad también tiene un costo asociado: se requiere más memoria para almacenar la imagen y más recursos computacionales para procesarla.

La Figura 69 presenta una comparación de una misma imagen utilizando diferentes resoluciones espaciales. Observa cómo, a medida que disminuye el número de píxeles, la imagen comienza a perder detalle y las transiciones se vuelven más abruptas.



Figura 69. Efecto de la resolución espacial sobre la calidad visual de la imagen.

Existe una analogía interesante con las señales de audio estudiadas en capítulos anteriores. En una señal de voz utilizábamos la frecuencia de muestreo (f_s) para determinar cuántas muestras se almacenaban por segundo (a mayor f_s, el audio "sonaba mejor"). En una imagen ocurre algo similar: la resolución espacial determina en cuántas filas y columnas se divide la imagen.

En ambos casos, aumentar el número de muestras o píxeles permite representar mejor la información original. Sin embargo, esta mejora tiene un costo asociado: se requiere más memoria para almacenar los datos y más recursos computacionales para procesarlos.

Ejemplo 21. Efecto de la resolución espacial y la profundidad de bits de la imagen

Supongamos que tenemos una imagen de:

512×512

Esto significa que la imagen contiene:

$512 \times 512 = 262,144 512 \times 512 = 262,144$ píxeles

Si la imagen es binaria, cada píxel requiere 1 bit:

$262,144 \times 1 = 262,144$ bits lo que equivale a:

$32,768$ bytes ≈ 32 KB

Nota: *En este libro utilizaremos las equivalencias tradicionales empleadas en informática, donde 1 KB = 1024 bytes y 1 MB = 1024 KB.*

Si la imagen está en escala de grises, cada píxel requiere 8 bits:

$262,144 \times 8 = 2'097,152$ bits es decir:

$262,144$ bytes ≈ 256 KB

Finalmente, si la imagen es RGB, cada píxel requiere 24 bits:

$262,144 \times 24 = 6'291,456$ bits

lo que equivale a:

$786,432$ bytes ≈ 768 KB

Podemos observar que la resolución espacial permanece igual en los tres casos. Lo que cambia es la profundidad de bits. Por esta razón, una imagen RGB de 512×512 píxeles requiere aproximadamente tres veces más memoria que una imagen en escala de grises y veinticuatro veces más que una imagen binaria.

Este ejemplo nos permite ver que el tamaño de una imagen depende de dos factores: cuántos píxeles tiene y cuántos bits se usan para representar cada píxel.

$$\text{Tamaño en bits} = \text{filas} \times \text{columnas} \times \text{bits por píxel}$$

Aunque se trata de un ejemplo simplificado que no considera encabezados ni mecanismos de compresión, permite comprender cómo la resolución espacial y la profundidad de bits impactan directamente el tamaño de almacenamiento requerido.

4.2 Formatos de imagen y compresión

Hasta este punto hemos asumido que las imágenes se almacenan exactamente como las hemos descrito: píxel por píxel y utilizando todos los bits necesarios para representar la información. De hecho, los tamaños calculados en el Ejemplo 21 corresponden a esta situación ideal, donde no existe ningún tipo de compresión.

Sin embargo, hagamos una pausa y pensemos en algo cotidiano. ¿Cuántas fotografías tienes almacenadas actualmente en tu celular? ¿Cien? ¿Mil? ¿Tal vez varias decenas de miles? Ahora imagina que cada una de ellas ocupara varios megabytes. Muy rápidamente nos quedaríamos sin espacio de almacenamiento.

Y el problema no termina ahí. Las imágenes no solamente se almacenan; también se envían por WhatsApp, se publican en redes sociales, se cargan en páginas web y se transmiten continuamente a través de Internet. Si cada fotografía mantuviera siempre su tamaño original, el espacio de almacenamiento se llenaría rápidamente (más).

Por esta razón, se desarrollaron diferentes métodos de compresión de imágenes cuyo objetivo es reducir el tamaño de los archivos sin afectar excesivamente su apariencia visual. Sin embargo, no todos producen el mismo efecto sobre la imagen. En términos generales, podemos clasificarlos en dos grandes categorías: compresión *lossless* y compresión *lossy*.

- **Compresión sin pérdida de información (*lossless*):** permite recuperar exactamente la imagen original después de descomprimirla.
- **Compresión con pérdida de información (*lossy*):** reduce aún más el tamaño del archivo eliminando parte de la información visual que se considera poco relevante para la percepción humana.

Entre los formatos más conocidos de compresión *lossless* encontramos PNG, GIF, TIFF y BMP. Por su parte, entre los formatos lossy destacan JPEG/JPG, WebP y HEIF/HEIC.

La Figura 70 presenta una comparación general de ambos enfoques.

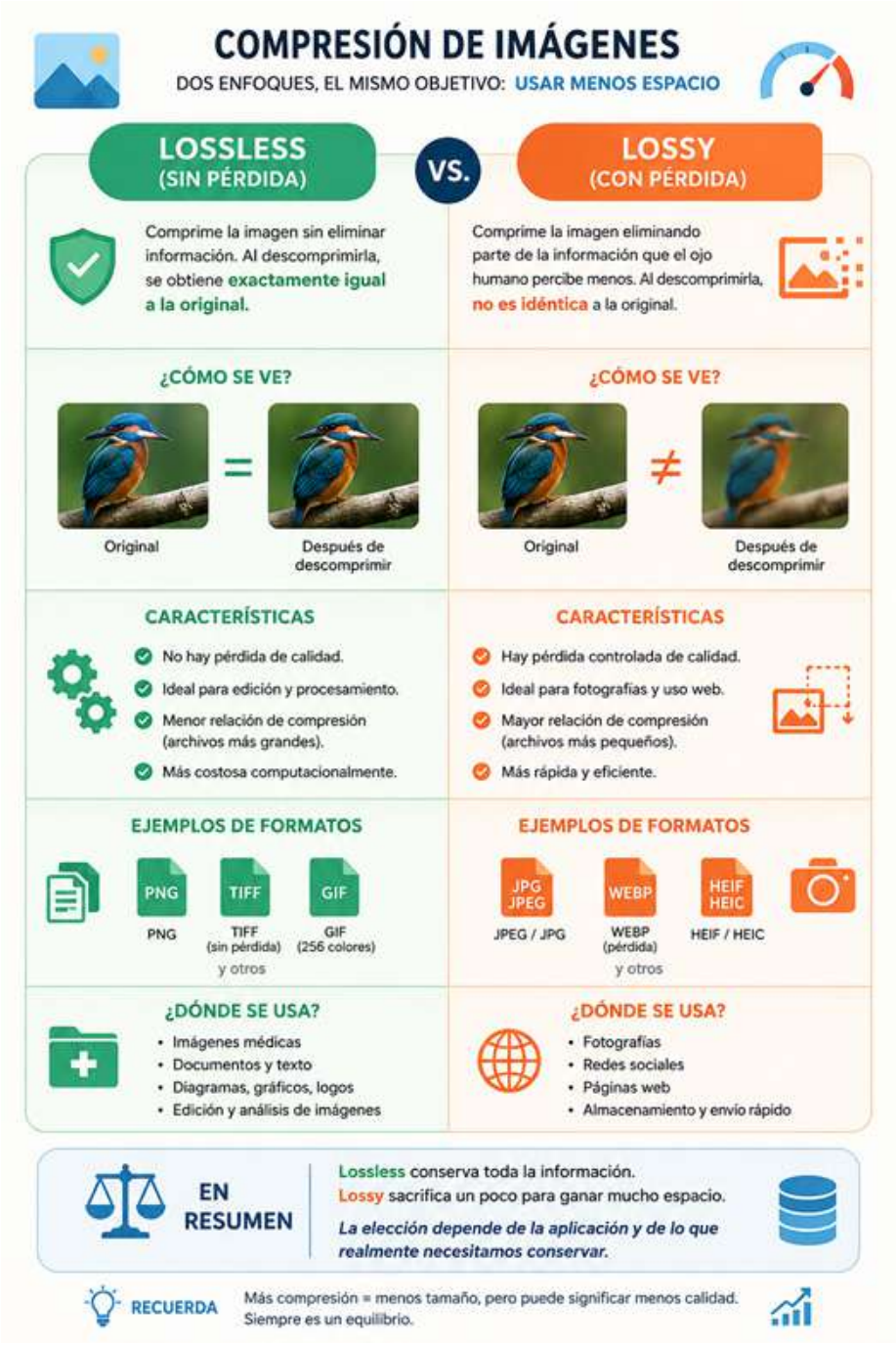


Figura 70. Compresión de imágenes: formatos *lossless* y *lossy*.

No podemos afirmar que una alternativa sea siempre mejor que la otra. La elección depende en gran medida de la aplicación. Por ejemplo, en imágenes médicas suele ser indispensable utilizar formatos *lossless*, ya que preservan todos los detalles de la imagen, algunos de los cuales pueden ser fundamentales para identificar una patología o realizar un diagnóstico adecuado.

Por el contrario, cuando se requiere almacenar o transmitir grandes cantidades de imágenes, suelen preferirse formatos *lossy*, ya que permiten reducir significativamente el tamaño de los archivos. En muchos casos, la pérdida de información es tan pequeña que resulta prácticamente imperceptible para el ojo humano.

En consecuencia, la selección del formato de imagen representa un compromiso entre calidad visual, capacidad de almacenamiento y velocidad de transmisión, una situación que encontraremos con frecuencia a lo largo de este libro.

4.3 Histograma y ecualización

Hasta este punto hemos estudiado cómo se almacena una imagen, cuántos píxeles la componen, cuántos bits utiliza cada uno de ellos y qué formatos existen para almacenarla. Sin embargo, aún no hemos analizado una pregunta fundamental: ¿cómo se distribuyen los valores de intensidad dentro de una imagen?

La respuesta la encontramos en el histograma. De manera intuitiva, es una representación gráfica de la distribución de los píxeles de una imagen. En otras palabras, indica cuántos píxeles existen para cada uno de los niveles de intensidad o color presentes en la imagen. Por ejemplo, en una imagen en escala de grises de 8 bits, los píxeles pueden tomar valores entre 0 y 255. El histograma contará cuántos píxeles poseen intensidad 0, cuántos poseen intensidad 1, cuántos poseen intensidad 2 y así sucesivamente hasta llegar a 255.

Matemáticamente, el histograma puede representarse como:

$h(i)$ donde:

i corresponde al nivel de intensidad.

$h(i)$ representa la cantidad de píxeles que poseen dicha intensidad.

La Figura 71 ilustra la relación entre una imagen en escala de grises y su correspondiente histograma.

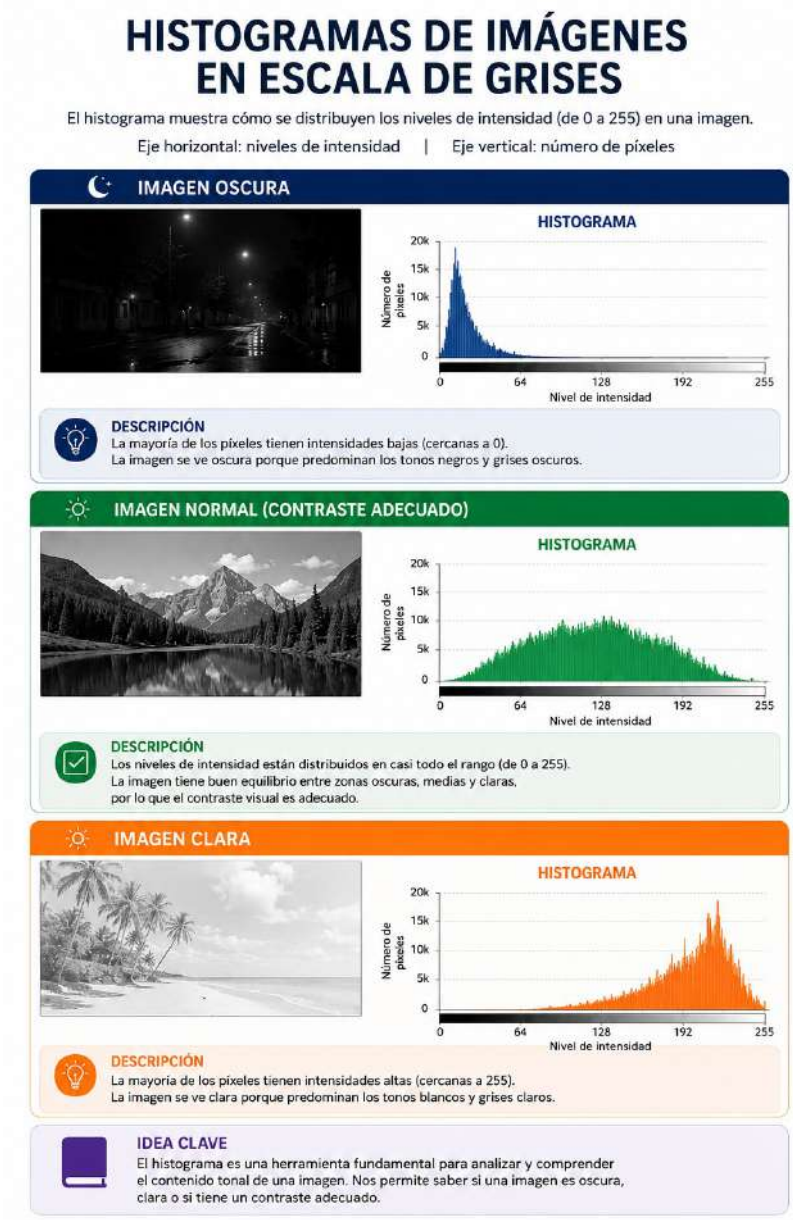


Figura 71. Ejemplos de histogramas de imágenes con diferentes características: oscura (izquierda), contraste adecuado (central) y clara (derecha).

Es importante aclarar que dos imágenes completamente diferentes pueden tener histogramas similares. Dado que, el histograma no describe la posición espacial de los objetos, sino únicamente la distribución global de intensidades presentes en la imagen.

Pero ¿cómo podemos interpretar un histograma?

Vamos a realizar este análisis con los tres ejemplos de la Figura 71. En la imagen de la izquierda, la distribución se concentra principalmente en niveles bajos de intensidad, dando lugar a una imagen oscura. En la imagen central, los niveles de intensidad ocupan una región más amplia del rango disponible, produciendo un contraste adecuado. Finalmente, en la imagen de la derecha, la distribución se desplaza hacia niveles altos de intensidad, generando una imagen clara.

Típicamente, una imagen oscura la obtenemos cuando tomamos una fotografía de noche sin utilizar flash. Por el contrario, una imagen clara puede aparecer cuando tomamos una fotografía en un día muy soleado. En ambos casos, el histograma nos permite identificar rápidamente estas características incluso antes de observar la imagen en detalle.

Ahora, ¿qué podemos hacer para mejorar la distribución de pixeles de la imagen?

La respuesta la encontramos en la **ecualización de histograma**, una de las técnicas más utilizadas para mejorar el contraste de una imagen.

La idea detrás de esta técnica es bastante sencilla. La ecualización redistribuye los niveles de intensidad de la imagen para aprovechar mejor el rango disponible. Como resultado, se mejora el contraste y se facilita la visualización de los detalles presentes en la imagen.

La Figura 72 ilustra de forma conceptual el efecto de la ecualización sobre una imagen y su correspondiente histograma.

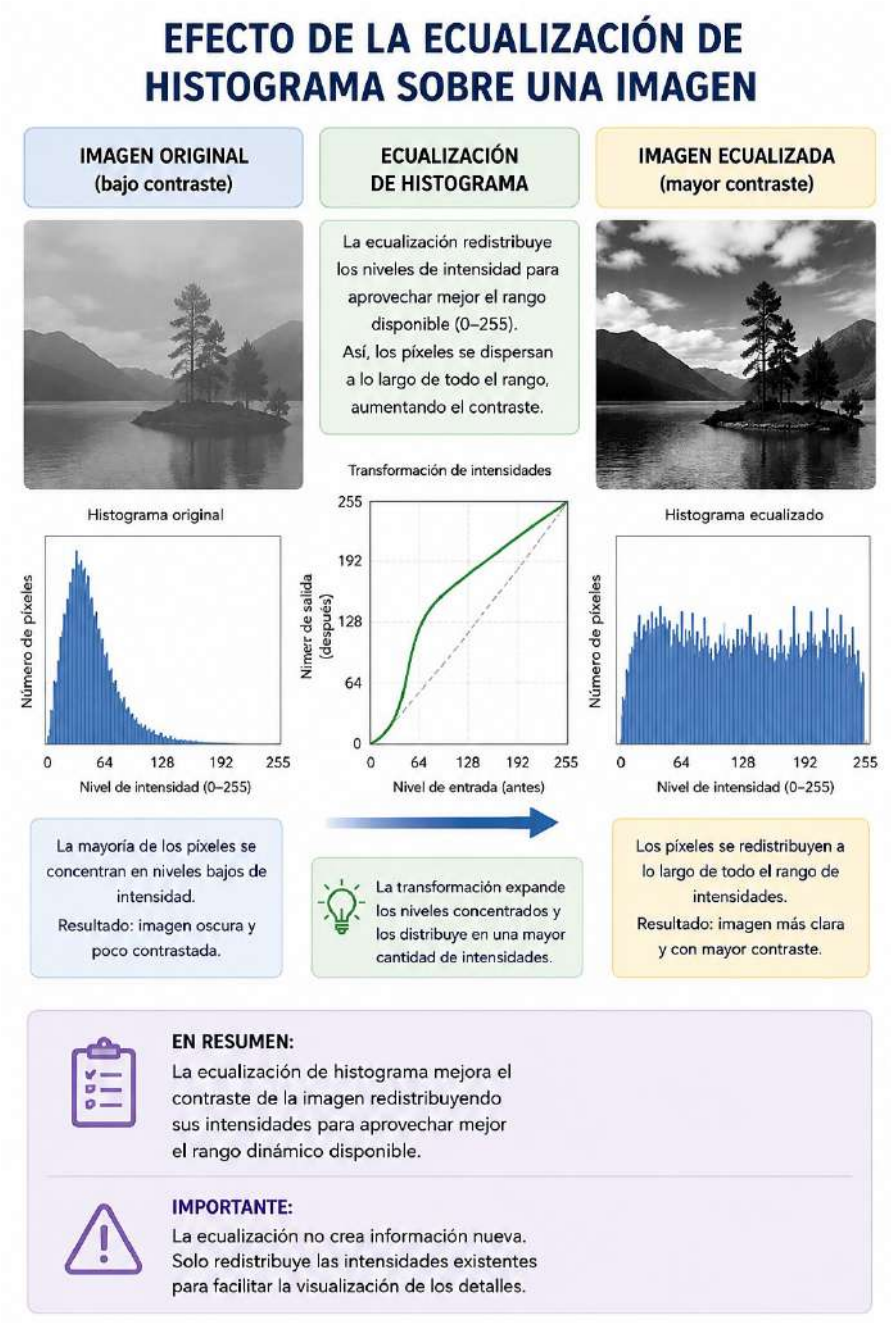


Figura 72. Efecto de la ecualización de histograma sobre una imagen.

Después de aplicar la ecualización, los detalles que antes resultaban difíciles de distinguir pueden hacerse más visibles, especialmente en imágenes con bajo contraste.

Esta técnica es ampliamente utilizada en aplicaciones como: procesamiento de imágenes médicas, análisis satelital, visión por computador, sistemas de inspección industrial y restauración de fotografías.

Sin embargo, la ecualización no siempre produce mejoras visuales. En algunas imágenes puede generar una apariencia artificial o inclusive amplificar ruido presente en regiones oscuras o brillantes. Por esta razón, su utilización debe evaluarse de acuerdo con la aplicación específica.

Ejemplo 22. Ecualización de imagen a escala de grises

A continuación, veremos un ejemplo en Python para ecualizar una imagen a escala de grises:

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('/content/imagen_oscura.png', cv2.IMREAD_GRAYSCALE)
img_eq = cv2.equalizeHist(img_dark)

cv2.imwrite("imagen_oscura.png", img_dark)
cv2.imwrite("imagen_ecualizada.png", img_eq)

hist = cv2.calcHist([img], [0], None, [256], [0,256])
hist_eq = cv2.calcHist([img_eq], [0], None, [256], [0,256])

plt.figure(figsize=(10, 7))

# Imagen original
plt.subplot(2, 2, 1); plt.imshow(img, cmap="gray")
plt.title("Imagen original"); plt.axis("off")

# Imagen ecualizada
plt.subplot(2, 2, 2); plt.imshow(img_eq, cmap="gray")
plt.title("Imagen ecualizada"); plt.axis("off")

# Histograma original
plt.subplot(2, 2, 3); plt.plot(hist)
plt.title("Histograma original")
plt.xlabel("Nivel de intensidad")
plt.ylabel("Número de píxeles")
plt.xlim([0, 255])

# Histograma ecualizado
plt.subplot(2, 2, 4); plt.plot(hist_eq)
plt.title("Histograma ecualizado")
plt.xlabel("Nivel de intensidad")
plt.ylabel("Número de píxeles")
plt.xlim([0, 255])

plt.tight_layout()
plt.show()
```

Figura 73. Código en Python del Ejemplo 22: ecualización de una imagen a escala de grises.

La Figura 74 presenta el efecto de aplicar la ecualización de histograma sobre una imagen oscura, correspondiente a un paisaje de lago. A simple vista, podemos observar que la imagen original (superior izquierda) presenta niveles de intensidad muy bajos, lo que dificulta distinguir detalles en elementos como el árbol, el agua y las montañas del fondo. Esta situación también se refleja en el histograma original (inferior izquierda), cuya distribución se concentra principalmente en valores bajos de intensidad.

Después de aplicar la ecualización, la imagen presenta un mayor contraste y los detalles resultan más fáciles de observar. Esto se debe a que las intensidades se redistribuyeron y se tienen amplitudes en valores más altos de nivel de intensidad.

Una particularidad de los histogramas ecualizados es la aparición de “picos” en algunos niveles de intensidad. Por esta razón, es común que el histograma pase de una distribución suave a una distribución más irregular, con cambios abruptos entre niveles consecutivos.

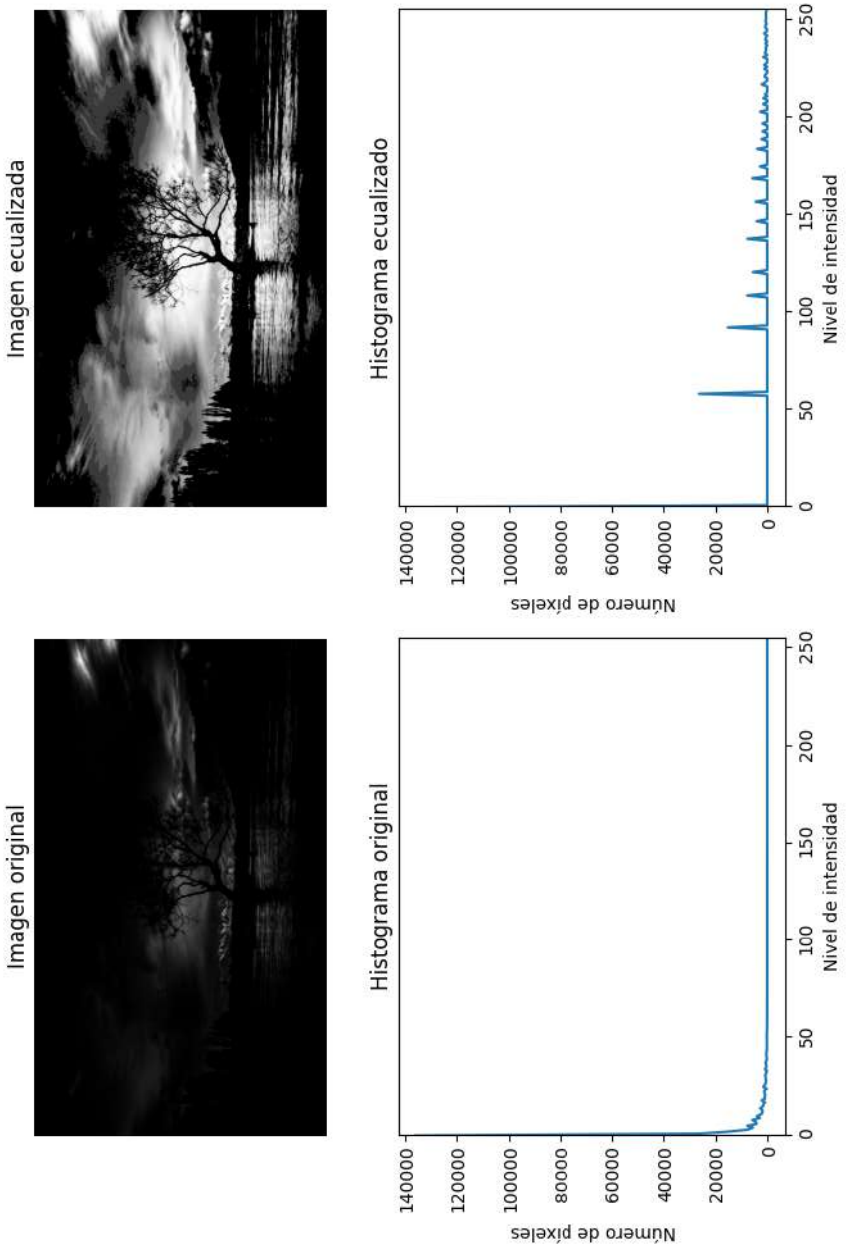


Figura 74. Ecualización de imagen a escala de grises (Ejemplo 22): imagen original vs. ecualizada (superior); histograma imagen original vs. histograma imagen ecualizada (inferior).

Ejemplo 23. Ecualización de imagen a color y modelo HSV

En el ejemplo anterior aplicamos la ecualización sobre una imagen en escala de grises. Sin embargo, en muchas aplicaciones resulta más común trabajar con imágenes a color. Y entonces, nos preguntamos:

¿Es posible aplicar la ecualización directamente sobre una imagen RGB?

La respuesta es sí, pero no suele ser la mejor alternativa. Aunque técnicamente es posible ecualizar cada canal de color por separado, esta práctica puede introducir cambios artificiales en la apariencia de la imagen. En consecuencia, los colores originales pueden alterarse y el resultado final puede verse poco natural.

Para evitar este inconveniente, en la práctica utilizanos el modelo de color **HSV (Hue, Saturation, Value)**, donde:

- **H (Hue):** representa el tono o color.
- **S (Saturation):** representa la saturación o pureza del color.
- **V (Value):** representa el brillo o intensidad.

La principal ventaja de este modelo es que separa la información de color de la información de brillo. De esta forma, podemos aplicar la ecualización únicamente sobre el canal **V**, mejorando el contraste de la imagen sin modificar significativamente sus colores originales.

La Figura 75 ilustra el proceso de ecualización de una imagen a color utilizando el modelo HSV.

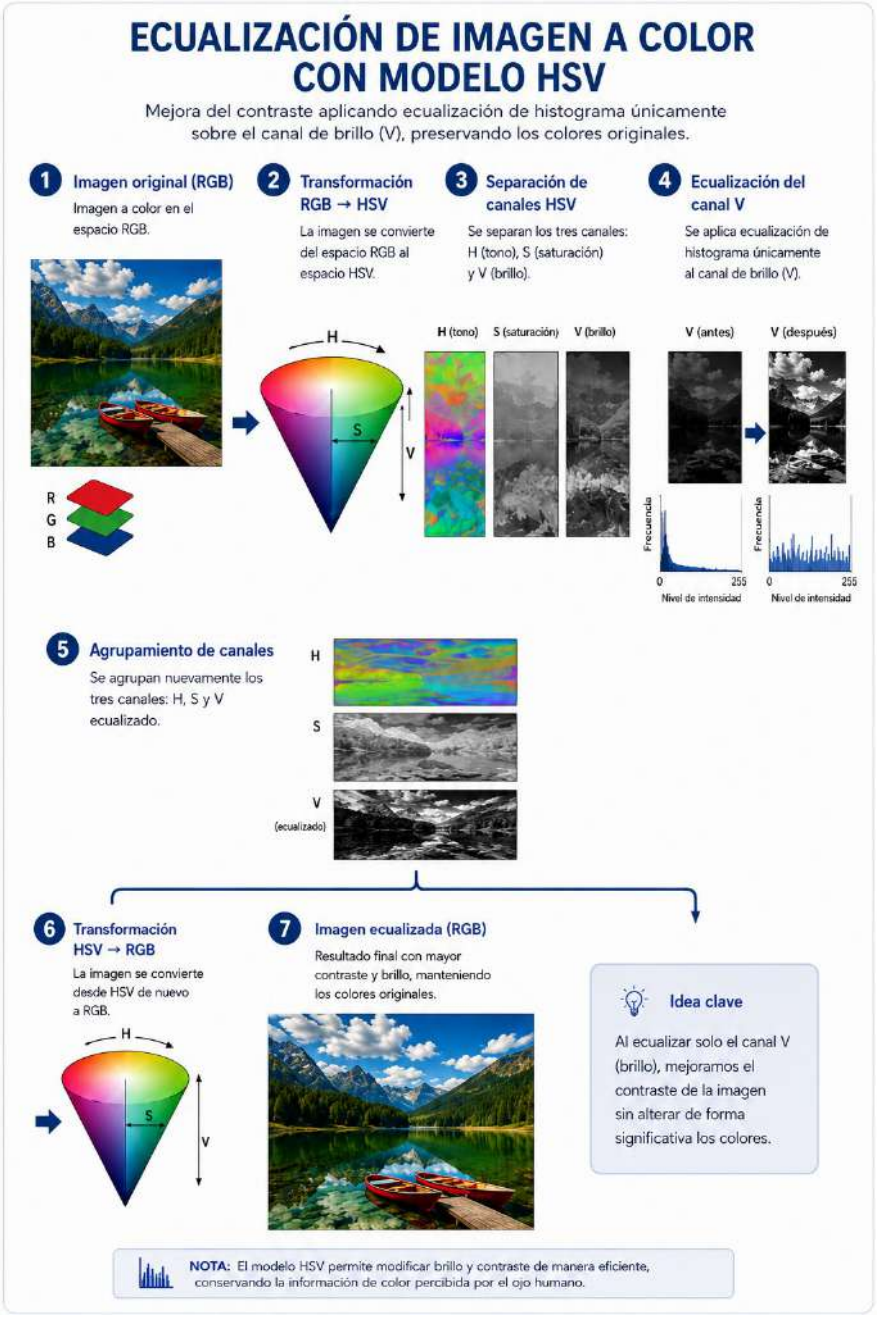


Figura 75. Proceso de ecualización de una imagen a color utilizando el modelo de color HSV.

A continuación, aplicaremos la ecualización sobre el canal V y compararemos el resultado con la imagen original.

```
import cv2
import matplotlib.pyplot as plt

imagen_bgr = cv2.imread("imagen_color.png") # Leer imagen en color
imagen_rgb = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2RGB) # Convertir de BGR a RGB
imagen_hsv = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2HSV) # Convertir de BGR a HSV
H, S, V = cv2.split(imagen_hsv) # Separar canales HSV
V_eq = cv2.equalizeHist(V) # Ecualizar únicamente el canal V
imagen_hsv_eq = cv2.merge([H, S, V_eq]) # Reconstruir imagen HSV
imagen_eq_rgb = cv2.cvtColor(imagen_hsv_eq, cv2.COLOR_HSV2RGB) # Convertir a RGB

# Histograma del canal 0 (R)
hist_original = cv2.calcHist([imagen_rgb], [0], None, [256], [0, 256])
hist_ecualizada = cv2.calcHist([imagen_eq_rgb], [0], None, [256], [0, 256])
plt.figure(figsize=(10, 8))

# Imagen original
plt.subplot(2, 2, 1);plt.imshow(imagen_rgb)
plt.title("Imagen original"); plt.axis("off")

# Imagen ecualizada
plt.subplot(2, 2, 2); plt.imshow(imagen_eq_rgb)
plt.title("Imagen ecualizada (HSV)"); plt.axis("off")

# Histograma original
plt.subplot(2, 2, 3); plt.plot(hist_original)
plt.title("Histograma original (Canal 0)"); plt.xlim([0, 256])
plt.xlabel("Nivel de intensidad"); plt.ylabel("Frecuencia")

# Histograma ecualizado
plt.subplot(2, 2, 4); plt.plot(hist_ecualizada)
plt.title("Histograma ecualizado (Canal 0)");plt.xlim([0, 256])
plt.xlabel("Nivel de intensidad"); plt.ylabel("Frecuencia")
```

Figura 76. Código en Python del Ejemplo 23: ecualización de imagen a color.

Obteniendo los siguientes resultados:

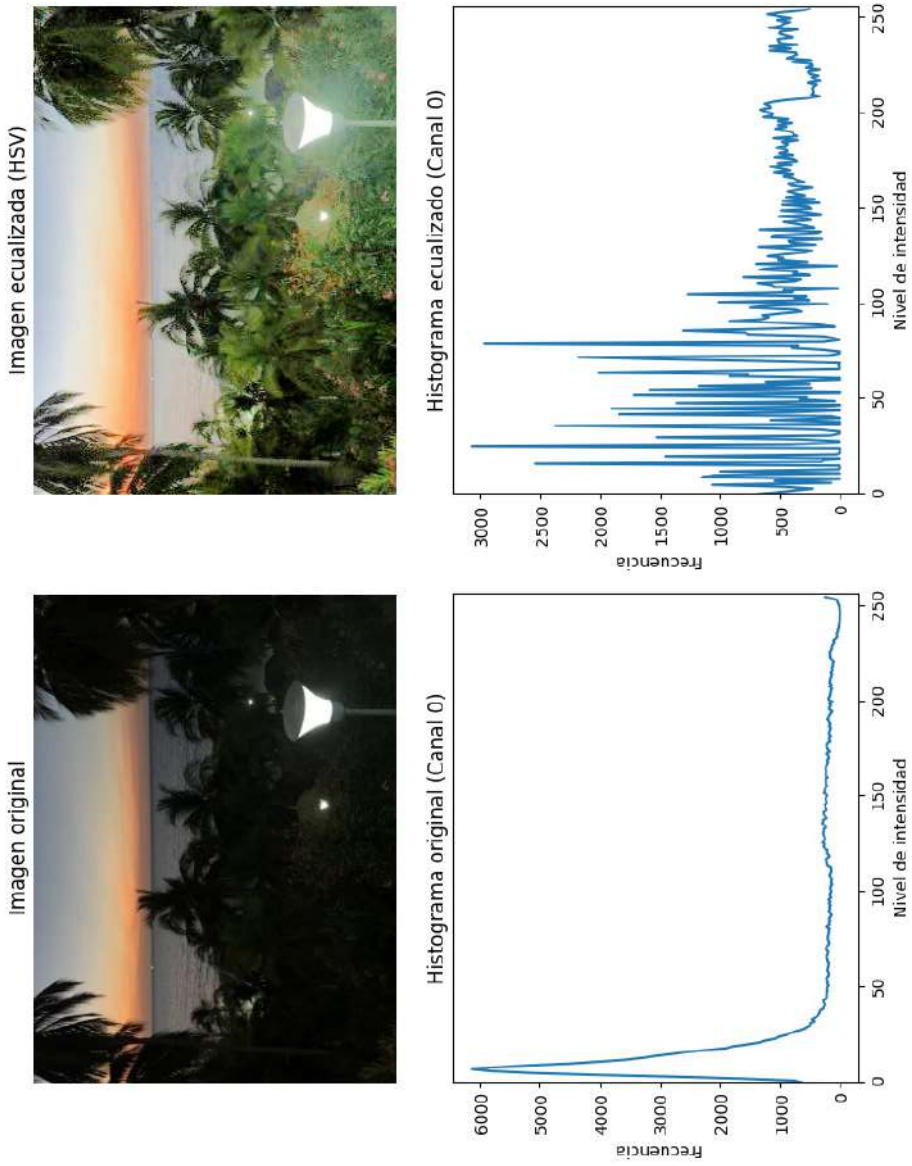


Figura 77. Ecualización de imagen a color (Ejemplo 23): imagen original vs. ecualizada (superior); histograma imagen original vs. histograma imagen ecualizada (inferior).

Como podemos observar, la imagen ecualizada presenta un mayor contraste, permitiendo apreciar con mayor claridad detalles que en la imagen original pasaban prácticamente desapercibidos, especialmente en las zonas más oscuras **(la puesta de sol ahora se ve mucho más hermosa)**. Esto se logra al redistribuir los niveles de intensidad del brillo de la imagen, aprovechando mejor el rango dinámico disponible.

Por otro lado, en la parte inferior comparamos el histograma del **canal R** antes y después de la ecualización. Podemos observar que el histograma original tiene una forma relativamente suave, con una alta concentración de píxeles en los niveles bajos de intensidad, característica de una imagen oscura. Después de la ecualización, el histograma cambia de forma y aparecen numerosos picos. Esto es completamente normal. La ecualización no busca generar una curva uniforme, sino redistribuir los niveles de intensidad para mejorar el contraste de la imagen. Por esta razón, algunos niveles de intensidad dejan de utilizarse, mientras que otros concentran un mayor número de píxeles, dando lugar a ese patrón de picos que ahora observamos.

4.4 Detección de bordes y segmentación

Hasta este momento hemos trabajado con técnicas orientadas a mejorar la apariencia de una imagen. Sin embargo, en muchas aplicaciones el objetivo no es únicamente obtener una imagen más agradable a la vista (por ejemplo, más clara), sino extraer información útil de ella. Pensemos en un vehículo autónomo que debe identificar el borde de la carretera, un sistema médico que necesita delimitar un órgano en una radiografía o un robot industrial que debe reconocer la posición de una pieza antes de manipularla.

Para ello aparecen dos conceptos fundamentales del procesamiento digital de imágenes: **la detección de bordes** y **la segmentación**. Aunque ambos buscan facilitar la interpretación de una imagen, sus objetivos son diferentes. La detección de bordes intenta localizar los cambios bruscos de intensidad que suelen indicar el límite entre dos regiones, mientras que la segmentación busca agrupar píxeles con características similares para separar los diferentes objetos presentes en la escena.

La siguiente figura presenta una breve comparación entre ambos conceptos.

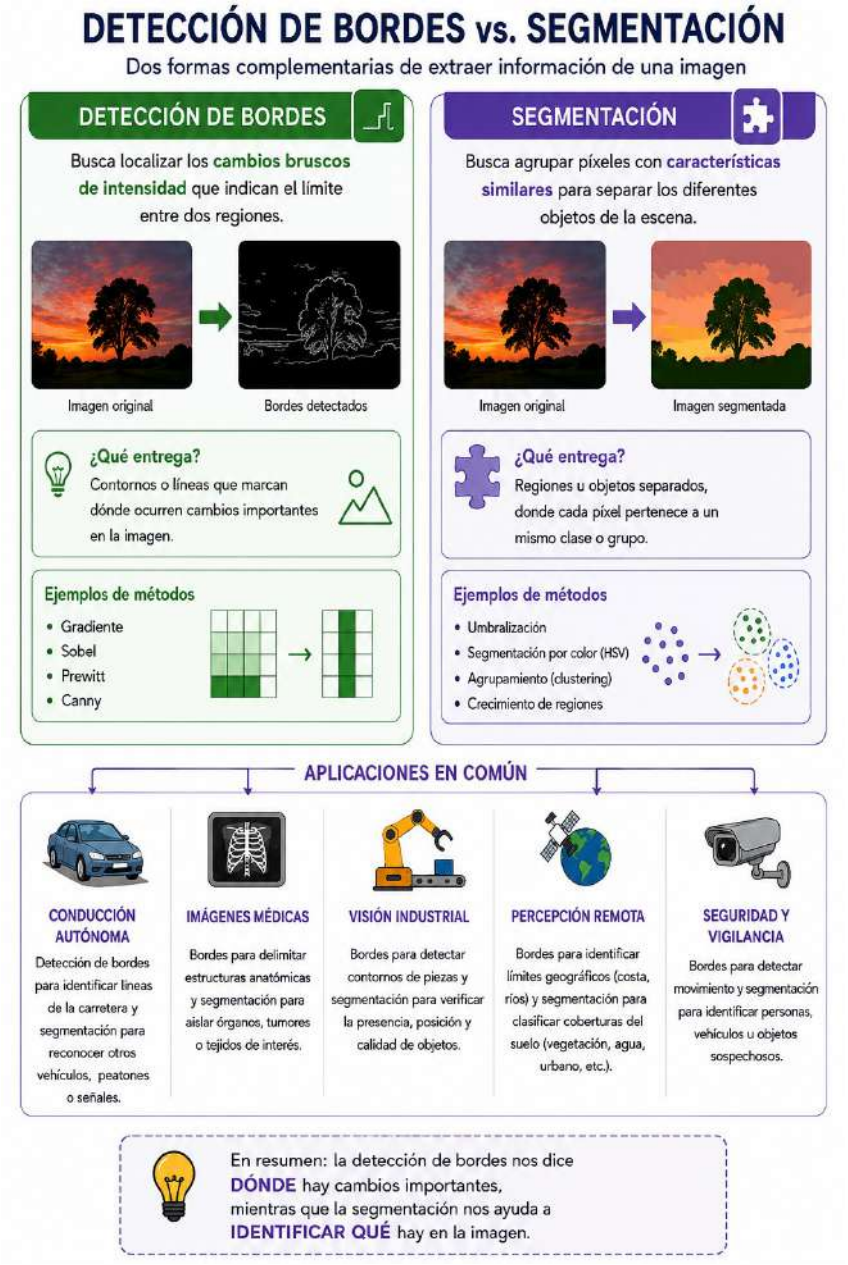


Figura 78. Comparación entre detección de bordes y segmentación.

Ahora, imagina que tenemos una fotografía como la que aparece en

la figura anterior: un árbol durante la puesta de sol. Si el objetivo es conocer la silueta del árbol, lo más importante será identificar dónde cambia bruscamente la intensidad entre el árbol y el cielo; es decir, detectar sus bordes. En cambio, si el objetivo es separar completamente el árbol del fondo para analizarlo o medir su tamaño, será necesario segmentar la imagen, asignando cada píxel al objeto o al fondo según sus características.

4.4.1 Detección de bordes

Una forma sencilla de identificar un borde consiste en comparar la intensidad de píxeles vecinos. Cuando la diferencia entre ellos es grande, es probable que exista un borde. Para ello se utilizan diferentes **máscaras** o **kernels**, los cuales detectan estos cambios mediante una convolución entre el kernel y la imagen (estrictamente, una correlación). Cada kernel está diseñado para resaltar cambios de intensidad en una dirección específica, como la horizontal, la vertical o la diagonal. La Figura 79 presenta algunos de los kernels clásicos utilizados para la detección de bordes.

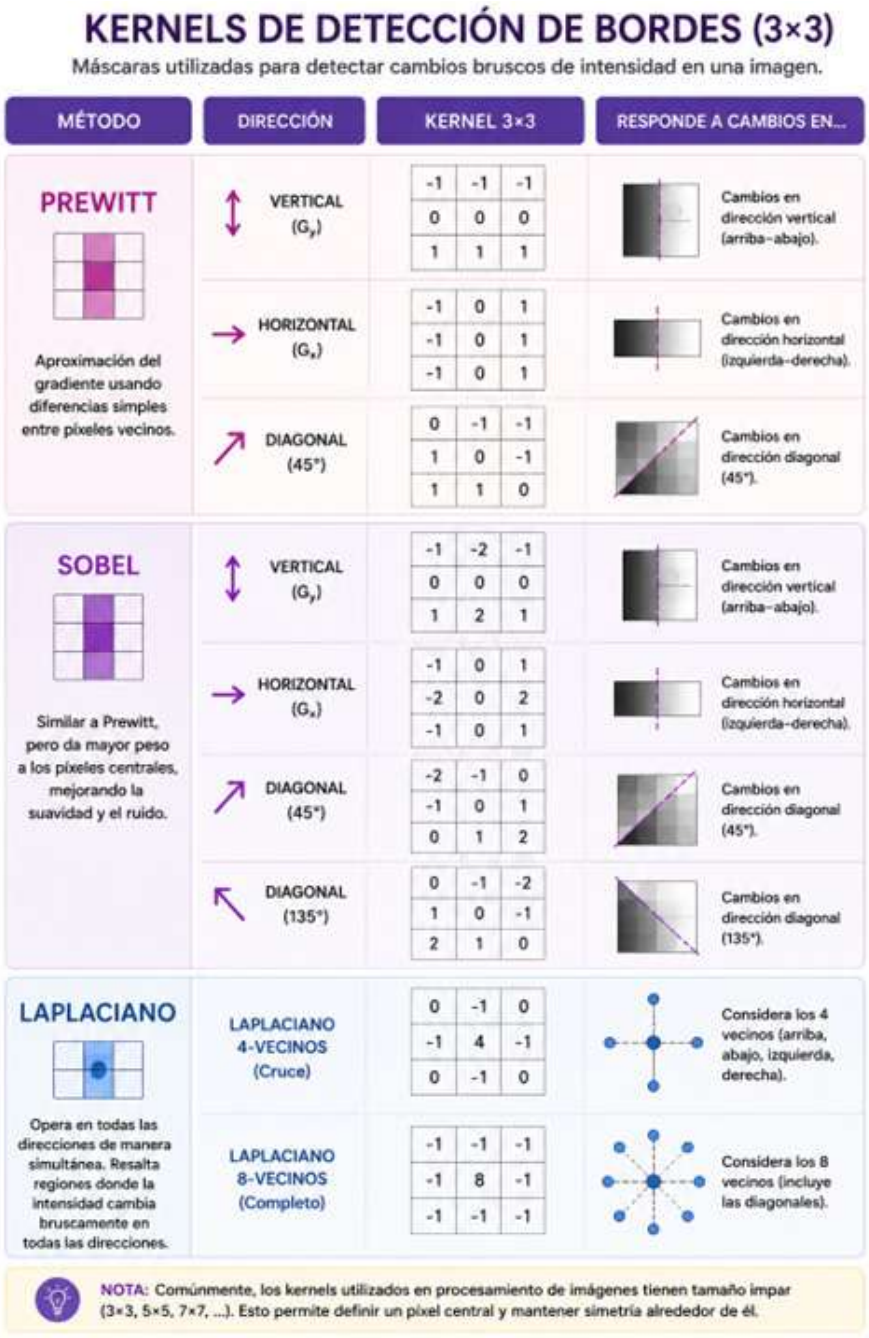


Figura 79. Kernels clásicos para detección de bordes: prewitt, sobel y laplaciano.

Los operadores **Prewitt** y **Sobel** permiten detectar cambios de intensidad en direcciones específicas, como la horizontal, la vertical o la diagonal. **Por ejemplo, para detectar bordes horizontales se utiliza un kernel compuesto por una fila de valores negativos, seguida de una fila de ceros y, finalmente, una fila de valores positivos.** La principal diferencia entre ellos es que Sobel asigna un mayor peso a los píxeles centrales, lo que lo hace ligeramente más robusto frente al ruido. Por su parte, el **operador Laplaciano** no detecta una dirección particular, sino que responde a cambios de intensidad en todas las direcciones, resaltando los contornos independientemente de su orientación.

¿Pero qué tienen en común los diferentes kernels?

Todos presentan un cambio de polaridad entre sus coeficientes y una suma total igual a cero, lo que les permite responder a cambios bruscos de intensidad. Sin embargo, en aplicaciones reales no suele aplicarse un único *kernel* de forma aislada. En su lugar, es común utilizar algoritmos más robustos, como el **método Canny**, el cual combina varias etapas de procesamiento para obtener bordes más precisos y continuos, reduciendo además la influencia del ruido. La Figura 80 resume su funcionamiento general.

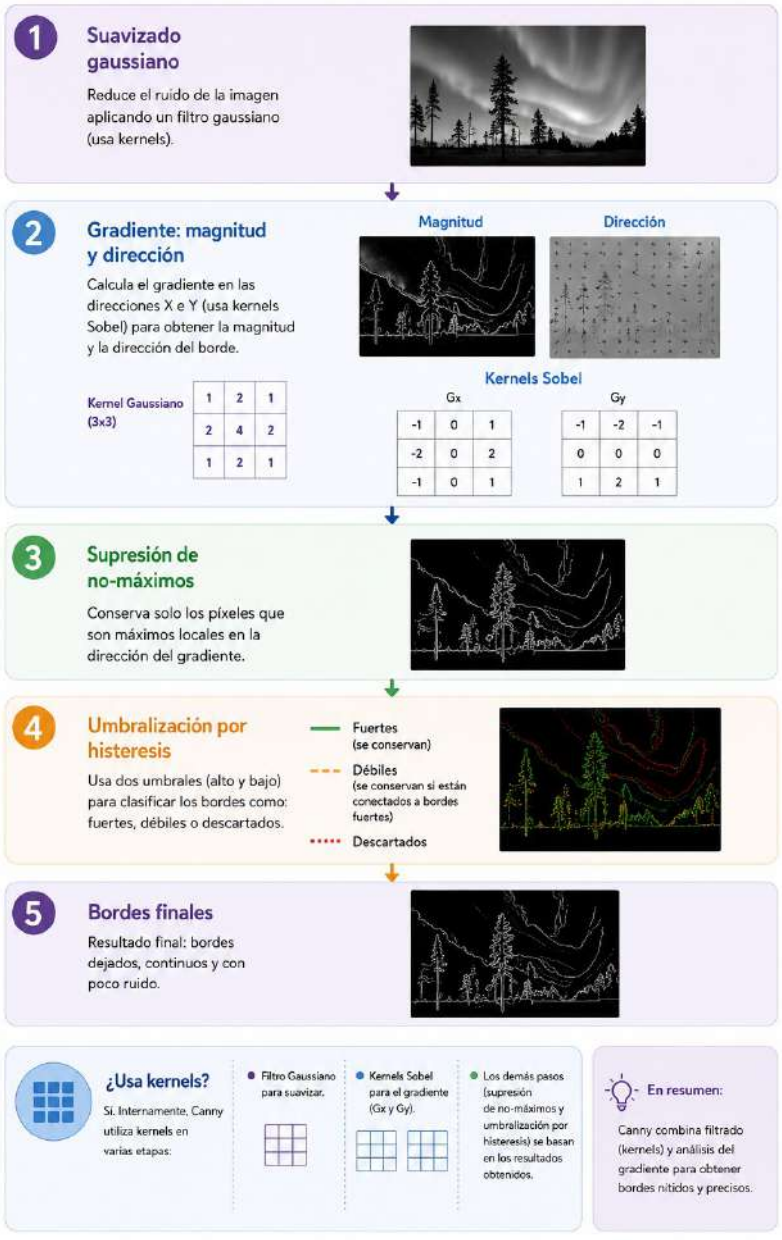


Figura 80. Algoritmo Canny: suavizado gaussiano, gradiente de magnitud y dirección, supresión de no-máximos, umbralización por histéresis y bordes finales.

Como podemos observar, el algoritmo **Canny** no se limita a aplicar un único kernel sobre la imagen. En su lugar, realiza una secuencia de etapas que incluyen el suavizado de la imagen, el cálculo del gradiente mediante **kernels** (como los operadores **Sobel** en las direcciones horizontal y vertical), la supresión de respuestas no máximas y la aplicación de umbrales para conservar únicamente los bordes más relevantes. El suavizado inicial se realiza mediante un kernel gaussiano, el cual estudiaremos al finalizar este capítulo.

Ejemplo 24. Detección de bordes con el algoritmo Canny

Vamos ahora a aplicar el algoritmo Canny para la detección de bordes en una imagen a color. En este ejemplo generaremos tres imágenes de salida, ya que utilizaremos tres pares de umbrales dentro del algoritmo: uno para una alta sensibilidad a los bordes, otro para una sensibilidad media y un último para una sensibilidad baja. A continuación, se presenta el código en Python.

```
import cv2
import matplotlib.pyplot as plt

imagen_bgr = cv2.imread("aurora_boreal.png") # Leer imagen
imagen_rgb = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2RGB) # Convertir a RGB
imagen_gray = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2GRAY) # Escala de grises

canny1 = cv2.Canny(imagen_gray, 20, 60) # Alta sensibilidad
canny2 = cv2.Canny(imagen_gray, 40, 120) # Sensibilidad media
canny3 = cv2.Canny(imagen_gray, 100, 240) # Baja sensibilidad

plt.figure(figsize=(10,8))

plt.subplot(2,2,1); plt.imshow(imagen_rgb)
plt.title("Imagen original"); plt.axis("off")

plt.subplot(2,2,2); plt.imshow(canny1, cmap="gray")
plt.title("Canny (20,60)"); plt.axis("off")

plt.subplot(2,2,3); plt.imshow(canny2, cmap="gray")
plt.title("Canny (40,120)"); plt.axis("off")

plt.subplot(2,2,4); plt.imshow(canny3, cmap="gray")
plt.title("Canny (100,240)"); plt.axis("off")

plt.tight_layout(); plt.show()
```

Figura 81. Segmentación por color utilizando el modelo HSV.

Obteniendo:

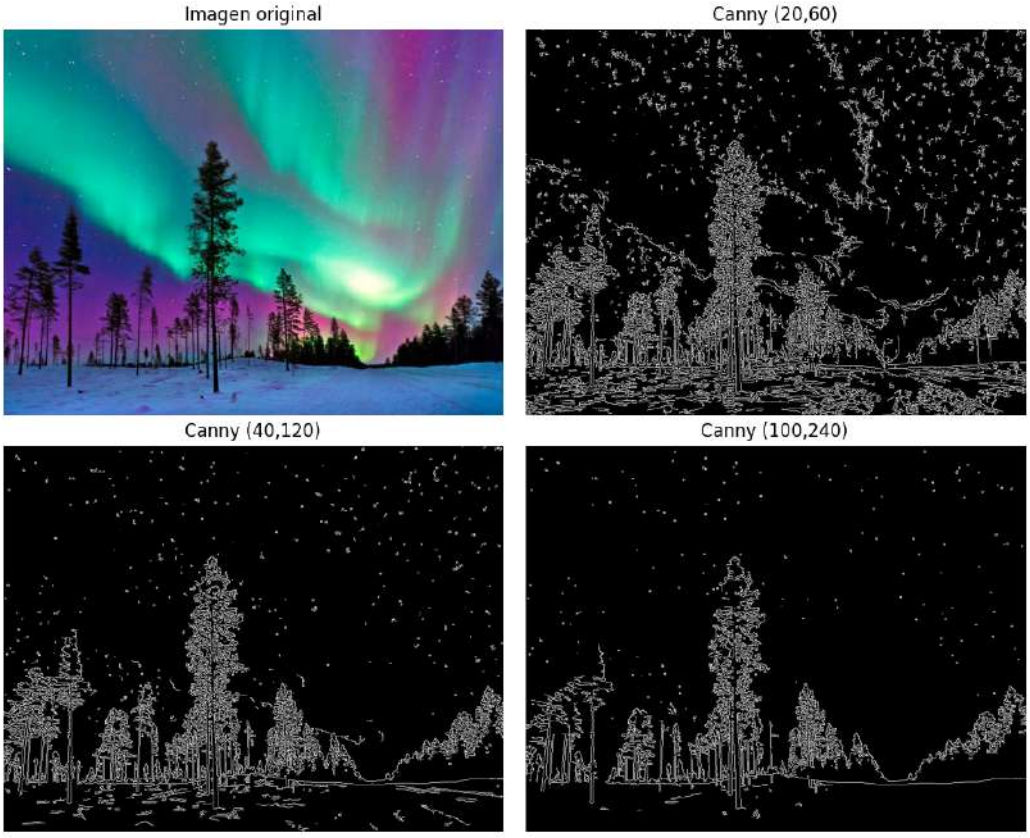


Figura 82. Imagen original (aurora boreal) y tres imágenes de detección de bordes utilizando el algoritmo Canny.

En la primera imagen de detección de bordes (superior derecha), correspondiente a los umbrales **20** y **60**, se observa una alta sensibilidad del algoritmo. Como resultado, se detectan numerosos bordes en la zona del bosque y también múltiples variaciones de intensidad presentes en la aurora boreal y en las estrellas, generando una imagen con una gran cantidad de detalles.

En la segunda imagen (inferior izquierda), obtenida con los umbrales **40** y **120**, disminuye el número de bordes detectados, conservándose principalmente los contornos más representativos del bosque y del horizonte. Finalmente, en la tercera imagen (inferior derecha), utilizando los umbrales **100** y **240**, únicamente

permanecen los bordes con mayor contraste, desapareciendo casi por completo las estructuras asociadas a la aurora boreal.

Como podemos observar, el resultado depende en gran medida de los dos umbrales utilizados por el algoritmo Canny: el **umbral inferior**, que determina la sensibilidad para detectar bordes débiles, y el **umbral superior**, que establece el nivel mínimo para considerar un borde como fuerte. Por esta razón, la selección de estos parámetros debe realizarse de acuerdo con las características de la imagen y el nivel de detalle que se desea conservar.

Ahora, pasaremos al concepto de segmentación.

4.4.2 Segmentación de color con el modelo HSV

Por otro lado, segmentar consiste en asignar cada píxel a una determinada región, facilitando así el análisis de los diferentes elementos presentes en la escena. Pensemos nuevamente en la fotografía del árbol durante la puesta de sol. Detectar los bordes nos permite conocer la silueta del árbol, mientras que segmentarlo significa separar todos los píxeles que pertenecen al árbol del resto de la imagen, obteniendo una nueva imagen donde el árbol aparece completamente aislado del fondo.

Existen diferentes criterios para realizar una segmentación. En algunos casos se utiliza la intensidad de los píxeles; en otros, el color, la textura o incluso información de profundidad. En este capítulo nos concentraremos en la **segmentación por color**, una de las técnicas más utilizadas por su simplicidad y eficiencia. Para lo cual, trabajamos con el modelo de color HSV (Figura 83).

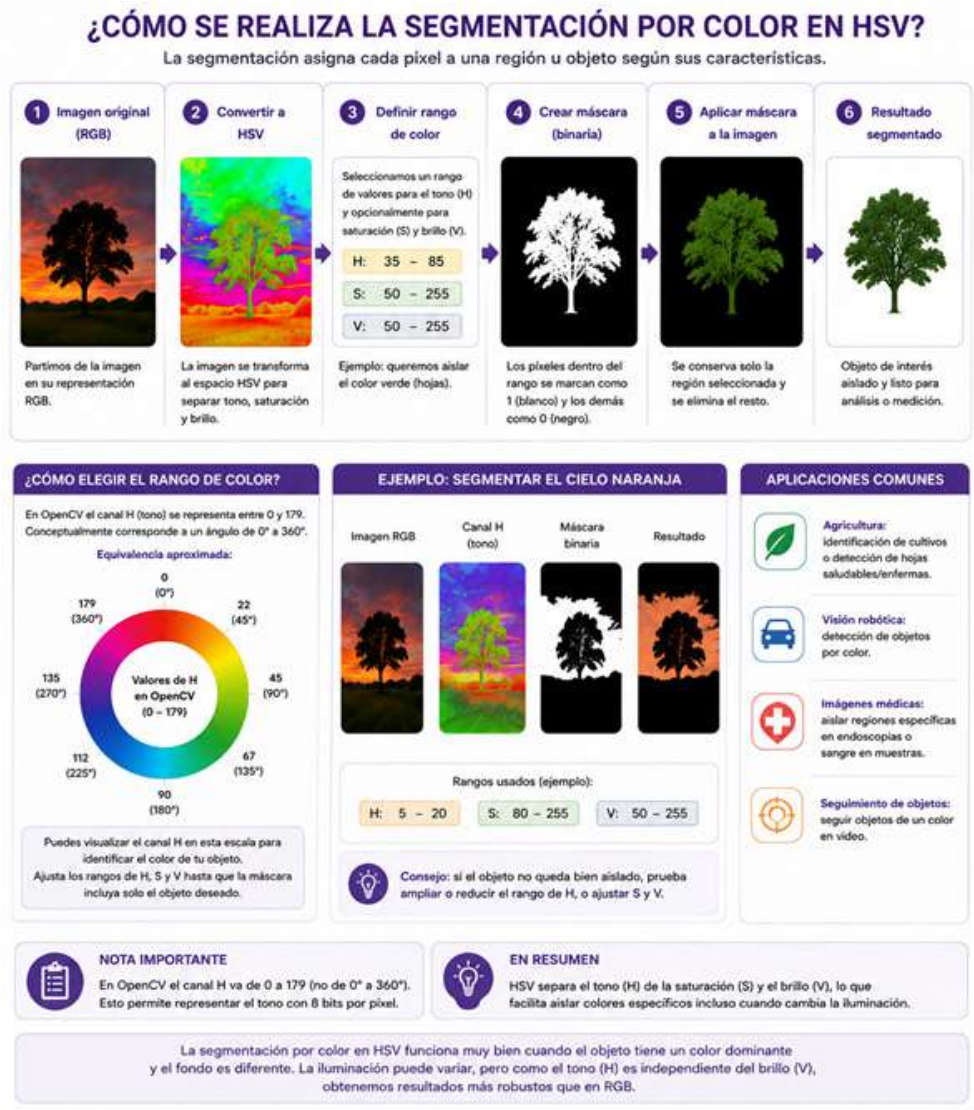


Figura 83. Segmentación por color utilizando el modelo HSV.

Supongamos que queremos segmentar el cielo con las tonalidades naranjas presentes durante la puesta de sol. Para ello definimos un rango de valores en el espacio HSV. El parámetro H (Hue) corresponde al tono o color y es el más importante en la segmentación por color. En este caso se selecciona el intervalo H = [5, 20], el cual abarca las tonalidades naranjas de la imagen (véase

la asignación de colores en la gráfica radial de la Figura 80). Los píxeles cuyo tono se encuentre fuera de este intervalo serán descartados.

Como recomendación, los valores mínimos de S (Saturation) y V (Value) no deberían ser demasiado bajos, ya que esto podría incluir píxeles poco saturados o con muy baja iluminación que no pertenecen al objeto de interés. En la práctica, estos parámetros los ajustamos de forma empírica según la imagen y las condiciones de iluminación.

En resumen, el parámetro **H** determina **qué color** se desea segmentar, mientras que **S** y **V** actúan como filtros adicionales que permiten eliminar regiones con poca saturación o escasa iluminación. Por esta razón, en la práctica normalmente ajustamos primero el rango de **H** y, posteriormente, refinamos los valores de **S** y **V** hasta obtener una segmentación adecuada.

Ejemplo 25. Segmentación de imagen con modelo HSV

En el **Ejemplo 24** vimos que el algoritmo **Canny** permite detectar bordes a partir de cambios de intensidad, resaltando principalmente los contornos de la zona del bosque. Sin embargo, en este ejemplo el propósito es completamente diferente: identificar la región de la imagen correspondiente a la **aurora boreal**, la cual presenta predominantemente tonalidades **verde turquesa (verde cian)**. Para ello utilizaremos el modelo **HSV**, definiendo un rango de valores para los parámetros **H**, **S** y **V** que permita segmentar dichas tonalidades.

Utilizaremos el siguiente código en Python:

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt

imagen_bgr = cv2.imread("aurora_boreal.png") # Leer imagen
imagen_rgb = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2RGB) # Convertir a RGB
imagen_hsv = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2HSV) # Convertir a HSV

lim_inf = np.array([35, 60, 50]) # Límite inferior HSV
lim_sup = np.array([90, 255, 255]) # Límite superior HSV

mascara = cv2.inRange(imagen_hsv, lim_inf, lim_sup) # Crear máscara binaria
segmentada = cv2.bitwise_and(imagen_rgb, imagen_rgb, mask=mascara) # Aplicar máscara

plt.figure(figsize=(10, 8))
plt.subplot(2, 2, 1); plt.imshow(imagen_rgb)
plt.title("Imagen original"); plt.axis("off")
plt.subplot(2, 2, 2); plt.imshow(imagen_hsv[:, :, 0], cmap="hsv")
plt.title("Canal H"); plt.axis("off")
plt.subplot(2, 2, 3); plt.imshow(mascara, cmap="gray")
plt.title("Máscara HSV"); plt.axis("off")
plt.subplot(2, 2, 4); plt.imshow(segmentada)
plt.title("Aurora segmentada"); plt.axis("off")
plt.tight_layout(); plt.show()
```

Figura 84. Código en Python del Ejemplo 25: segmentación de imagen con HSV.

Antes de analizar los valores utilizados, observemos cómo se construyen los rangos de **H**, **S** y **V**. En el código definimos dos vectores: un **límite inferior** (lim_inf) y un **límite superior** (lim_sup). El primer elemento de cada vector corresponde al parámetro H (Hue), el segundo a S (Saturation) y el tercero a V (Value). Por lo tanto, el rango de cada parámetro se obtiene tomando el valor correspondiente del límite inferior y del límite superior. En este ejemplo se tiene:

- H = [35, 90]
- S = [60, 255]
- V = [50, 255]

Es decir, un píxel será seleccionado únicamente si su valor de **H** está entre **35 y 90**, su valor de **S** entre **60 y 255** y su valor de **V** entre **50 y 255**. Si alguno de estos tres parámetros queda por fuera de su intervalo, el píxel no hará parte de la segmentación.

En este ejemplo, el intervalo **H = [35, 90]** fue seleccionado porque corresponde a las tonalidades **verde turquesa** presentes en la aurora boreal. Recordemos que, en OpenCV, el canal **H** toma valores entre **0 y 179**, por lo que este rango abarca la mayor parte de las tonalidades verdes observadas en la imagen. Por su parte, el rango **S = [60, 255]** descarta píxeles con muy poca saturación, mientras que **V = [50, 255]** elimina regiones demasiado oscuras. De esta forma, la segmentación se concentra principalmente en las zonas de la aurora con un color verde bien definido.

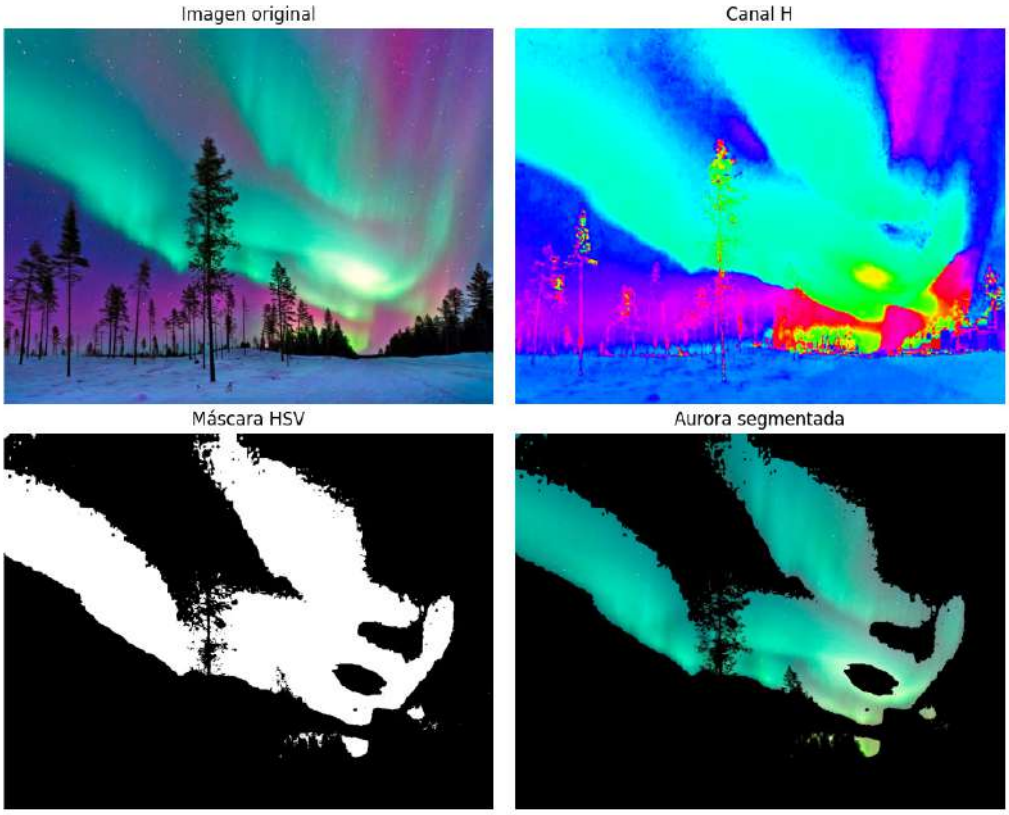


Figura 85. Imagen segmentada enfocada en la zona de aurora boreal.

La máscara binaria actúa como un filtro sobre la imagen original. Conceptualmente, puede interpretarse como una multiplicación píxel a píxel, donde los píxeles blancos de la máscara conservan el color original de la imagen, mientras que los píxeles negros se convierten en negro. Por lo cual, para nuestro ejemplo, solamente las zonas en blanco de la máscara HSV (inferior izquierda) al multiplicarse con la imagen original producen la imagen segmentada en la que aparece en negro la zona descartada y en verde turquesa la zona de interés.

Pero qué pasaría si seleccionamos de forma equivocada el rango para H. Vamos a ver el efecto con los siguientes umbrales:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

imagen_bgr = cv2.imread("aurora_boreal.png") # Leer imagen
imagen_rgb = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2RGB) # Convertir a RGB
imagen_hsv = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2HSV) # Convertir a HSV

# Rangos de H diferentes, manteniendo S y V constantes
lim_inf_1 = np.array([20, 60, 50]); lim_sup_1 = np.array([40, 255, 255])
lim_inf_2 = np.array([35, 60, 50]); lim_sup_2 = np.array([90, 255, 255])
lim_inf_3 = np.array([90, 60, 50]); lim_sup_3 = np.array([130, 255, 255])

mascara_1 = cv2.inRange(imagen_hsv, lim_inf_1, lim_sup_1) # Rango H bajo
mascara_2 = cv2.inRange(imagen_hsv, lim_inf_2, lim_sup_2) # Rango H adecuado
mascara_3 = cv2.inRange(imagen_hsv, lim_inf_3, lim_sup_3) # Rango H alto

seg_1 = cv2.bitwise_and(imagen_rgb, imagen_rgb, mask=mascara_1)
seg_2 = cv2.bitwise_and(imagen_rgb, imagen_rgb, mask=mascara_2)
seg_3 = cv2.bitwise_and(imagen_rgb, imagen_rgb, mask=mascara_3)
```

Figura 86. Código en Python del Ejemplo 25: segmentación de imagen con HSV, utilizando tres umbrales diferentes.

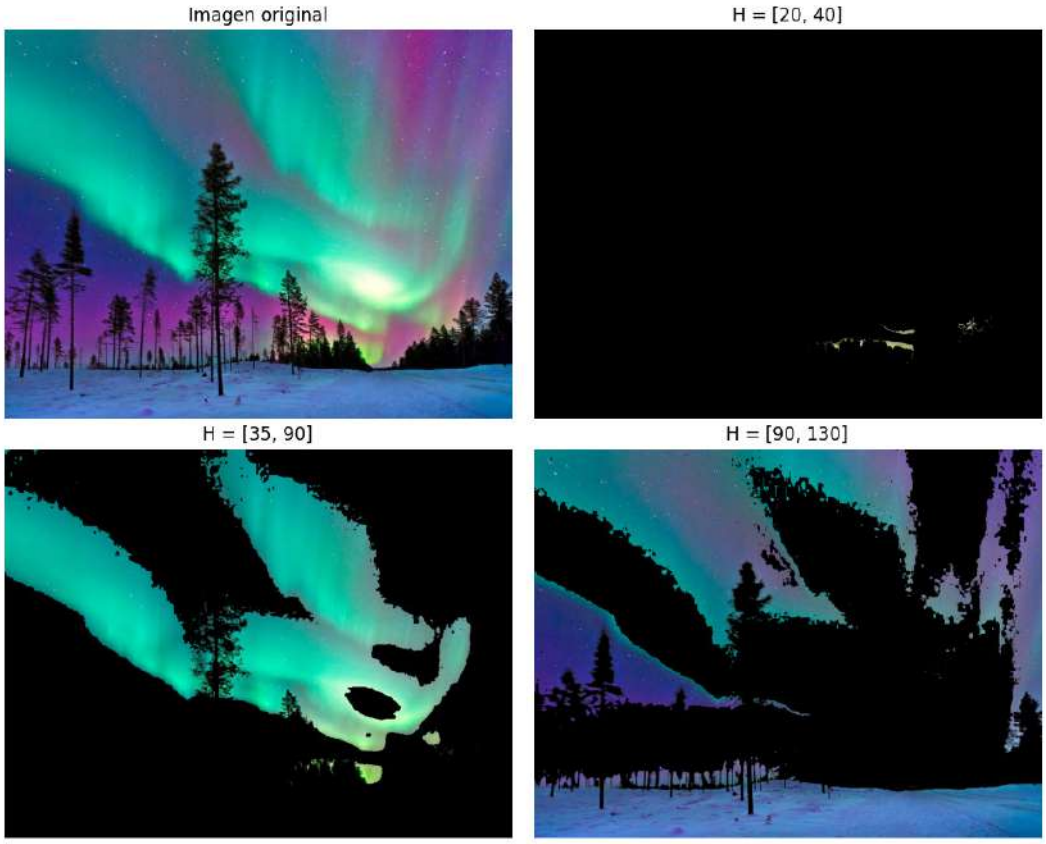


Figura 87. Imagen segmentada: comparación de resultados con tres umbrales diferentes.

La Figura 87 muestra el efecto de modificar únicamente el rango del parámetro **H (Hue)**, manteniendo constantes los valores de **S** y **V**. Como podemos observar, una selección inadecuada del rango de **H** puede alterar significativamente el resultado de la segmentación.

En la imagen superior derecha se utilizó el intervalo **H = [20, 40]**. Este rango no corresponde a las tonalidades predominantes de la aurora boreal, por lo que prácticamente toda la imagen es descartada y únicamente permanecen unas pocas regiones aisladas. En la imagen inferior izquierda se empleó el intervalo **H = [35, 90]**, el cual coincide con las tonalidades **verde turquesa**

de la aurora. Como resultado, la mayor parte de la aurora es correctamente segmentada, mientras que el resto de la escena se elimina.

Finalmente, en la imagen inferior derecha se utilizó el intervalo **H = [90, 130]**. En este caso, el algoritmo comienza a seleccionar tonalidades diferentes al verde turquesa, incorporando regiones que no pertenecen al objeto de interés y perdiendo simultáneamente parte de la aurora. Este ejemplo demuestra que la correcta selección del rango de **H** es el aspecto más importante de la segmentación por color en el espacio HSV.

Al comparar los resultados del Ejemplo 24 y del Ejemplo 25, nos damos cuenta de que la detección de bordes responde a la pregunta "¿dónde cambian los píxeles?", mientras que la segmentación responde a la pregunta "¿qué píxeles pertenecen al objeto de interés?". Por esta razón, la detección de bordes es especialmente útil para identificar contornos y formas, mientras que la segmentación permite separar regiones completas para su posterior análisis, medición o reconocimiento.

Ejemplo 26. El rojo, un caso especial en HSV

Y para finalizar la temática de segmentación por color, resumiremos algunos de los rangos de color más utilizados dentro del modelo HSV, los cuales servirán como referencia al momento de diseñar máscaras de segmentación. La Figura 88 presenta algunos de estos rangos.

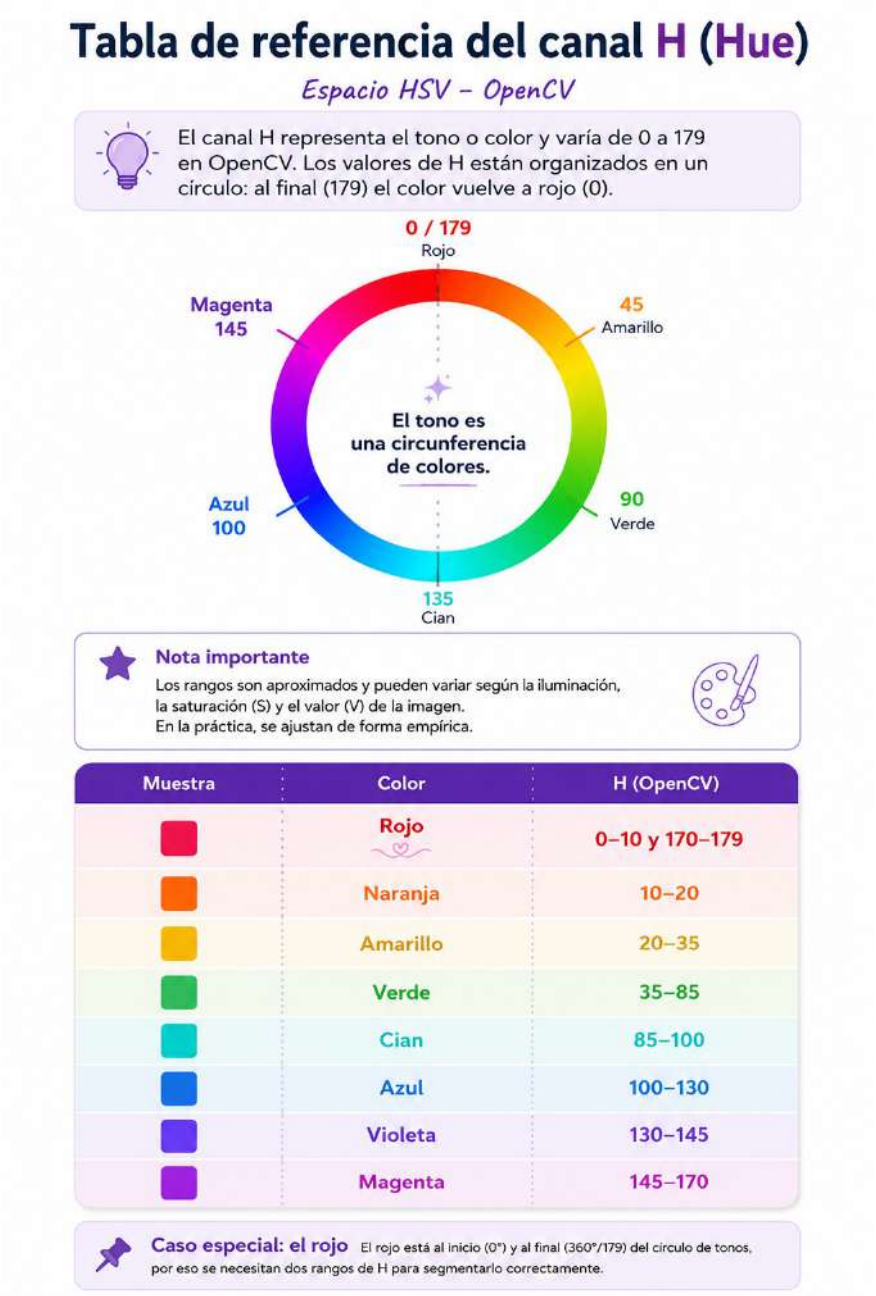


Figura 88. Rangos de referencia del canal **H (Hue)** para algunos colores en OpenCV.

Como podemos observar, el color **rojo** constituye un caso especial, ya que se encuentra ubicado tanto al inicio como al final del círculo de color. Por esta razón, cuando se desea segmentar este color es necesario generar **dos máscaras**, una para cada intervalo de **H**, y posteriormente combinarlas en una sola. El procedimiento general se presenta en la Figura 89.

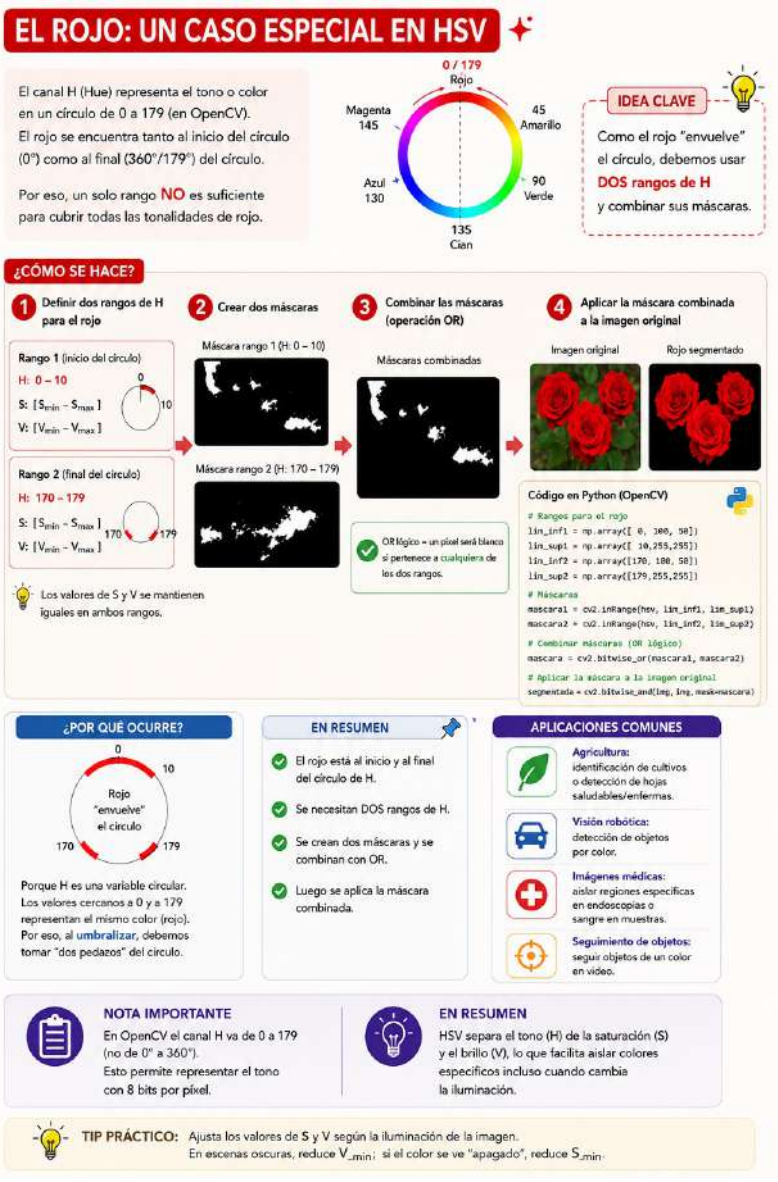


Figura 89. Caso especial de la segmentación del color rojo en HSV.

De esta manera, al fusionar ambas máscaras, es posible segmentar correctamente todas las regiones de la imagen que corresponden al color rojo.

4.5. Ruido en Imágenes y Filtrado

Hasta este momento hemos estudiado diferentes técnicas para mejorar la apariencia de una imagen, detectar sus bordes y segmentar regiones de interés. Sin embargo, en aplicaciones reales las imágenes rara vez son perfectas. Es común que durante el proceso de adquisición, transmisión o almacenamiento aparezcan alteraciones conocidas como **ruido**, las cuales pueden degradar la calidad visual de la imagen y afectar el desempeño de algoritmos como la detección de bordes o la segmentación.

En esta sección estudiaremos los principales tipos de ruido presentes en imágenes digitales y realizaremos una breve introducción a los filtros espaciales utilizados para reducir su efecto. Estos conceptos servirán como base para el caso de estudio que desarrollaremos en el siguiente capítulo.

4.5.1 Tipos de ruido

Seguramente has observado fotografías que presentan una apariencia granulada o pequeños puntos distribuidos sobre la imagen, especialmente cuando fueron tomadas en condiciones de poca iluminación. Estas alteraciones reciben el nombre de **ruido** y corresponden a modificaciones no deseadas de los valores originales de los píxeles. Su origen puede estar asociado al sensor de la cámara, las condiciones de iluminación, el proceso de transmisión o incluso la compresión y el almacenamiento de la imagen.

Desde un punto de vista matemático, una forma sencilla de representar el ruido consiste en asumir que este se suma a la imagen original. A este modelo se le conoce como **modelo aditivo de ruido** y podemos expresarlo mediante la siguiente ecuación:

$$g(x,y) = f(x,y) + n(x,y)$$

donde:

- $f(x,y)$: imagen original. Donde x,y corresponden a fila y columna, respectivamente.
- $n(x,y)$: ruido.
- $g(x,y)$: imagen observada o imagen con ruido.

En otras palabras, cada píxel de la imagen original puede verse alterado por una pequeña perturbación aleatoria, dando lugar a la imagen que finalmente observamos.

Aunque existen diferentes modelos de ruido, en este libro estudiaremos los tres más utilizados en procesamiento digital de imágenes: ruido gaussiano, ruido uniforme y ruido sal y pimienta.



Figura 90. Ruido en imagen: gaussiano, uniforme, sal y pimienta.

Como podemos observar, cada tipo de ruido modifica la imagen de una forma diferente. El **ruido gaussiano** produce una apariencia similar a una imagen envejecida o ligeramente desenfocada, donde los bordes pierden parte de su nitidez (como una fotografía impresa que guardamos durante muchos años). El ruido uniforme también afecta toda la imagen, pero genera una textura más homogénea, como si se hubiera añadido un grano fino distribuido de manera uniforme. Finalmente, el ruido sal y pimienta se caracteriza por la aparición de píxeles blancos y negros aislados (como si la imagen tuviera pecas), los cuales resultan especialmente visibles en regiones de color uniforme, como el cielo.

A simple vista podríamos pensar que todos estos tipos de ruido pueden eliminarse utilizando el mismo filtro. Sin embargo, cada uno presenta características diferentes y, por tanto, requiere una estrategia de filtrado específica. La siguiente sección introduce algunos de los filtros espaciales más utilizados para reducir el efecto del ruido.

4.5.2 Tipos de filtro

Al igual que en la detección de bordes, la mayoría de los filtros espaciales utilizan un kernel o máscara que se desplaza sobre toda la imagen realizando una convolución (estrictamente, una correlación). Sin embargo, el objetivo ahora es completamente diferente. Mientras que en la detección de bordes se buscaba resaltar los cambios bruscos de intensidad, en el filtrado el propósito es reducir el ruido, preservando en la medida de lo posible la información original de la imagen.

Esta diferencia también se refleja en los valores del kernel. Recordemos que los operadores de detección de bordes presentaban cambios de polaridad y una suma total de sus coeficientes igual a cero, ya que su objetivo era responder a las variaciones de intensidad. En cambio, los filtros espaciales tradicionales utilizan coeficientes positivos cuya suma es igual a uno, evitando modificar el nivel promedio de brillo de la imagen mientras realizan el proceso de suavizado.

La principal excepción es el filtro de mediana, el cual no utiliza una suma ponderada de los valores contenidos en el kernel. En su lugar, reemplaza el píxel central por la mediana de los valores presentes en la vecindad, razón por la cual presenta un comportamiento diferente al de los demás filtros espaciales.

En la Figura 91 se resumen las principales diferencias entre los filtros utilizados para la detección de bordes y aquellos empleados para la reducción de ruido, incluyendo como caso especial el filtro de mediana. En los dos primeros casos, el resultado se obtiene mediante una convolución (estrictamente, una correlación) entre la imagen de entrada y una máscara o kernel, el cual se desplaza por toda la imagen, normalmente de izquierda a derecha y de arriba hacia abajo.

Para evitar que la imagen de salida sea más pequeña que la imagen original, es habitual **ampliar sus bordes** antes de realizar el filtrado. Una estrategia muy utilizada consiste en añadir un borde de ceros alrededor de la imagen, aunque también existen otras técnicas de extensión de tamaño de la imagen. De esta manera, al aplicar el kernel se obtiene una imagen de salida con el mismo tamaño que la imagen de entrada.

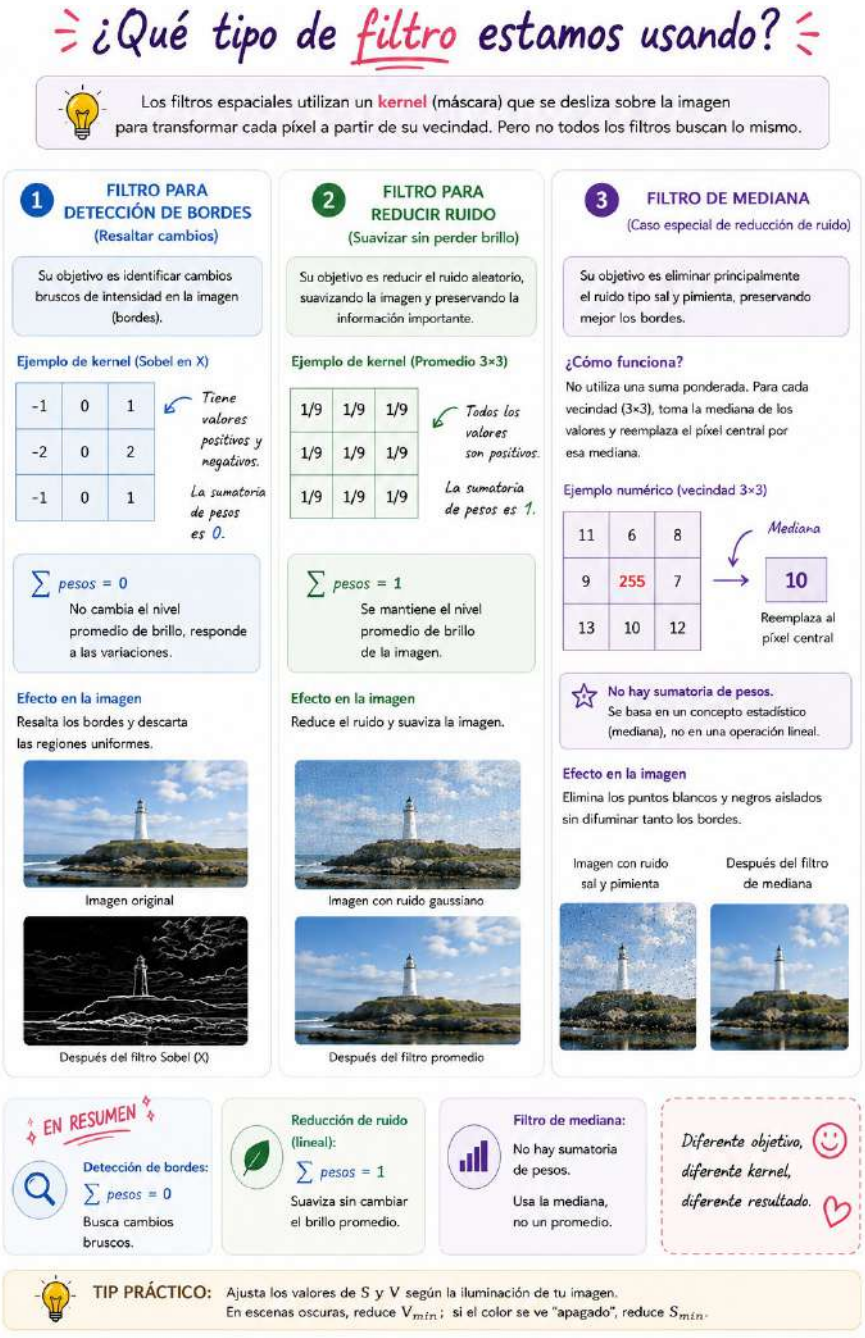


Figura 91. Filtro para detección de bordes vs. filtro para reducción de ruido.

Veamos ahora el proceso de convolución entre el kernel y la imagen.

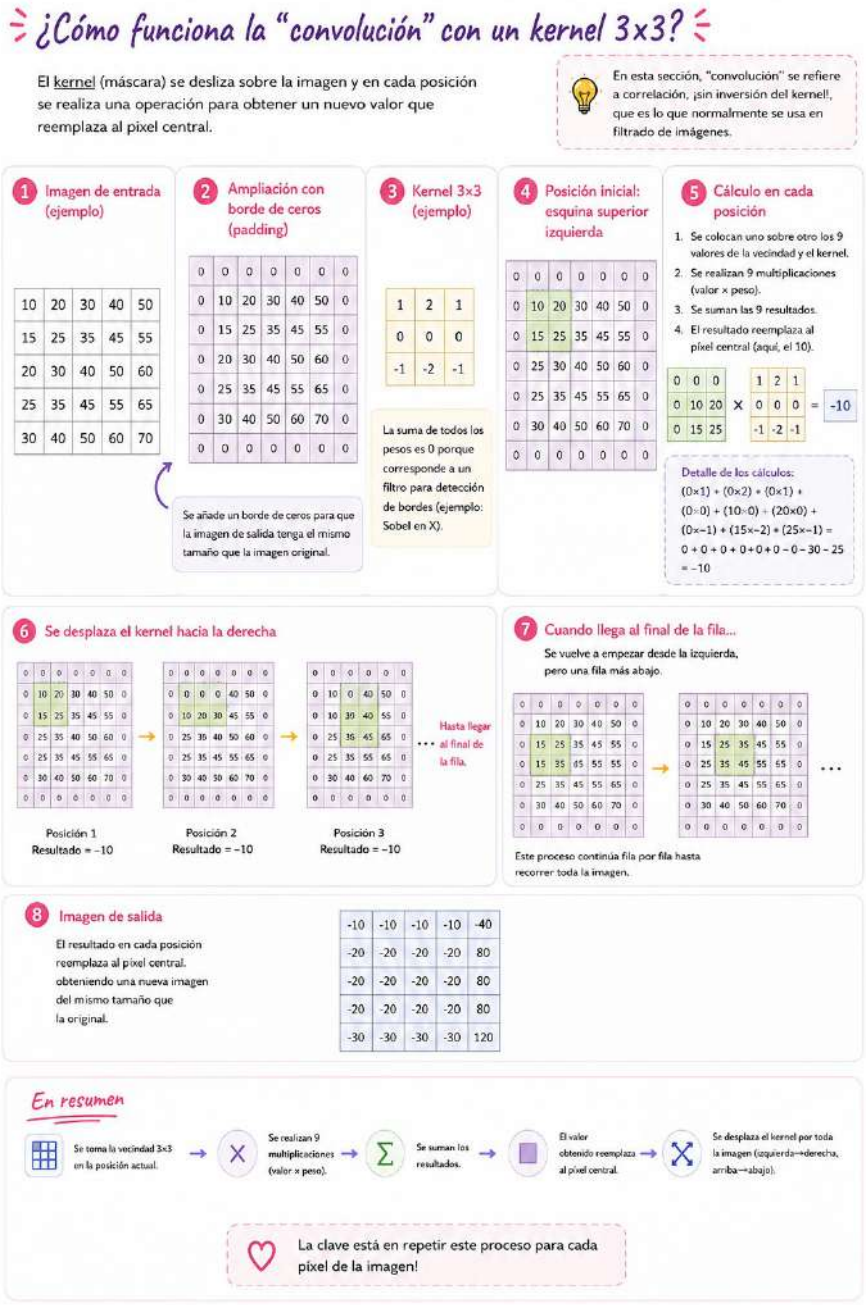


Figura 92. Explicación proceso de convolución.

Y de esta forma obtenemos la imagen de salida, la cual corresponderá a una imagen con bordes resaltados o a una imagen con ruido reducido, dependiendo del tipo de kernel utilizado. Sin embargo, dado que una imagen digital almacena valores enteros de intensidad entre 0 y 255, es posible que durante la convolución algunos resultados queden por fuera de este intervalo.

En estos casos se aplican dos reglas muy sencillas:

- Si el valor calculado para un píxel es menor que 0, se reemplaza por 0.
- Si el valor calculado para un píxel es mayor que 255, se reemplaza por 255.

Este proceso, conocido como saturación o recorte de valores (*clipping*), garantiza que todos los píxeles de la imagen de salida permanezcan dentro del rango permitido (este comportamiento es especialmente común en la detección de bordes, como se observa en la Figura 92).

Por otro lado, en el caso del filtro de mediana ocurre algo diferente. En lugar de utilizar un kernel con coeficientes para realizar multiplicaciones y sumas, se emplea una especie de ventana (o kernel sin pesos) que también se desplaza por toda la imagen. Sin embargo, su propósito no es realizar una convolución, sino encontrar el valor de la mediana de los 9 píxeles contenidos dentro de la ventana de 3×3. Una vez calculada la mediana, este valor reemplaza al píxel central de la vecindad.

Ahora que ya entendimos las diferencias entre los distintos tipos de filtros y el proceso de convolución, estamos listos para conocer los filtros más utilizados en procesamiento digital de imágenes: filtro de promedio, filtro gaussiano, filtro bilateral y filtro de mediana. La explicación de cada uno de ellos, junto con su correspondiente instrucción en Python utilizando OpenCV, se presenta en la Figura 93.

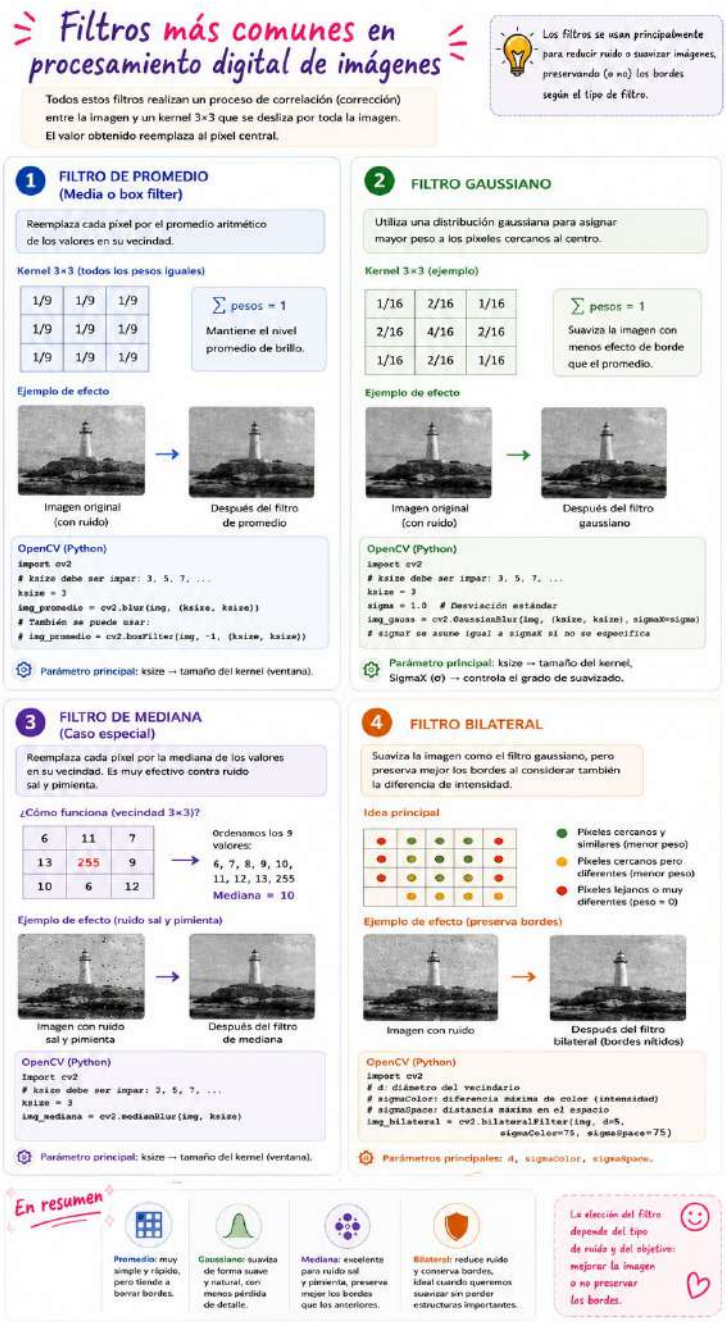


Figura 93. Filtros para reducción de ruido en imágenes: kernel, descripción y código en Python.

Y de esta forma, tenemos que de manera general:

- Si la imagen presenta **ruido sal y pimienta**, el filtro más recomendado es el **filtro de mediana**. Esto se debe a que los valores de sal (255) y pimienta (0) suelen ubicarse en los extremos al ordenar los valores de la vecindad, por lo que normalmente no corresponden a la mediana utilizada para reemplazar el píxel central. En este filtro, el principal parámetro por ajustar es el tamaño del kernel, el cual suele ser de 3 × 3, 5 × 5, 7 × 7, etc. En general, no se recomienda utilizar kernels demasiado grandes, ya que pueden eliminar detalles importantes de la imagen.
- Por otro lado, si la imagen presenta **ruido gaussiano**, el filtro más recomendado es el **filtro bilateral**, ya que considera simultáneamente la proximidad espacial entre los píxeles y la similitud de intensidad (o color). Gracias a ello, es capaz de reducir el ruido preservando mejor los bordes que otros filtros de suavizado.

  En este caso deben ajustarse tres parámetros principales:
  - ❖ d: diámetro de la vecindad utilizada durante el filtrado.
  - ❖ sigmaColor: controla qué tanta diferencia de intensidad o color pueden tener dos píxeles para ser considerados similares.
  - ❖ sigmaSpace: controla la influencia de la distancia espacial entre los píxeles de la vecindad.
- Finalmente, los filtros de promedio y gaussiano también permiten reducir ruido, especialmente cuando las perturbaciones son suaves o aproximadamente uniformes. Sin embargo, ambos tienden a difuminar los bordes de la imagen, por lo que es recomendable utilizar kernels pequeños. Si aun así observamos una pérdida importante de detalle, una mejor alternativa es utilizar el filtro bilateral.

De esta forma, es importante identificar el tipo de ruido presente en la imagen, para así seleccionar el tipo de filtro que mejor reduce el ruido y configurar adecuadamente el o los parámetros del filtro.

4.6. ¿Qué aprendimos en este capítulo?

Como te has podido dar cuenta, el mundo del procesamiento digital de imágenes tiene sus propias particularidades respecto al procesamiento de

señales de audio, razón por la cual necesitábamos una nueva sección para conocer, entender y mejorar este tipo de señales.

Creo que una de las ideas más importantes que aprendimos es que una imagen no solo puede visualizarse, sino también **analizarse, transformarse y utilizarse para extraer información**. A lo largo de este capítulo vimos cómo mejorar su contraste mediante histogramas y ecualización, cómo identificar regiones de interés utilizando detección de bordes y segmentación, y cómo el ruido puede afectar significativamente la calidad de una imagen y, como consecuencia, el desempeño de los algoritmos que trabajan sobre ella.

También comprendimos que no existe un único filtro adecuado para todas las situaciones. La selección del filtro depende del tipo de ruido presente en la imagen y del objetivo que se persiga, ya sea suavizar la imagen, preservar los bordes o eliminar valores atípicos. En ingeniería, como ocurre con frecuencia, **la mejor herramienta siempre dependerá del problema que se desea resolver**.

En el siguiente capítulo dejaremos un poco la teoría para pasar a la práctica. De forma similar a lo que hicimos en la Sección I, abordaremos un caso de estudio que nos permitirá aplicar los conceptos aprendidos a un problema real de segmentación de imágenes, evaluando distintas estrategias para determinar cuál ofrece los mejores resultados.

Al igual que ocurrió con las señales de audio, comprender primero los fundamentos hará que las aplicaciones prácticas del siguiente capítulo tengan mucho más sentido. **Y quizá descubras que una buena solución de ingeniería no comienza cuando ejecutamos un algoritmo, sino mucho antes: cuando entendemos el problema y aprendemos a mirar la información con los ojos adecuados.**

CAPÍTULO V

SEGMENTACIÓN DE IMÁGENES CON TÉCNICAS CLÁSICAS DE PDI

Hasta este momento hemos recorrido los principales fundamentos del procesamiento digital de imágenes. Como habrás notado, este libro no pretende sustituir los textos clásicos de Procesamiento Digital de Imágenes. En ellos encontrarás un tratamiento mucho más profundo de los temas estudiados en el capítulo anterior, así como numerosas técnicas que aquí no hemos abordado, entre ellas las operaciones morfológicas avanzadas, la transformada de Hough, la detección de esquinas, la restauración de imágenes y muchas otras herramientas que forman parte de esta disciplina.

Sin embargo, el propósito de este libro es diferente. Más que presentar un catálogo de algoritmos, busca mostrar cómo un ingeniero construye una solución a partir de un problema real. En la práctica, desarrollar un sistema de visión por computador implica mucho más que aplicar unas cuantas funciones de una librería como OpenCV. Significa comprender el problema, analizar las características de las imágenes disponibles, decidir cómo prepararlas y entender que cada una de estas decisiones influirá en el desempeño de las etapas posteriores.

Con esta idea en mente, durante este capítulo y el siguiente trabajaremos sobre un caso de estudio inspirado en un proyecto de investigación desarrollado hace algunos años, cuyo objetivo era la detección automática de enfermedades en hojas de café. En aquel momento nos enfrentamos a un reto que, a primera vista, parecía sencillo: entrenar un modelo capaz de identificar diferentes

enfermedades a partir de imágenes. Sin embargo, pronto descubrimos que el verdadero desafío comenzaba mucho antes del entrenamiento del modelo.

Las imágenes habían sido capturadas en condiciones reales de cultivo. Por esta razón, presentaban diferencias de iluminación, fondos muy variados, hojas con diferentes tamaños, orientaciones y niveles de afectación, además de ruido e información que no aportaba al problema. Antes de pensar en un modelo de clasificación o en Inteligencia Artificial, fue necesario desarrollar una estrategia que permitiera preparar el conjunto de datos y obtener imágenes más homogéneas y representativas.

Precisamente ese será el objetivo de este capítulo. Aplicaremos los conceptos estudiados hasta el momento para construir un procedimiento de segmentación basado en técnicas clásicas de procesamiento digital de imágenes. A lo largo del capítulo desarrollaremos **tres soluciones**, cada una más elaborada que la anterior, analizaremos sus fortalezas y limitaciones, y evaluaremos cuál de ellas ofrece los mejores resultados para este caso de estudio.

Y, lo más interesante, comprobaremos que incluso una solución que parece funcionar correctamente siempre puede mejorarse cuando se analiza con mayor detalle.

5.1. Conociendo nuestro conjunto de imágenes (RoCole)

Antes de desarrollar cualquier sistema basado en inteligencia artificial, el primer paso consiste en conocer las características del conjunto de datos disponible. Una inspección visual de las imágenes permite identificar diferentes situaciones que pueden afectar el desempeño del modelo y que deberán ser consideradas durante la etapa de preparación del *dataset*.

Para nuestro caso de estudio utilizaremos imágenes provenientes del **dataset RoCoLe (Robusta Coffee Leaf Images Dataset)**, publicado en 2019 por investigadores de la **Universidad de Santiago de Chile**. Este conjunto de datos fue desarrollado para apoyar investigaciones relacionadas con la detección automática de enfermedades en hojas de café, especialmente roya

y otras afecciones foliares, utilizando imágenes capturadas directamente en condiciones reales de cultivo.

Una de las principales fortalezas de este *dataset* es que no fue construido en un entorno controlado de laboratorio, sino en campo, bajo condiciones reales de adquisición. Como consecuencia, las imágenes presentan iluminación natural, diferentes fondos, hojas completas y parciales, distintas orientaciones, variaciones de tamaño y diversos niveles de afectación. Aunque esta variabilidad representa un reto para cualquier sistema de visión por computador, también convierte a RoCoLe en un excelente caso de estudio para integrar los conceptos desarrollados en el **Capítulo 4** de este libro.

La Figura 94 presenta algunos ejemplos de imágenes extraídas del *dataset* RoCoLe. Este conjunto de datos está conformado por **1560 imágenes a color** correspondientes a hojas de café clasificadas en las siguientes categorías: **saludable, ácaro rojo, roya nivel 1, roya nivel 2, roya nivel 3 y roya nivel 4**.

Además, las imágenes presentan una gran variabilidad en sus condiciones de captura, incluyendo **fondos diferentes, iluminación variable, cambios de escala, distintos tamaños, orientaciones diversas y la presencia de ramas, suelo y vegetación**. En consecuencia, el reto no consiste únicamente en reconocer una enfermedad, sino en hacerlo de forma confiable aun cuando las imágenes hayan sido adquiridas bajo condiciones muy diferentes entre sí.

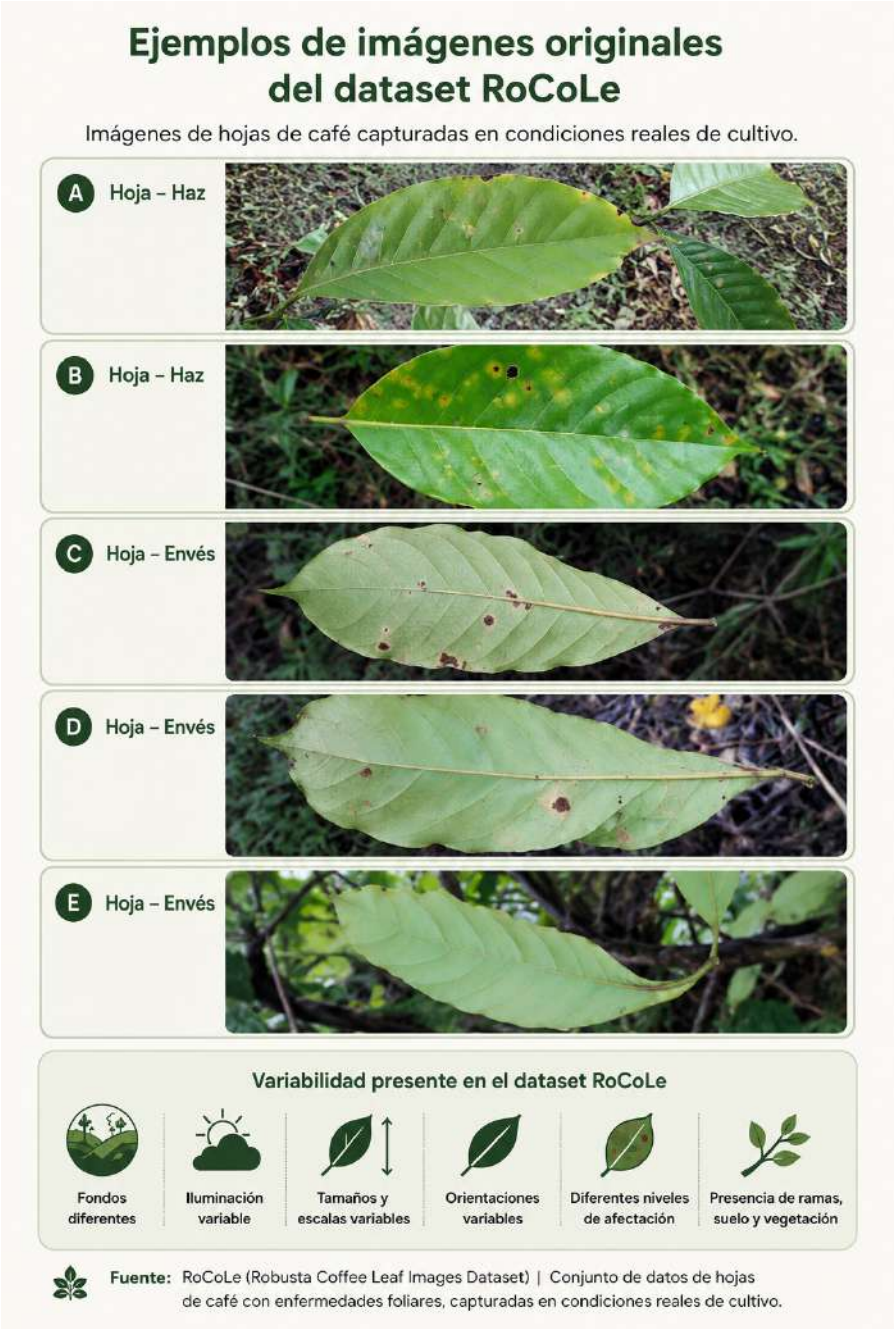


Figura 94. Ejemplo ilustrativo de imágenes pertenecientes al dataset RoCole.

Después de realizar esta primera inspección del conjunto de imágenes, surge una pregunta fundamental:

¿Están realmente listas estas imágenes para ser utilizadas por un modelo de inteligencia artificial?

La respuesta, como ocurre en la mayoría de los proyectos reales de visión por computador, es no del todo. Aunque las imágenes contienen la información requerida para identificar las enfermedades, también presentan múltiples variaciones que pueden dificultar el proceso de aprendizaje del modelo. Antes del entrenamiento es necesario analizar cuidadosamente cuáles de esas variaciones aportan información útil y cuáles, por el contrario, podrían desviar la atención del algoritmo hacia patrones irrelevantes.

Por ejemplo, **imagina que todas las imágenes correspondientes a hojas con roya de nivel 1 hubieran sido capturadas sobre el suelo y con la hoja orientada siempre en la misma dirección**. En ese caso, el modelo podría aprender a asociar el **fondo** o la **orientación de la hoja** con la clase *roya nivel 1*, en lugar de aprender las características propias de la enfermedad. Como consecuencia, cuando el modelo recibiera una nueva imagen con un fondo diferente o con otra orientación, su desempeño podría disminuir considerablemente para la identificación de este nivel de roya.

Por esta razón, la preparación del conjunto de datos tiene como objetivo reducir aquellas variaciones que no aportan información al problema de clasificación, permitiendo que el modelo concentre su atención en las características realmente importantes.

Entonces, nuestra primera tarea consistirá en eliminar automáticamente el fondo de las imágenes. El objetivo es muy sencillo: hacer que el modelo “mire” principalmente la hoja y no el entorno donde fue tomada la fotografía. Para ello reemplazaremos el fondo por un color uniforme (negro o blanco), disminuyendo la influencia de elementos como el suelo, las ramas o la vegetación durante el proceso de aprendizaje.

Pero aparece una pregunta que supongo en este momento te estás haciendo:

¿Cómo le enseñamos a un computador cuáles píxeles pertenecen a la hoja y cuáles pertenecen al fondo?

La buena noticia es que **ya conocemos una posible estrategia**. En el **Capítulo 4** estudiamos la segmentación de imágenes utilizando el modelo **HSV**, donde aprendimos que es posible aislar regiones de una imagen seleccionando rangos específicos de color. En nuestro caso, dado que las hojas presentan predominantemente tonalidades verdes, utilizaremos esta técnica como punto de partida para separar la hoja del fondo.

Como ocurre en la mayoría de los problemas de ingeniería, nuestro objetivo no será obtener una segmentación perfecta desde el primer intento, sino desarrollar un procedimiento que conserve la mayor parte de la hoja y elimine la mayor cantidad posible de información del fondo.

5.2. Reto: Separar automáticamente la hoja principal del fondo

A primera vista, separar automáticamente la hoja del fondo puede parecer una tarea sencilla para una persona, ya que nuestro cerebro identifica de forma casi inmediata cuál es la hoja principal y cuál es el fondo. Sin embargo, para un computador esta información no existe de manera explícita: una imagen no es más que una matriz de píxeles, donde cada píxel almacena un valor de color.

Por esta razón, debemos diseñar un procedimiento que permita diferenciar automáticamente los píxeles pertenecientes a la hoja de aquellos correspondientes al fondo. Una de las estrategias más utilizadas consiste en aprovechar las diferencias de color entre ambos, utilizando para ello el modelo **HSV**, estudiado en el capítulo anterior.

Nuestro objetivo será generar una **máscara binaria**, donde los píxeles pertenecientes a la hoja se representen en blanco y el fondo en negro. Posteriormente, dicha máscara se utilizará para conservar únicamente la región de interés y eliminar el resto de la imagen.

Ejemplo 27. Segmentación de la hoja mediante una máscara binaria en el modelo HSV

La idea consiste en definir un rango de tonalidades verdes y generar una máscara binaria, donde los píxeles que cumplen las condiciones establecidas se representan en blanco y los demás en negro. Posteriormente, aplicaremos esta máscara sobre la imagen original para conservar únicamente las regiones seleccionadas.

Vamos a aplicar esta estrategia sobre cinco imágenes obtenidas del *dataset* RoCoLe, utilizando el siguiente código en Python, muy similar al presentado en el **Ejemplo 25**.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

nombres = ["C10P11E2.jpg", "C10P10H2.jpg", "C10P10H1.jpg",
           "C10P10E2.jpg", "C10P10E1.jpg"] # Imágenes de RoCoLe

lim_inf = np.array([25, 35, 25]) # Límite inferior HSV para tonalidades verdes
lim_sup = np.array([95, 255, 255]) # Límite superior HSV para tonalidades verdes

plt.figure(figsize=(15, 7))

for i, nombre in enumerate(nombres):
    imagen_bgr = cv2.imread(nombre) # Leer imagen
    imagen_rgb = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2RGB) # Convertir a RGB
    imagen_hsv = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2HSV) # Convertir a HSV

    mascara = cv2.inRange(imagen_hsv, lim_inf, lim_sup) # Crear máscara binaria
    segmentada = cv2.bitwise_and(imagen_rgb, imagen_rgb, mask=mascara) # Aplicar máscara

    plt.subplot(2, 5, i + 1); plt.imshow(imagen_rgb)
    plt.title(f"Original {i + 1}"); plt.axis("off")
    plt.subplot(2, 5, i + 6); plt.imshow(segmentada)
    plt.title(f"Segmentada {i + 1}"); plt.axis("off")
plt.tight_layout(); plt.show()
```

Figura 95. Código en Python del Ejemplo 27. Segmentación de la imagen mediante una máscara de color.

El código comienza definiendo los límites inferior ([25, 35, 25]) y superior ([95, 255, 255]) del modelo HSV para segmentar tonalidades verdes. Recordemos que el canal H (Hue) es el encargado de definir el color, mientras que los canales S y V permiten restringir la saturación y el brillo. Con estos límites, la función cv2.inRange() genera una máscara binaria donde los píxeles dentro del rango toman el valor 255 (blanco) y los demás 0 (negro). Finalmente, esta máscara se aplica sobre la imagen original para conservar únicamente las regiones segmentadas. El resultado se presenta en la Figura 95.

Como era de esperarse, la segmentación basada únicamente en el modelo HSV no logra separar completamente la hoja del fondo. Aunque la hoja principal fue correctamente identificada en todos los casos, también se segmentaron otras regiones de la imagen con tonalidades verdes similares, como hojas y vegetación presentes en el fondo.

Sin embargo, observamos que los mejores resultados se obtuvieron en las imágenes **Original 2, Original 3 y Original 4**. En estos casos, el fondo está constituido principalmente por tierra, hojas secas y vegetación con tonalidades diferentes al verde de la hoja, lo que facilita la segmentación por color. En cambio, en las imágenes **Original 1** y **Original 5** aparecen otras hojas verdes muy próximas a la hoja de interés, las cuales también son seleccionadas por la máscara al compartir un rango de color similar.

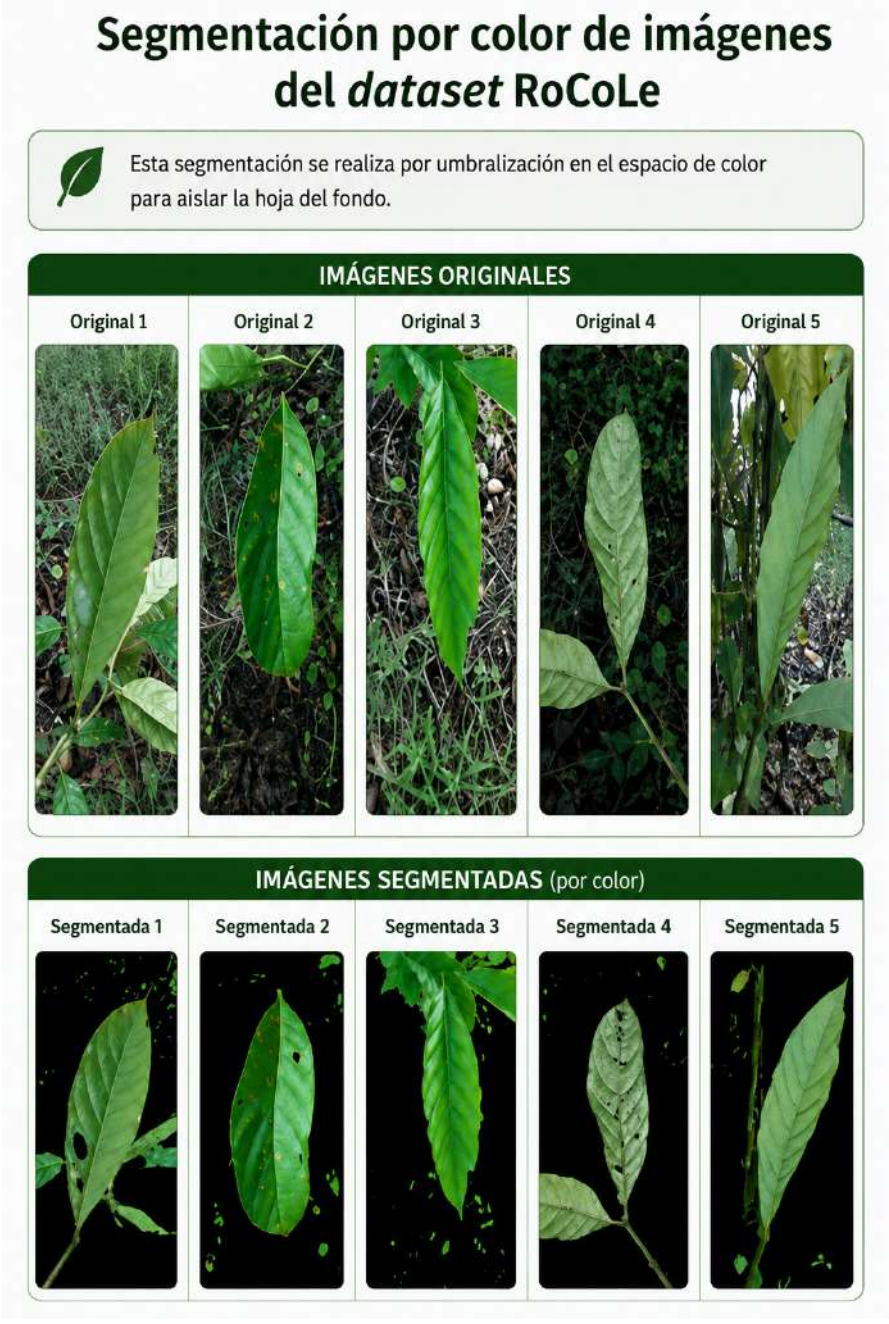


Figura 96. Segmentación por color de imágenes del *dataset* RoCoLe. Primera solución utilizando máscara de color.

En consecuencia, esta primera aproximación constituye un buen punto de partida, pero aún requiere una etapa adicional que permita conservar únicamente la hoja principal.

Entonces surge una nueva pregunta:

¿Cómo podemos indicarle al computador cuál de todas las regiones verdes corresponde realmente a la hoja que queremos analizar?

Ejemplo 28. Segmentación de la hoja con contornos y máscara binaria en el modelo HSV

En el ejemplo anterior comprobamos que la segmentación basada únicamente en el modelo HSV permite identificar correctamente las tonalidades verdes de la imagen. Sin embargo, también observamos que esta estrategia conserva numerosas regiones pertenecientes al fondo, como hojas, ramas y vegetación, ya que todas ellas presentan colores similares a los de la hoja de interés.

Una primera idea para mejorar este resultado consiste en aprovechar una característica adicional de las imágenes del *dataset* RoCoLe. Al observar cuidadosamente las fotografías, notamos que la hoja que deseamos analizar suele corresponder a la región verde de mayor tamaño dentro de la escena. A partir de esta hipótesis, ya no conservaremos todas las regiones segmentadas, sino únicamente aquella cuyo contorno presente el área más grande.

En otras palabras, la segmentación basada en color continuará siendo el primer paso del algoritmo, pero posteriormente incorporaremos un criterio basado en el tamaño de las regiones para eliminar gran parte de la vegetación del fondo y conservar únicamente la hoja principal.

Aplicaremos esta estrategia sobre las mismas cinco imágenes utilizadas en el ejemplo anterior para comparar los resultados obtenidos.

```python
import numpy as np
import matplotlib.pyplot as plt

nombres = ["C10P11E2.jpg", "C10P10H2.jpg", "C10P10H1.jpg",
           "C10P10E2.jpg", "C10P10E1.jpg"] # Imágenes de RoCoLe

lim_inf = np.array([25, 35, 25]) # Límite inferior HSV
lim_sup = np.array([95, 255, 255]) # Límite superior HSV

plt.figure(figsize=(15, 7))

for i, nombre in enumerate(nombres):
    imagen_bgr = cv2.imread(nombre) # Leer imagen
    imagen_rgb = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2RGB) # Convertir a RGB
    imagen_hsv = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2HSV) # Convertir a HSV

    mascara = cv2.inRange(imagen_hsv, lim_inf, lim_sup) # Máscara inicial
    contornos, _ = cv2.findContours(mascara, cv2.RETR_EXTERNAL,
                                    cv2.CHAIN_APPROX_SIMPLE) # Encontrar regiones
    mascara_roi = np.zeros_like(mascara) # Crear máscara negra

    if contornos:
        mayor = max(contornos, key=cv2.contourArea) # Contorno de mayor área
        cv2.drawContours(mascara_roi, [mayor], -1, 255, -1) # Rellenar región
    segmentada = cv2.bitwise_and(imagen_rgb, imagen_rgb,
                                 mask=mascara_roi) # Conservar hoja principal

    plt.subplot(2, 5, i + 1); plt.imshow(imagen_rgb)
    plt.title(f"Original {i + 1}"); plt.axis("off")
    plt.subplot(2, 5, i + 6); plt.imshow(segmentada)
    plt.title(f"Región de interés {i + 1}"); plt.axis("off")
plt.tight_layout(); plt.show()
```

Figura 97. Código en Python del Ejemplo 28. Segmentación de la imagen mediante una máscara de color y análisis de contornos.

Observa que la mayor parte del código es idéntica al del ejemplo anterior. La única diferencia consiste en que ahora no utilizamos directamente la máscara binaria obtenida mediante HSV. En su lugar, identificamos todos los contornos presentes en la máscara, seleccionamos el de mayor área y construimos una nueva máscara utilizando únicamente esa región. De esta forma, el algoritmo conserva la hoja principal y elimina una parte importante de la vegetación del fondo.

El resultado obtenido se presenta en la Figura 98.

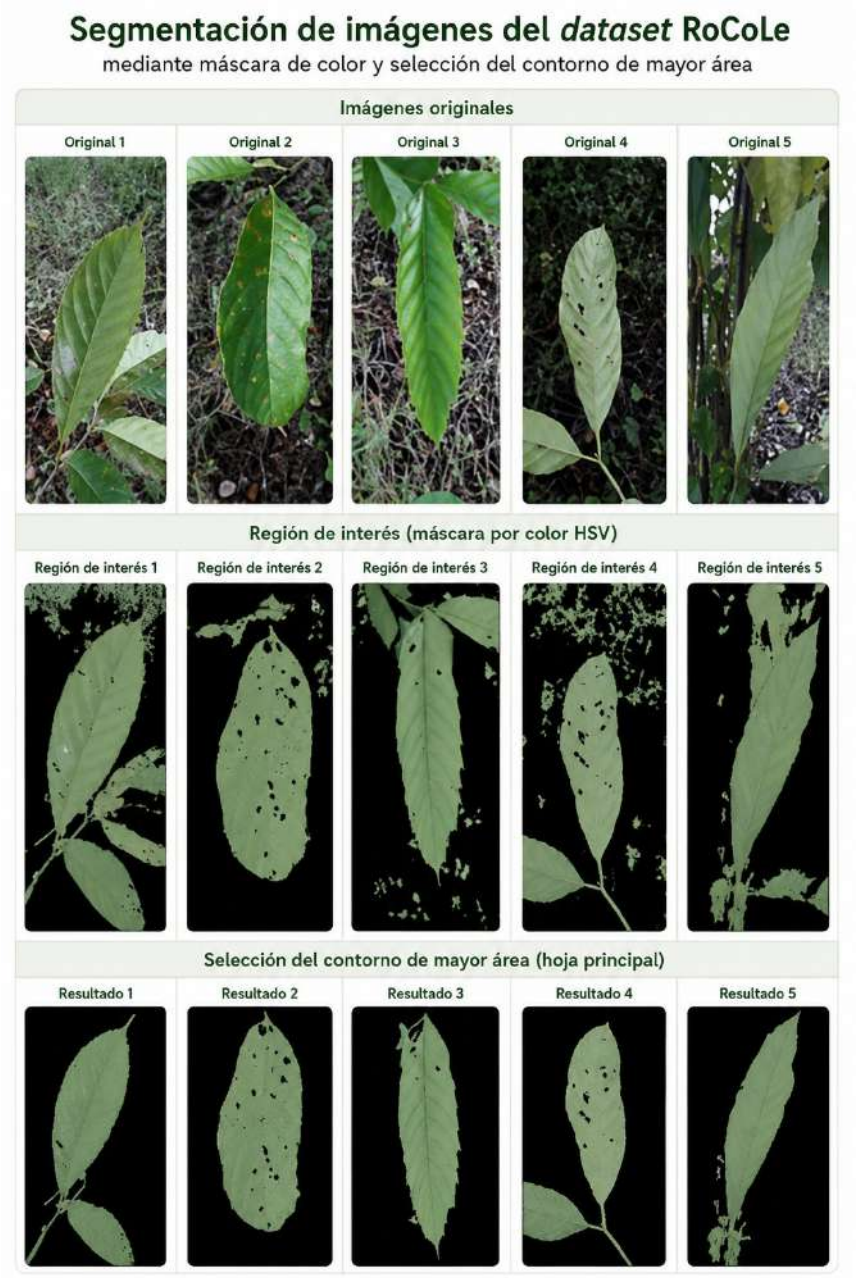


Figura 98. Segmentación de imágenes del *dataset* RoCoLe mediante máscara de color y selección del contorno de mayor área. Segunda solución.

Aunque todavía no obtenemos una segmentación perfecta, sí es posible apreciar una mejora respecto a la primera solución. En términos generales, la incorporación del análisis de contornos permite:

- Reducir la cantidad de vegetación del fondo conservada en la imagen segmentada.
- Disminuir la aparición de falsos negros en el interior de algunas hojas, preservando mejor algunas regiones correspondientes a manchas blancas o amarillas presentes en la hoja original.
- Obtener una hoja principal más continua, especialmente en las imágenes Región de Interés 2, Región de Interés 3 y Región de Interés 4.
- Mantener una mayor parte de la información útil de la hoja sin incrementar significativamente la complejidad del algoritmo.
-

Sin embargo, observamos que todavía permanecen regiones del fondo conectadas a la hoja principal en algunos casos. Esto nos lleva a plantear una nueva pregunta:

¿es posible incorporar información adicional para seleccionar de manera más precisa la región de interés?

Ejemplo 29. Segmentación de la hoja mediante contornos, operaciones morfológicas y análisis de centroides

En el ejemplo anterior utilizamos el área del contorno como criterio para seleccionar la hoja principal. Aunque esta estrategia permitió reducir una parte importante de la vegetación presente en el fondo, todavía se observaron algunos casos donde la región seleccionada incluía información no deseada o presentaba pequeños huecos en el interior de la hoja.

Una forma de refinar esta solución consiste en incorporar información adicional sobre las regiones segmentadas. Para ello, antes de seleccionar la región de interés, aplicaremos dos operaciones morfológicas sobre la máscara binaria: una **apertura**, con el propósito de eliminar pequeñas regiones aisladas, y un **cierre**, para reducir pequeños huecos y conectar regiones próximas pertenecientes a la misma hoja.

Posteriormente, identificaremos todas las regiones conectadas presentes en la máscara y calcularemos, para cada una de ellas, diferentes características, entre ellas su **área**, la **posición de su centroide** y la proporción de píxeles ubicados en la zona central de la imagen. Finalmente, se conservará la región que obtenga la mejor puntuación de acuerdo con estos criterios.

Aplicaremos esta nueva estrategia sobre las mismas cinco imágenes utilizadas en los ejemplos anteriores para comparar los resultados obtenidos.

```
for i, nombre in enumerate(nombres):
    imagen_bgr = cv2.imread(nombre) # Leer imagen
    imagen_rgb = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2RGB) # Convertir a RGB
    imagen_hsv = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2HSV) # Convertir a HSV

    mascara = cv2.inRange(imagen_hsv, lim_inf, lim_sup) # Máscara inicial
    kernel = np.ones((5, 5), np.uint8)
    mascara = cv2.morphologyEx(mascara, cv2.MORPH_OPEN,
                               kernel, iterations=1) # Eliminar regiones pequeñas
    mascara = cv2.morphologyEx(mascara, cv2.MORPH_CLOSE,
                               kernel, iterations=2) # Cerrar huecos pequeños
    num, etiquetas, stats, centroides = cv2.connectedComponentsWithStats(
        mascara, connectivity=8) # Regiones conectadas

    alto, ancho = mascara.shape
    centro_img = np.array([ancho / 2, alto / 2])

    # Región central de referencia
    x1, x2 = int(0.25 * ancho), int(0.75 * ancho)
    y1, y2 = int(0.15 * alto), int(0.90 * alto)

    mejor_etiqueta = 0; mejor_puntaje = 0
    for j in range(1, num):
        x, y, w, h, area = stats[j]
        centro = centroides[j]

        region = etiquetas == j
        pixeles_centrales = np.sum(region[y1:y2, x1:x2])
        proporcion_central = pixeles_centrales / max(area, 1)

        distancia = np.linalg.norm(centro - centro_img)
        cercania = 1 / (1 + distancia)

        toca_borde = x == 0 or y == 0 or x + w >= ancho or y + h >= alto
        penalizacion = 0.40 if toca_borde else 1.0

        puntaje = area * (1 + 3 * proporcion_central) * cercania * penalizacion

        if puntaje > mejor_puntaje:
            mejor_puntaje = puntaje
            mejor_etiqueta = j
    mascara_roi = np.zeros_like(mascara)
    mascara_roi[etiquetas == mejor_etiqueta] = 255 # Conservar región seleccionada

    # Rellenar huecos internos de la hoja
    contornos, _ = cv2.findContours(mascara_roi, cv2.RETR_EXTERNAL,
                                    cv2.CHAIN_APPROX_SIMPLE)
    mascara_final = np.zeros_like(mascara_roi)

    if contornos:
        mayor = max(contornos, key=cv2.contourArea)
        cv2.drawContours(mascara_final, [mayor], -1, 255, -1)

    segmentada = cv2.bitwise_and(imagen_rgb, imagen_rgb,
                                 mask=mascara_final) # Aplicar máscara

    plt.subplot(2, 5, i + 1); plt.imshow(imagen_rgb)
    plt.title(f"Original {i + 1}"); plt.axis("off")

    plt.subplot(2, 5, i + 6); plt.imshow(segmentada)
    plt.title(f"Región de interés {i + 1}"); plt.axis("off")

plt.tight_layout(); plt.show()
```

Figura 99. Código en Python del Ejemplo 29. Segmentación de la hoja mediante operaciones morfológicas, análisis de contornos y centroides.

Observa que este algoritmo mantiene la estrategia de segmentación basada en HSV y la selección de la región principal mediante contornos. La diferencia respecto al ejemplo anterior consiste en que ahora se incorporan **operaciones morfológicas** para mejorar la calidad de la máscara binaria y un **criterio de selección más robusto**, basado no solo en el área de cada región, sino también en su ubicación dentro de la imagen.

El resultado obtenido se presenta en la Figura 100.

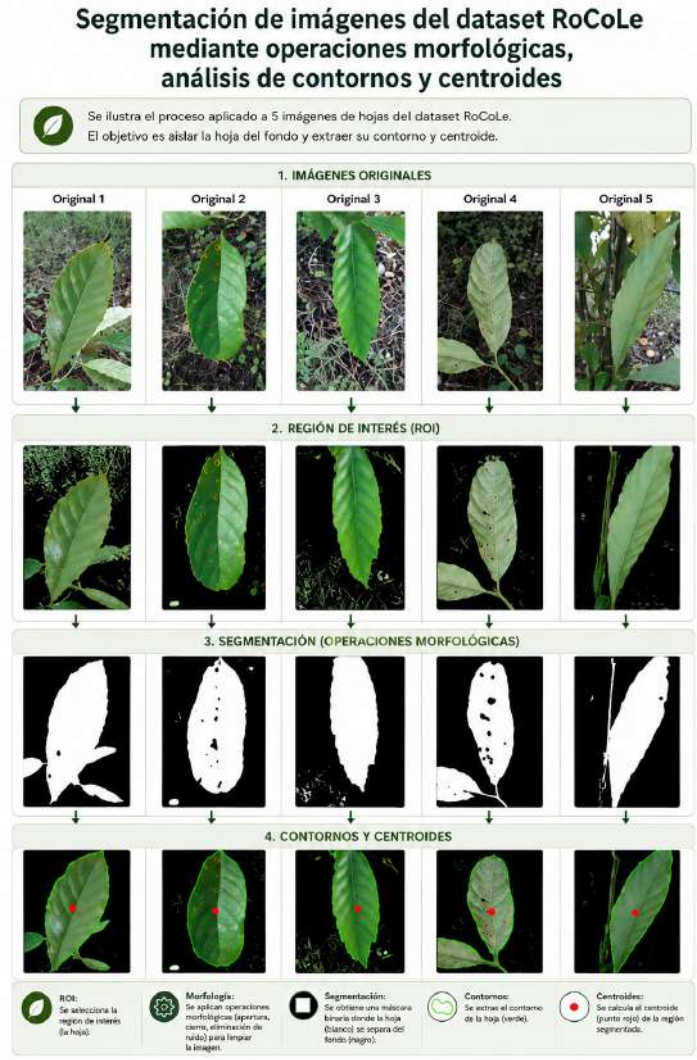


Figura 100. Segmentación de imágenes del dataset RoCoLe mediante operaciones morfológicas, análisis de contornos y centroides. Tercera solución.

Si bien las diferencias respecto a la segunda solución son, en términos generales, sutiles, en la **Región de Interés 4** sí se aprecia una mejora importante. En esta imagen se reducen varios de los falsos negros presentes en el interior de la hoja, conservando mejor las manchas originalmente asociadas con la roya. Además, se cumple parcialmente el objetivo de convertir el entorno en un fondo homogéneo, ya que buena parte de ramas, hojas secas y suelo pasan a representarse en color negro. Aunque la eliminación del fondo aún no es perfecta, la hoja adquiere un mayor protagonismo y queda mejor diferenciada de la escena.

5.3. Análisis comparativo de las soluciones propuestas en el Reto

Después de desarrollar las tres estrategias de segmentación, resulta conveniente comparar sus resultados para identificar las ventajas y limitaciones de cada una de ellas. Recordemos que el objetivo del reto no consistía en obtener una segmentación perfecta, sino en construir, paso a paso, una solución capaz de reducir progresivamente la influencia del fondo y resaltar la hoja de interés.

La Figura 101 resume la evolución de las tres soluciones desarrolladas durante este reto.

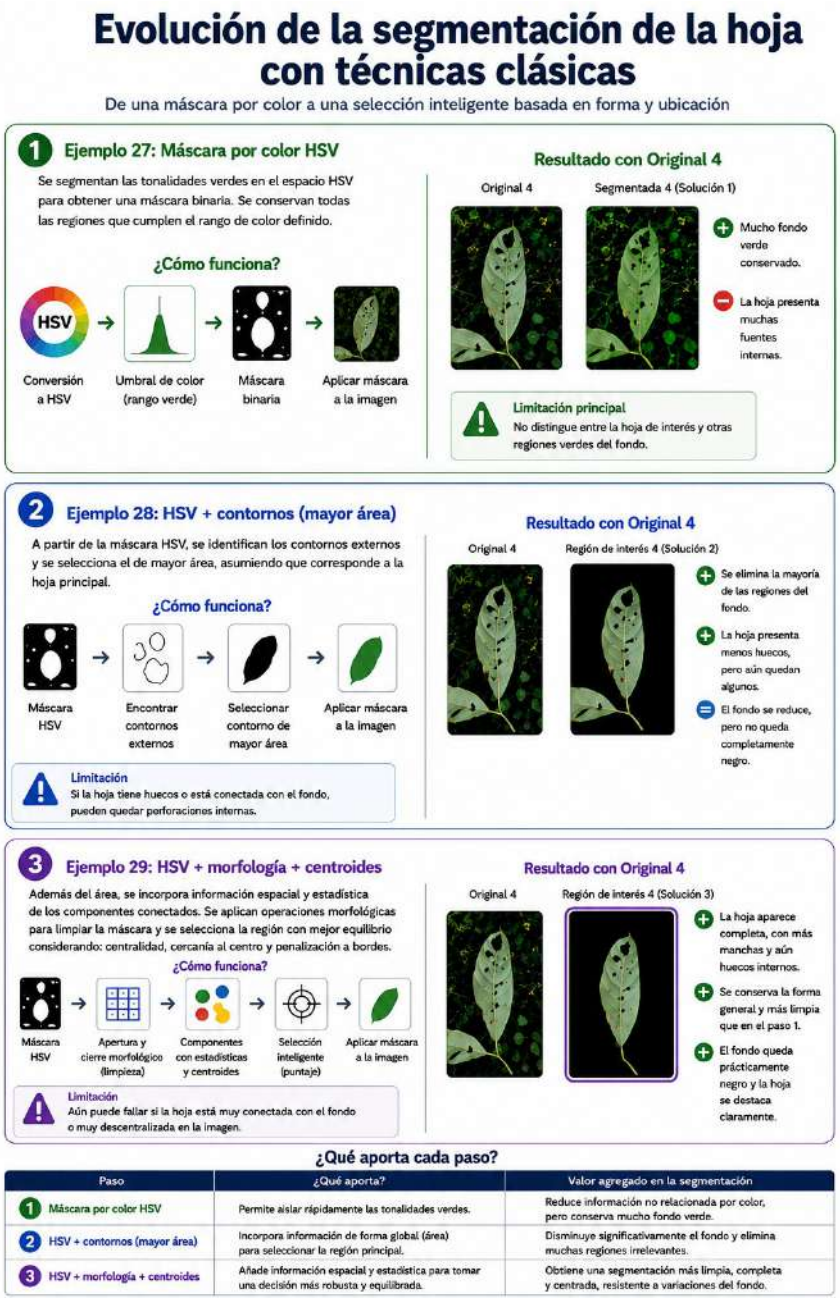


Figura 101. Comparación de las tres estrategias propuestas para la segmentación de la hoja.

A partir de esta comparación podemos observar que cada nueva solución incorpora una etapa adicional de procesamiento y consigue mejorar parcialmente los resultados obtenidos en la etapa anterior. La primera estrategia, basada únicamente en el modelo HSV, permite identificar las tonalidades verdes, pero conserva una gran cantidad de vegetación presente en el fondo y genera falsos negros en el interior de algunas hojas.

La segunda solución incorpora el análisis de contornos para conservar únicamente la región de mayor área, logrando reducir una parte importante del fondo y preservar mejor la estructura de la hoja.

Finalmente, la tercera solución añade operaciones morfológicas y un criterio de selección basado en centroides, refinando la máscara obtenida y mejorando algunos casos particulares, como la **Región de Interés 4**, donde se conservan de mejor manera las manchas asociadas con la roya.

Sin embargo, es importante destacar que **ninguna de las tres soluciones consigue eliminar completamente el fondo en todas las imágenes**. Sin embargo, el objetivo del reto se cumple parcialmente, ya que fue posible transformar gran parte de ramas, hojas secas y suelo en un fondo negro homogéneo, permitiendo que la hoja de interés adquiera un mayor protagonismo dentro de la escena. Más importante aún, este ejercicio demuestra que el desarrollo de una solución de ingeniería rara vez finaliza en el primer intento; por el contrario, es el resultado de un proceso continuo de observación, análisis y refinamiento.

En ingeniería, una solución útil no siempre es una solución perfecta. Con frecuencia, el mayor avance consiste en comprender las limitaciones de una primera aproximación y utilizarlas como punto de partida para construir una mejor.

CAPÍTULO VI

SEGMENTACIÓN AUTOMÁTICA CON INTELIGENCIA ARTIFICIAL

En el capítulo anterior abordamos uno de los problemas más comunes en visión por computador: separar automáticamente un objeto de interés del resto de la escena. Para ello desarrollamos diferentes estrategias basadas en técnicas clásicas de procesamiento digital de imágenes, comenzando con una simple segmentación por color en el modelo HSV y refinando progresivamente los resultados mediante contornos, operaciones morfológicas y análisis de regiones.

Como pudimos comprobar, cada nueva estrategia permitió mejorar parcialmente la calidad de la segmentación. Sin embargo, ninguna de ellas consiguió resolver correctamente todos los casos presentes en el *dataset* RoCoLe. La gran variabilidad en las condiciones de captura, la presencia de hojas superpuestas, ramas, diferentes niveles de iluminación y fondos complejos dificultan la construcción de un conjunto de reglas capaz de funcionar adecuadamente para todas las imágenes.

Entonces surge una pregunta natural:

¿Es posible que un computador aprenda automáticamente a separar la hoja del fondo sin que el ingeniero tenga que diseñar explícitamente todas las reglas de segmentación?

La respuesta es sí, y precisamente ese es uno de los problemas para los cuales el aprendizaje profundo (*Deep Learning*) ha demostrado excelentes resultados. En lugar de programar manualmente un algoritmo de segmentación, construiremos un modelo capaz de aprender esta tarea a partir de ejemplos previamente segmentados por un experto.

En este capítulo utilizaremos una de las arquitecturas más importantes para segmentación semántica de imágenes: **U-Net**. A diferencia de las redes neuronales utilizadas para clasificación, cuyo objetivo consiste en asignar una única etiqueta a toda la imagen, una U-Net realiza una predicción para cada uno de sus píxeles, permitiendo identificar con precisión las regiones que pertenecen al objeto de interés.

Con fines didácticos, trabajaremos inicialmente con un conjunto de entrenamiento muy reducido, compuesto únicamente por cinco imágenes del *dataset* RoCoLe y sus correspondientes máscaras segmentadas manualmente. Aunque este conjunto de datos es insuficiente para desarrollar un sistema robusto, resulta ideal para comprender el flujo completo de entrenamiento de una red de segmentación y analizar cómo el tamaño del conjunto de entrenamiento influye en la capacidad de generalización del modelo.

6.1. ¿Por qué utilizar IA para segmentar imágenes?

La principal dificultad que identificamos en el Capítulo 5 radica en la enorme variabilidad que presentan las imágenes del dataset RoCole. Aunque todas pertenecen a un mismo conjunto de imágenes, existen diferencias en la iluminación, el tamaño de las hojas, su orientación, la presencia de ramas, vegetación, hojas superpuestas y fondos con tonalidades similares al objeto de interés. En estas condiciones, diseñar manualmente un conjunto de reglas capaz de funcionar correctamente para todos los casos puede convertirse en una tarea extremadamente compleja.

Entonces surge una pregunta diferente a las que hemos planteado hasta ahora:

¿Y si, en lugar de programar las reglas de segmentación, permitimos que el computador las aprenda automáticamente a partir de ejemplos?

Esta pregunta representa uno de los cambios más importantes introducidos por la inteligencia artificial moderna. Mientras los métodos clásicos dependen del conocimiento del ingeniero para definir umbrales, filtros o criterios de decisión, los modelos de aprendizaje profundo construyen esas reglas durante el entrenamiento, analizando un conjunto de imágenes para las cuales ya se conoce la respuesta correcta.

En otras palabras, el objetivo deja de ser escribir un algoritmo que separe la hoja del fondo. Ahora el objetivo consiste en enseñar al modelo, mediante ejemplos previamente segmentados, cómo debe realizar esa tarea sobre imágenes que nunca ha visto.

6.2. ¿Qué es U-Net?

En 2015 fue propuesta una arquitectura denominada **U-Net**, desarrollada por Olaf Ronneberger y colaboradores para la segmentación de imágenes biomédicas. Su objetivo inicial consistía en identificar automáticamente regiones de interés, como células, tejidos o lesiones, permitiendo separar con precisión estas estructuras del resto de la imagen. Gracias a su excelente desempeño, esta arquitectura se convirtió rápidamente en un referente para problemas de segmentación y, actualmente, es una de las más utilizadas en aplicaciones de visión por computador, incluyendo agricultura de precisión, imágenes satelitales, inspección industrial, vehículos autónomos y diagnóstico médico.

A diferencia de una red neuronal diseñada para clasificación (por ejemplo, **saludable**, **roya nivel 1**, **roya nivel 2**, etc.), cuya salida corresponde a una única etiqueta para toda la imagen, una **U-Net** genera una imagen de salida con las mismas dimensiones que la imagen de entrada. En otras palabras, tanto la entrada como la salida de la arquitectura corresponden a imágenes. La

diferencia es que, en la imagen de salida, **cada píxel representa la predicción realizada por el modelo**.

En nuestro caso de estudio, cada píxel de la imagen de salida indicará si pertenece a la hoja principal de café o al fondo, permitiendo construir automáticamente la máscara de segmentación.

Pero, te preguntarás ¿Internamente como está conformada una U-Net para poder realizar segmentación de imagen? Los bloques que la conforman y su manera de conexión se presenta en la Figura 101.

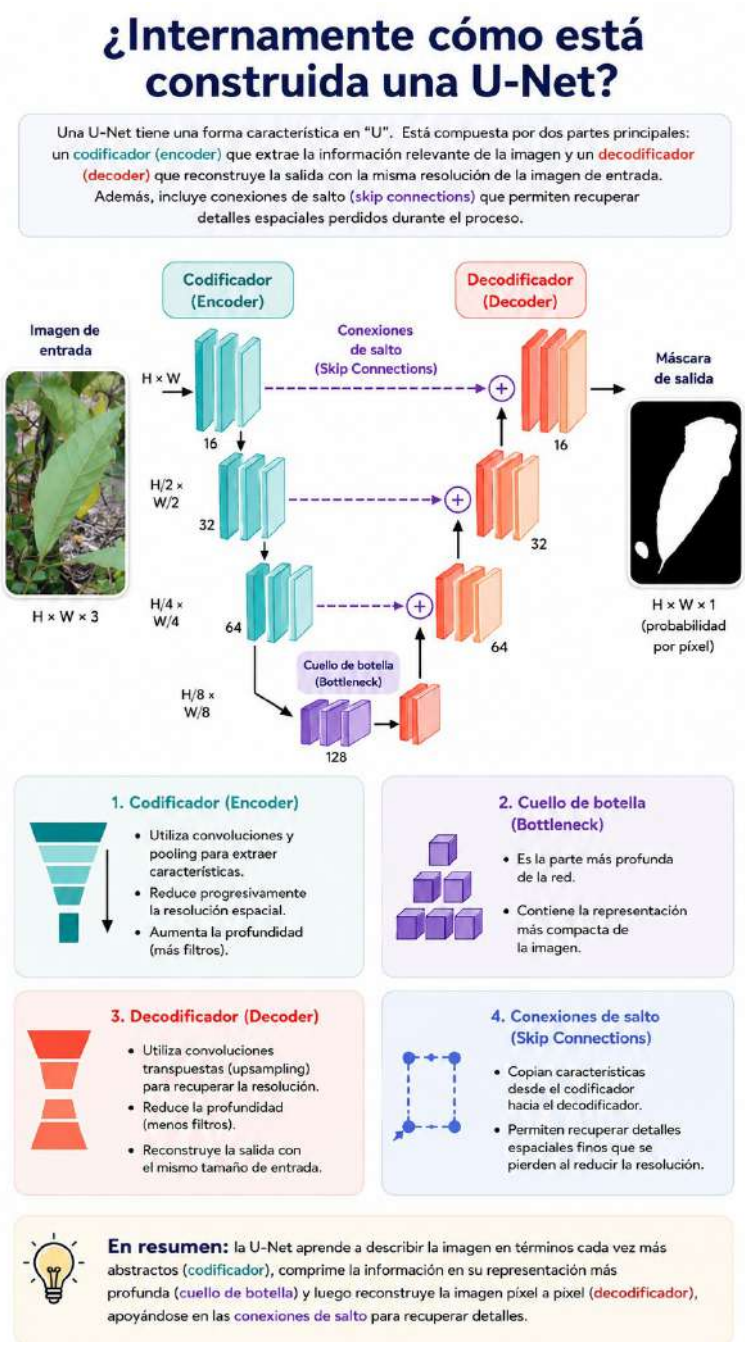


Figura 102. Explicación de la arquitectura U-Net.

Una característica importante de U-Net es que el número de canales cambia continuamente a medida que la información avanza por la red. La imagen de entrada corresponde a una imagen RGB de dimensiones $H \times W \times 3$, donde los tres canales representan las componentes de color **rojo (R)**, **verde (G)** y **azul (B)**.

Sin embargo, después de las primeras convoluciones (como vimos en el Capítulo 4 entre la imagen y el kernel), esos tres canales dejan de representar colores y son reemplazados por un conjunto de **mapas de características** aprendidos automáticamente por la red. Por ejemplo, el primer bloque puede generar **16 mapas de características**, el siguiente **32**, luego **64** y así sucesivamente. Cada uno de estos mapas responde a diferentes patrones presentes en la imagen, como *bordes*, *texturas*, *formas* o combinaciones más complejas aprendidas durante el entrenamiento.

Mientras el codificador reduce progresivamente la resolución espacial de la imagen, el número de mapas de características aumenta, permitiendo representar la información de manera cada vez más rica y abstracta. Posteriormente, durante el decodificador ocurre el proceso contrario: la resolución espacial vuelve a aumentar, mientras que el número de canales disminuye gradualmente hasta obtener una única imagen de salida de dimensiones $H \times W \times 1$.

Ese único canal ya no representa un color, sino una **probabilidad**. En nuestro caso de estudio, cada píxel almacena la probabilidad de pertenecer a la hoja de café. Finalmente, aplicando un umbral (por ejemplo, 0.5), esa imagen probabilística se transforma en la máscara binaria utilizada para segmentar automáticamente la hoja principal del fondo.

6.3. Preparación del conjunto de imágenes

Como ocurre con cualquier modelo supervisado basado en aprendizaje profundo, una U-Net no puede aprender únicamente observando imágenes. Para entrenarla es necesario disponer de ejemplos donde se conozca previamente la respuesta correcta (es decir, parejas de entrada vs. salida). En un problema

de segmentación, esto significa que **cada imagen de entrada debe estar acompañada de una máscara de referencia**, también conocida como **Ground Truth**, que indique cuáles píxeles pertenecen al objeto de interés y cuáles corresponden al fondo.

En nuestro caso de estudio, el objetivo consiste en segmentar automáticamente la hoja principal de café presente en cada fotografía. Por esta razón, para cada imagen original construimos manualmente una máscara de referencia, eliminando el fondo y conservando únicamente la hoja de interés. Estas máscaras representan la respuesta correcta que la U-Net utilizará durante el entrenamiento para aprender la tarea de segmentación.

Nota aclaratoria: en el aprendizaje supervisado, el modelo aprende a partir de ejemplos donde se conoce previamente la respuesta correcta. Durante el entrenamiento, compara continuamente su predicción con dicha respuesta y ajusta sus parámetros para reducir el error. Este mismo principio se utiliza en problemas de clasificación, regresión y segmentación de imágenes.

La Figura 103 presenta las cinco parejas de imágenes utilizadas en este capítulo. En la fila superior se muestran las imágenes originales del *dataset* RoCoLe y, en la fila inferior, las correspondientes **máscaras de referencia (*Ground Truth*)**, obtenidas mediante la herramienta **iloveimg** y posteriormente refinadas mediante edición manual. Durante el entrenamiento, la red recibe la imagen original como entrada y compara continuamente su predicción con la máscara de referencia, ajustando sus parámetros para reducir el error entre ambas.

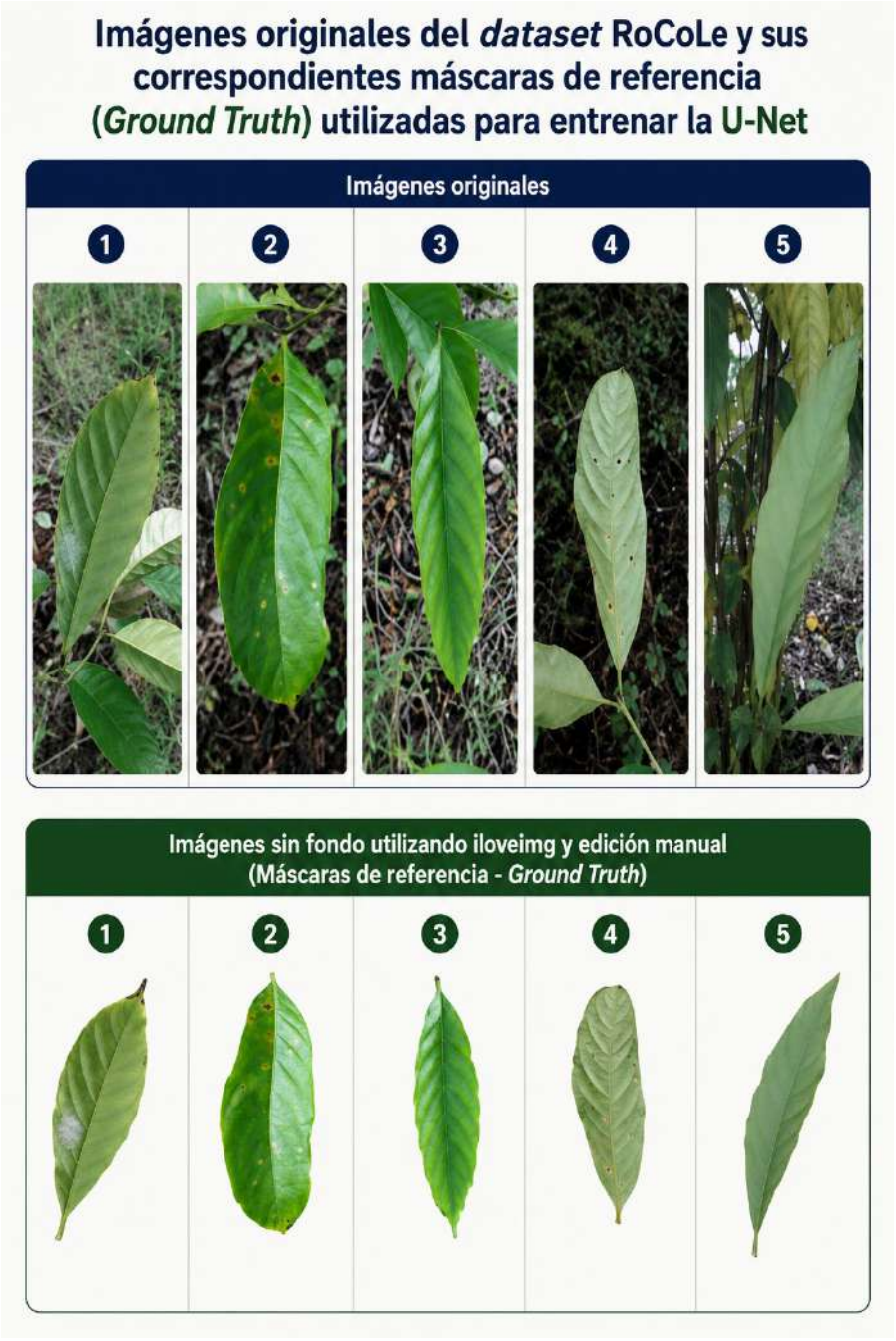


Figura 103. Imágenes originales del dataset RoCoLe y sus correspondientes máscaras de referencia (Ground Truth) utilizadas para entrenar la U-Net.

Aunque en aplicaciones reales suelen emplearse cientos o miles de parejas de entrenamiento, en este capítulo utilizaremos únicamente cinco imágenes con fines didácticos. Este conjunto reducido nos permitirá comprender el flujo completo de entrenamiento de una red de segmentación y, al mismo tiempo, analizar cómo el tamaño del conjunto de entrenamiento influye directamente en la capacidad de generalización del modelo.

6.4. Entrenamiento de la U-Net y Resultados

Una vez preparado el conjunto de entrenamiento, el siguiente paso consiste en entrenar la U-Net para que aprenda a segmentar automáticamente la hoja de café. Durante este proceso, la red analiza repetidamente cada una de las parejas de entrenamiento, formadas por la imagen original y su correspondiente máscara de referencia, ajustando progresivamente sus parámetros internos con el propósito de reducir el error entre la segmentación predicha y la segmentación esperada.

Desde un punto de vista conceptual, el entrenamiento de una U-Net puede resumirse en cuatro etapas fundamentales: lectura y preprocesamiento de las imágenes, construcción de la arquitectura, proceso de entrenamiento, y evaluación del modelo. La Figura 104 resume el flujo general seguido durante el entrenamiento de la U-Net.

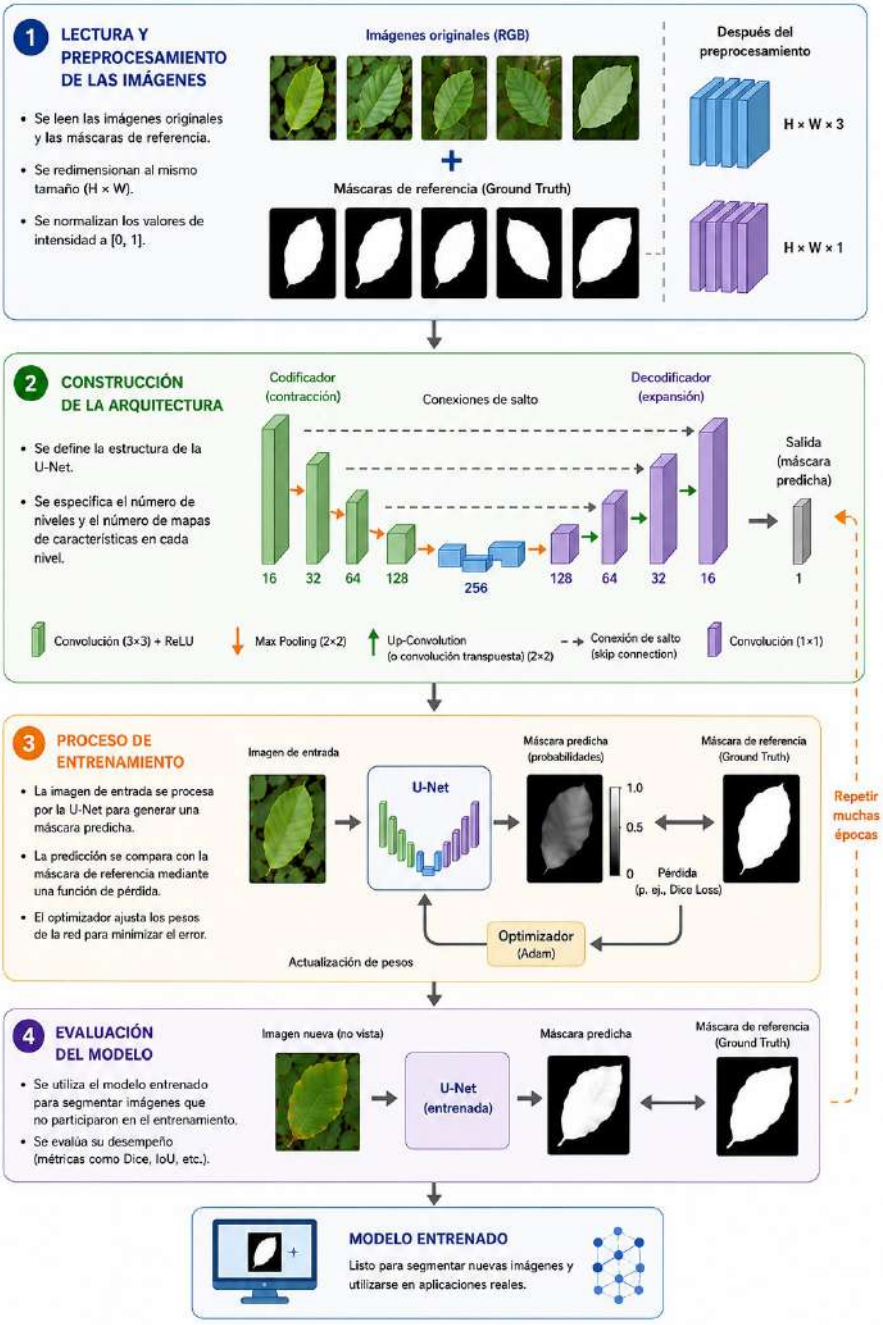


Figura 104. Flujo general de entrenamiento de una U-Net.

A manera de listado, en cada una de las etapas realizaremos:

Lectura y preprocesamiento de las imágenes

- Se verifican las dimensiones del dataset.
- Se calcula el porcentaje de hoja en cada máscara.
- Se visualizan las cinco parejas de entrenamiento.
- El estudiante comprueba que los datos quedaron correctamente preparados.

Construcción de la arquitectura

- Se genera automáticamente el resumen (*summary*) de la U-Net.
- El estudiante verifica que la arquitectura construida corresponde a la explicada en la teoría.

Proceso de entrenamiento

- Se observan las métricas (Dice, IoU y pérdida).
- Se visualizan las curvas de entrenamiento y validación.
- El estudiante comprende cómo evoluciona el aprendizaje de la red.

Evaluación del modelo

- Se obtiene la imagen de probabilidades.
- Se genera la máscara binaria.
- Se produce la imagen segmentada.
- Se compara con el *Ground Truth*.

A continuación, presentaremos un ejemplo de solución utilizando U-Net.

Ejemplo 30. Entrenamiento de una U-Net para la segmentación de la hoja principal del dataset RoCoLe

En este ejemplo entrenaremos una U-Net para segmentar automáticamente la hoja principal presente en las imágenes del *dataset* RoCoLe. Para ello utilizaremos las cinco parejas preparadas previamente, formadas por la imagen original y su correspondiente máscara de referencia. Cuatro parejas se emplearán durante el entrenamiento y la quinta se reservará para evaluar el desempeño del modelo sobre una imagen que no participó en el ajuste de sus parámetros.

Debido a que el conjunto de entrenamiento es extremadamente reducido, el objetivo de este ejemplo no consiste en construir un sistema robusto, sino en comprobar si la arquitectura es capaz de aprender la tarea de segmentación y analizar las limitaciones asociadas a trabajar con pocos datos.

Vamos a dividir el código en varios apartados:

```python
import cv2
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras import layers, Model

# ============================================================
# 1. LECTURA Y PREPROCESAMIENTO DE LAS IMÁGENES
# ============================================================
# 1.1. Configuración inicial
np.random.seed(42)
tf.random.set_seed(42)

TAM = 224

originales = ["C10P11E2.jpg", "C10P10H2.jpg", "C10P10H1.jpg",
              "C10P10E2.jpg", "C10P10E1.jpg"]

editadas = ["mascara_1.png", "mascara_2.png", "mascara_3.png",
            "mascara_4.png", "mascara_5.png"]

for nombre in originales + editadas:
    if not os.path.exists(nombre):
        raise FileNotFoundError(f"No se encontró el archivo: {nombre}")
```

```python
# 1.2. Lectura y normalización de las imágenes originales
X = []

for nombre in originales:
    imagen_bgr = cv2.imread(nombre)

    if imagen_bgr is None:
        raise ValueError(f"No fue posible leer la imagen: {nombre}")

    imagen_rgb = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2RGB)
    imagen_rgb = cv2.resize(imagen_rgb, (TAM, TAM),
                            interpolation=cv2.INTER_AREA)
    imagen_rgb = imagen_rgb.astype(np.float32) / 255.0
    X.append(imagen_rgb)

X = np.array(X, dtype=np.float32)
```

```python
# 1.3. Construcción de las máscaras de referencia
Y = []
for nombre in editadas:
    imagen = cv2.imread(nombre, cv2.IMREAD_UNCHANGED)
    if imagen is None:
        raise ValueError(f"No fue posible leer la imagen editada: {nombre}")
    # Caso 1: imagen PNG con transparencia
    if imagen.ndim == 3 and imagen.shape[2] == 4:
        canal_alfa = imagen[:, :, 3]
        mascara = (canal_alfa > 10).astype(np.uint8)
    # Caso 2: imagen con fondo blanco
    else:
        if imagen.ndim == 2:
            imagen_rgb = cv2.cvtColor(imagen, cv2.COLOR_GRAY2RGB)
        else:
            imagen_rgb = cv2.cvtColor(imagen, cv2.COLOR_BGR2RGB)

        fondo_blanco = np.all(imagen_rgb >= 245, axis=2)
        mascara = (~fondo_blanco).astype(np.uint8)
    mascara = cv2.resize(mascara, (TAM, TAM),
                         interpolation=cv2.INTER_NEAREST)
    mascara = mascara.astype(np.float32)
    mascara = np.expand_dims(mascara, axis=-1)
    Y.append(mascara)
Y = np.array(Y, dtype=np.float32)
```

```python
# 1.4. Verificación de imágenes y máscaras
print("Forma de X:", X.shape)
print("Forma de Y:", Y.shape)

for i in range(len(Y)):
    valores = np.unique(Y[i])
    porcentaje_hoja = 100 * np.mean(Y[i])
    print(f"Máscara {i + 1}: valores = {valores}, "
          f"porcentaje de hoja = {porcentaje_hoja:.2f}%")

plt.figure(figsize=(15, 6))

for i in range(5):
    plt.subplot(2, 5, i + 1)
    plt.imshow(X[i])
    plt.title(f"Original {i + 1}")
    plt.axis("off")

    plt.subplot(2, 5, i + 6)
    plt.imshow(Y[i, :, :, 0], cmap="gray", vmin=0, vmax=1)
    plt.title(f"Máscara {i + 1}")
    plt.axis("off")

plt.tight_layout()
plt.show()
```

```python
# 1.5. División entre entrenamiento y validación
X_train, Y_train = X[:4], Y[:4]
X_val, Y_val = X[4:], Y[4:]

print("Entrenamiento:", X_train.shape, Y_train.shape)
print("Validación:", X_val.shape, Y_val.shape)
```

Figura 105. Código en Python del Ejemplo 30: Sección Lectura y Preprocesamiento de las Imágenes.

Esta **primera sección**, denominada **Lectura y Preprocesamiento de las Imágenes**, está conformada por las siguientes etapas: configuración inicial, lectura y normalización de las imágenes originales, construcción de las máscaras de referencia, verificación de imágenes y máscaras y división del conjunto de datos en entrenamiento y validación.

Es importante que tanto las imágenes originales como las máscaras de referencia tengan las dimensiones requeridas por la arquitectura U-Net (224 × 224 píxeles) y que sus valores se encuentren normalizados. Estas etapas garantizan que todas las parejas de entrenamiento sean consistentes y puedan ser procesadas adecuadamente por la red neuronal. Finalmente, de las cinco parejas disponibles, cuatro se destinan al entrenamiento del modelo y la restante se reserva para la validación, permitiendo evaluar su capacidad de segmentar una imagen que no participó en el proceso de aprendizaje.

Ahora, continuamos con la segunda sección de código, que se presenta en la Figura 106.

```python
# ==========================================
# 2. CONSTRUCCIÓN DE LA ARQUITECTURA
# ==========================================
# 2.1. Métricas y función de pérdida
def dice_coef(y_true, y_pred, suavizado=1e-6):
    y_true = tf.reshape(tf.cast(y_true, tf.float32), [-1])
    y_pred = tf.reshape(tf.cast(y_pred, tf.float32), [-1])

    interseccion = tf.reduce_sum(y_true * y_pred)
    suma = tf.reduce_sum(y_true) + tf.reduce_sum(y_pred)

    return (2.0 * interseccion + suavizado) / (suma + suavizado)

def iou_coef(y_true, y_pred, suavizado=1e-6):
    y_true = tf.reshape(tf.cast(y_true, tf.float32), [-1])
    y_pred = tf.reshape(tf.cast(y_pred, tf.float32), [-1])

    interseccion = tf.reduce_sum(y_true * y_pred)
    union = tf.reduce_sum(y_true) + tf.reduce_sum(y_pred) - interseccion

    return (interseccion + suavizado) / (union + suavizado)

def perdida_bce_dice(y_true, y_pred):
    bce = tf.keras.losses.binary_crossentropy(y_true, y_pred)
    bce = tf.reduce_mean(bce)
    return bce + (1.0 - dice_coef(y_true, y_pred))


# 2.2. Bloque convolucional
def bloque_conv(x, filtros):
    x = layers.Conv2D(filtros, 3, padding="same", activation="relu",
                      kernel_initializer="he_normal")(x)

    x = layers.Conv2D(filtros, 3, padding="same", activation="relu",
                      kernel_initializer="he_normal")(x)
    return x
```

```python
# 2.3. Construcción de una U-Net pequeña
entrada = layers.Input(shape=(TAM, TAM, 3))

# Codificador
c1 = bloque_conv(entrada, 16)
p1 = layers.MaxPooling2D((2, 2))(c1)
c2 = bloque_conv(p1, 32)
p2 = layers.MaxPooling2D((2, 2))(c2)
c3 = bloque_conv(p2, 64)
p3 = layers.MaxPooling2D((2, 2))(c3)

# Cuello de botella
puente = bloque_conv(p3, 128)

# Decodificador
u3 = layers.Conv2DTranspose(64, 2, strides=2, padding="same")(puente)
u3 = layers.Concatenate()([u3, c3])
c4 = bloque_conv(u3, 64)
u2 = layers.Conv2DTranspose(32, 2, strides=2, padding="same")(c4)
u2 = layers.Concatenate()([u2, c2])
c5 = bloque_conv(u2, 32)
u1 = layers.Conv2DTranspose(16, 2, strides=2, padding="same")(c5)
u1 = layers.Concatenate()([u1, c1])
c6 = bloque_conv(u1, 16)

# Salida: probabilidad de pertenecer a la hoja
salida = layers.Conv2D(1, 1, activation="sigmoid")(c6)
modelo = Model(inputs=entrada, outputs=salida, name="UNet_pequena")
modelo.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4),
               loss=perdida_bce_dice,
               metrics=[dice_coef, iou_coef])
modelo.summary()
```

Figura 106. Código en Python del Ejemplo 30: Sección Construcción de la Arquitectura.

Esta **segunda sección**, denominada **Construcción de la Arquitectura**, implementa una U-Net de tamaño reducido para la segmentación automática de la hoja principal. En ella se definen los bloques que conforman el codificador, el cuello de botella y el decodificador, así como las conexiones de salto entre ambos. Finalmente, se especifican el optimizador, la función de pérdida y las métricas utilizadas para entrenar y evaluar el modelo.

A continuación, pasamos a la tercera sección.

```
# ============================================================
# 3. PROCESO DE ENTRENAMIENTO
# ============================================================
# 3.1. Configuración de callbacks
callbacks = [
    tf.keras.callbacks.EarlyStopping(
        monitor="val_loss", patience=25, restore_best_weights=True
    ),

    tf.keras.callbacks.ReduceLROnPlateau(
        monitor="val_loss", factor=0.5, patience=10, min_lr=1e-6
    )
]

# 3.2. Entrenamiento de la U-Net
historial = modelo.fit(
    X_train, Y_train,
    validation_data=(X_val, Y_val),
    epochs=150,
    batch_size=1,
    callbacks=callbacks,
    verbose=1
)
```

```
# 3.3. Curvas de entrenamiento
plt.figure(figsize=(8, 4))
plt.plot(historial.history["loss"], label="Pérdida entrenamiento")
plt.plot(historial.history["val_loss"], label="Pérdida validación")

plt.xlabel("Época")
plt.ylabel("Pérdida")
plt.title("Evolución del entrenamiento")
plt.legend()
plt.tight_layout()
plt.show()
```

Figura 107. Código en Python del Ejemplo 30: Sección Proceso de Entrenamiento.

En la **tercera sección**, denominada **Proceso de Entrenamiento**, se configuran los parámetros necesarios para entrenar la U-Net. En primer lugar, se definen los *callbacks*, encargados de detener automáticamente el entrenamiento cuando el modelo deja de mejorar y de ajustar la tasa de aprendizaje si es necesario. Posteriormente, la red procesa de forma iterativa las imágenes de entrenamiento, compara las máscaras predichas con las máscaras de referencia y actualiza sus parámetros internos para reducir el error. Finalmente, se presentan las curvas de entrenamiento y validación, las cuales permiten analizar la evolución del proceso de aprendizaje a lo largo de las diferentes épocas.

Finalmente, tenemos la cuarta sección del código.

```
# ============================================================
# 4. EVALUACIÓN DEL MODELO
# ============================================================

# 4.1. Predicción sobre la imagen de validación
prediccion = modelo.predict(X_val, verbose=0)[0, :, :, 0]

umbral = 0.5
mascara_predicha = (prediccion >= umbral).astype(np.uint8)
segmentada = X_val[0] * mascara_predicha[:, :, np.newaxis]


# 4.3. Comparación con la máscara de referencia
plt.figure(figsize=(12, 4))

plt.subplot(1, 3, 1)
plt.imshow(Y_val[0, :, :, 0], cmap="gray", vmin=0, vmax=1)
plt.title("Máscara de referencia")
plt.axis("off")

plt.subplot(1, 3, 2)
plt.imshow(prediccion, cmap="gray", vmin=0, vmax=1)
plt.title("Probabilidad predicha")
plt.axis("off")

plt.subplot(1, 3, 3)
plt.imshow(mascara_predicha, cmap="gray", vmin=0, vmax=1)
plt.title("Máscara binaria predicha")
plt.axis("off")

plt.tight_layout()
plt.show()
```

```
# 4.2. Visualización del resultado
plt.figure(figsize=(16, 4))

plt.subplot(1, 4, 1)
plt.imshow(X_val[0])
plt.title("Imagen original")
plt.axis("off")

plt.subplot(1, 4, 2)
plt.imshow(prediccion, cmap="gray", vmin=0, vmax=1)
plt.title("Probabilidad predicha")
plt.axis("off")

plt.subplot(1, 4, 3)
plt.imshow(mascara_predicha, cmap="gray", vmin=0, vmax=1)
plt.title("Máscara predicha")
plt.axis("off")

plt.subplot(1, 4, 4)
plt.imshow(segmentada)
plt.title("Imagen segmentada")
plt.axis("off")

plt.tight_layout()
plt.show()
```

Figura 108. Código en Python del Ejemplo 30: Sección Evaluación del Modelo.

En la **cuarta sección**, denominada **Evaluación del Modelo,** se utiliza la U-Net entrenada para segmentar la imagen reservada para validación. Inicialmente, la red genera una imagen de probabilidades, donde cada píxel indica la probabilidad de pertenecer a la hoja principal. Posteriormente, mediante la aplicación de un umbral, esta imagen probabilística se convierte en una máscara binaria de segmentación. Finalmente, la máscara obtenida se aplica sobre la imagen original y se compara con la máscara de referencia (*Ground Truth*), permitiendo evaluar visualmente el desempeño alcanzado por el modelo.

Pasemos ahora a analizar los resultados obtenidos durante la etapa de preprocesamiento del conjunto de entrenamiento. Inicialmente, verificaremos las dimensiones de las imágenes de entrada y de salida, así como el porcentaje de la imagen ocupado por la hoja en cada una de las máscaras de referencia.

```
Forma de X: (5, 224, 224, 3)
Forma de Y: (5, 224, 224, 1)
Máscara 1: valores = [0. 1.], porcentaje de hoja = 31.03%
Máscara 2: valores = [0. 1.], porcentaje de hoja = 32.79%
Máscara 3: valores = [0. 1.], porcentaje de hoja = 25.52%
Máscara 4: valores = [0. 1.], porcentaje de hoja = 19.97%
Máscara 5: valores = [0. 1.], porcentaje de hoja = 25.48%
```

Observamos que el conjunto de datos está conformado por **cinco imágenes de entrada (X)** y sus correspondientes **cinco máscaras de referencia (Y)**. Todas las imágenes fueron redimensionadas a **224 × 224 píxeles**, de acuerdo con los requerimientos de la arquitectura implementada. Sin embargo, las imágenes de entrada poseen **tres canales** (RGB), mientras que las máscaras contienen **un único canal**, donde el valor **1** representa la hoja y el valor **0** corresponde al fondo.

Adicionalmente, el porcentaje de la imagen ocupado por la hoja varía aproximadamente entre el **20 % y el 33 %**, lo que indica que la mayor parte de los píxeles pertenece al fondo. Esta característica es habitual en problemas de segmentación y explica la importancia de utilizar funciones de pérdida, como la combinación de **Binary Cross-Entropy** y **Dice Loss**, que permiten entrenar el

modelo de manera más robusta cuando existe un desbalance entre la región de interés y el fondo.

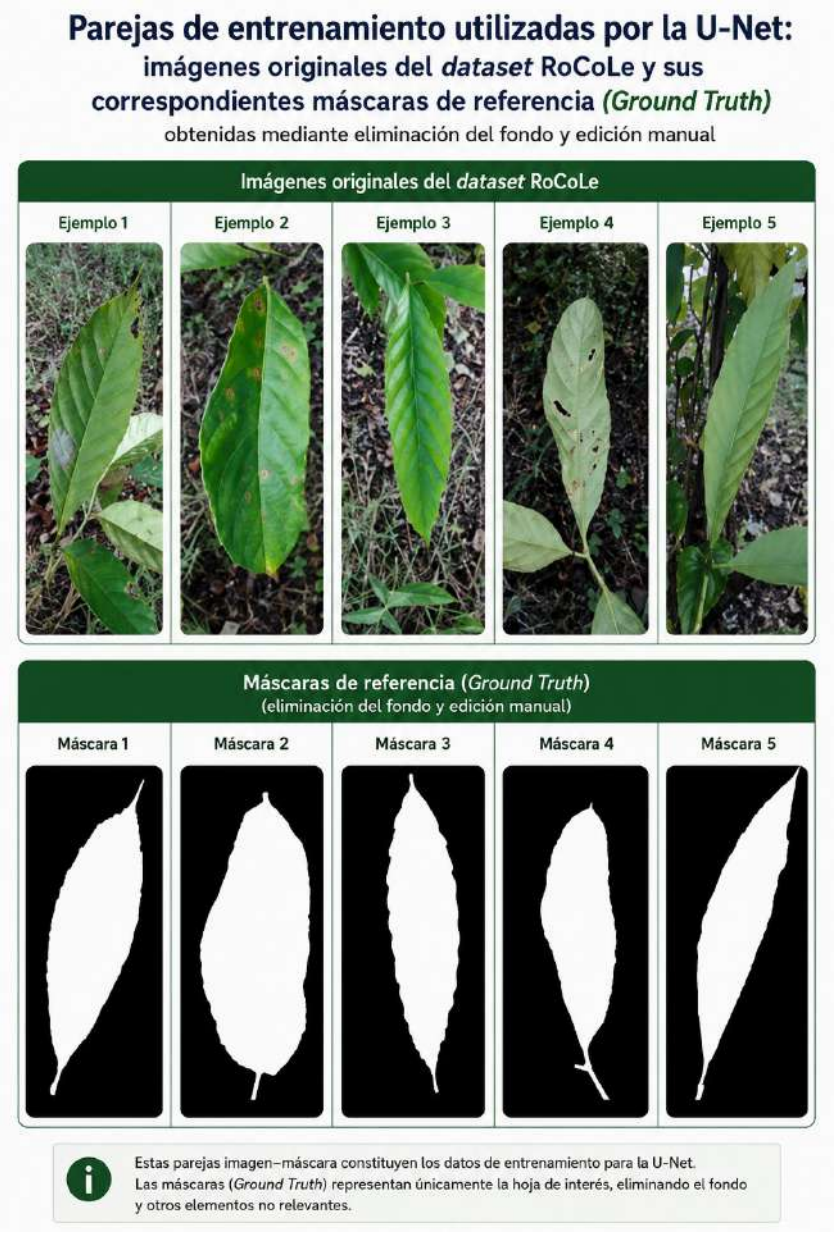


Figura 109. Parejas de entrenamiento utilizadas por la U-Net: imágenes originales del *dataset* RoCoLe y sus correspondientes máscaras de referencia (*Ground Truth*) obtenidas mediante eliminación del fondo y edición manual.

La **Figura 109** presenta las máscaras de referencia (Ground Truth) obtenidas al **finalizar la Sección 1 del código**, correspondiente a la etapa de Lectura y Preprocesamiento de las Imágenes. Es importante destacar que estas no corresponden exactamente a las imágenes editadas manualmente mostradas en la Figura 102. Dichas imágenes constituyen únicamente el punto de partida. Durante el preprocesamiento, el código identifica automáticamente la región de la hoja (a partir del canal alfa o del fondo blanco), genera una máscara binaria, la redimensiona a 224 × 224 píxeles y la convierte en una imagen de un solo canal, donde el valor 1 representa la hoja y el valor 0 el fondo. **Estas son las máscaras que finalmente utiliza la U-Net durante el entrenamiento**.

Posteriormente, obtenemos la información de la arquitectura, así:

Entrenamiento: (4, 224, 224, 3) (4, 224, 224, 1)
Validación: (1, 224, 224, 3) (1, 224, 224, 1)
Model: "UNet_pequeno"

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 224, 224, 3)	0	-
conv2d (Conv2D)	(None, 224, 224, 16)	448	input_layer[0][0]
conv2d_1 (Conv2D)	(None, 224, 224, 16)	2,320	conv2d[0][0]
max_pooling2d (MaxPooling2D)	(None, 112, 112, 16)	0	conv2d_1[0][0]
conv2d_2 (Conv2D)	(None, 112, 112, 32)	4,640	max_pooling2d[0]…
conv2d_3 (Conv2D)	(None, 112, 112, 32)	9,248	conv2d_2[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 56, 56, 32)	0	conv2d_3[0][0]
conv2d_4 (Conv2D)	(None, 56, 56, 64)	18,496	max_pooling2d_1[…
conv2d_5 (Conv2D)	(None, 56, 56, 64)	36,928	conv2d_4[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 28, 28, 64)	0	conv2d_5[0][0]
conv2d_6 (Conv2D)	(None, 28, 28, 128)	73,856	max_pooling2d_2[…
conv2d_7 (Conv2D)	(None, 28, 28, 128)	147,584	conv2d_6[0][0]
conv2d_transpose (Conv2DTranspose)	(None, 56, 56, 64)	32,832	conv2d_7[0][0]
concatenate (Concatenate)	(None, 56, 56, 128)	0	conv2d_transpose… conv2d_5[0][0]
conv2d_8 (Conv2D)	(None, 56, 56, 64)	73,792	concatenate[0][0]
conv2d_9 (Conv2D)	(None, 56, 56, 64)	36,928	conv2d_8[0][0]
conv2d_transpose_1 (Conv2DTranspose)	(None, 112, 112, 32)	8,224	conv2d_9[0][0]
concatenate_1 (Concatenate)	(None, 112, 112, 64)	0	conv2d_transpose… conv2d_3[0][0]
conv2d_10 (Conv2D)	(None, 112, 112, 32)	18,464	concatenate_1[0]…
conv2d_11 (Conv2D)	(None, 112, 112, 32)	9,248	conv2d_10[0][0]
conv2d_transpose_2 (Conv2DTranspose)	(None, 224, 224, 16)	2,064	conv2d_11[0][0]
concatenate_2 (Concatenate)	(None, 224, 224, 32)	0	conv2d_transpose… conv2d_1[0][0]
conv2d_12 (Conv2D)	(None, 224, 224, 16)	4,624	concatenate_2[0]…
conv2d_13 (Conv2D)	(None, 224, 224, 16)	2,320	conv2d_12[0][0]
conv2d_14 (Conv2D)	(None, 224, 224, 1)	17	conv2d_13[0][0]

Total params: 482,033 (1.84 MB)
Trainable params: 482,033 (1.84 MB)
Non-trainable params: 0 (0.00 B)

Figura 110. Resumen dela arquitectura U-Net construida por TensorFlow, mostrando las capas, las dimensiones de salida y el número de parámetros entrenables.

La **Figura 110** presenta el resumen de la arquitectura U-Net construida en la **Sección 2 del código**, correspondiente a la etapa de **Construcción de la Arquitectura**. En ella se observan las capas que conforman el codificador, el cuello de botella y el decodificador, así como las dimensiones de salida de cada una y el número de parámetros entrenables. En total, la red está compuesta por **482 mil parámetros** (aprox.), los cuales serán ajustados automáticamente durante el entrenamiento.

A continuación, analizamos los resultados obtenidos durante el entrenamiento de la U-Net, generados en la **tercera sección** del código, correspondiente al ***Proceso de Entrenamiento*.** En cada época, el modelo evalúa su desempeño tanto sobre las imágenes de entrenamiento como sobre la imagen reservada para validación. Para este ejemplo, en la última época ejecutada se obtuvo:

```
Epoch 76/150
4/4 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3s 719ms/step - dice_coef: 0.9557 - iou_coef: 0.9155 - loss: 0.0965 - val_dice_coef: 0.8591 - val_iou_coef: 0.7530 - val_loss: 0.4993 - learning_rate: 2.5000e-05
```

El **coeficiente Dice** mide el grado de coincidencia entre la máscara predicha y la máscara de referencia, tomando valores entre **0 y 1**, donde **1 representa una segmentación perfecta**. En este caso, un valor de **0.9557** indica que la U-Net logró una alta coincidencia sobre las imágenes utilizadas durante el entrenamiento.

Por su parte, el **coeficiente IoU (*Intersection over Union*)** cuantifica la proporción de solapamiento entre la región segmentada por el modelo y la región real. Al igual que el coeficiente Dice, sus valores oscilan entre **0 y 1**, siendo **1 el valor ideal**. El resultado obtenido (**0.9155**) confirma que existe una elevada correspondencia entre ambas regiones.

En cuanto a la validación, los valores de **Dice** e **IoU** son ligeramente inferiores a los obtenidos durante el entrenamiento, lo cual es esperable, ya que el modelo está siendo evaluado sobre una imagen que no participó en el proceso de aprendizaje. No obstante, la diferencia entre ambos conjuntos no es excesiva, lo que sugiere que la U-Net ha logrado una capacidad razonable de generalización, a pesar del reducido tamaño del conjunto de entrenamiento.

En este ejemplo se fijó una semilla aleatoria para favorecer la reproducibilidad del experimento. Por ello, si el código se ejecuta nuevamente bajo las mismas condiciones de software y hardware, es esperable obtener resultados muy similares. Sin embargo, pequeñas diferencias pueden aparecer al utilizar versiones distintas de TensorFlow, diferentes dispositivos de procesamiento (CPU o GPU) u otros entornos de ejecución.

Adicionalmente, obtenemos la gráfica de evolución del entrenamiento:

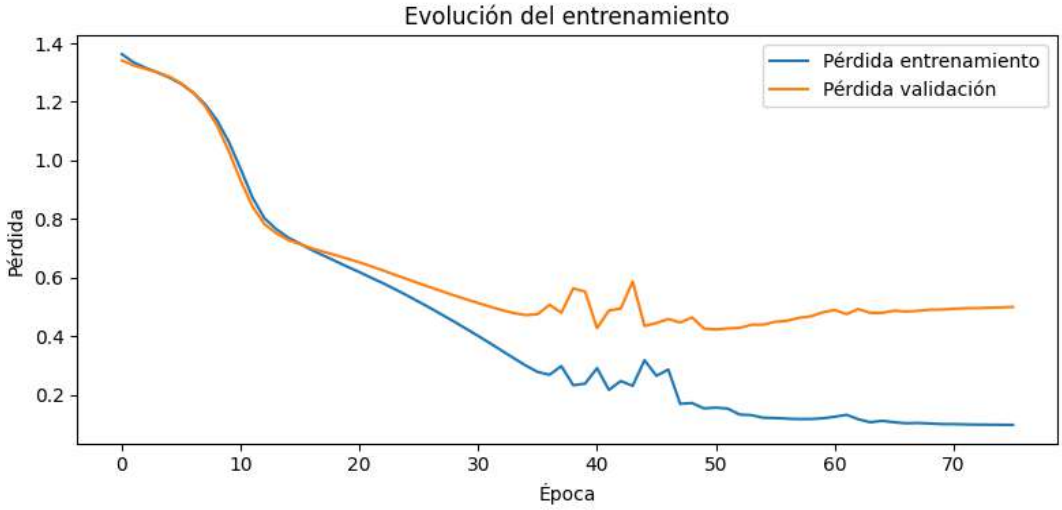


Figura 111. Evolución del entrenamiento de la U-Net.

La **Figura 111** presenta la evolución de la función de pérdida durante el entrenamiento de la U-Net para los conjuntos de entrenamiento y validación. En ambos casos se observa una disminución progresiva del error durante las primeras épocas, indicando que la red aprende gradualmente a segmentar la hoja principal. La pérdida comienza alrededor de **1.35** y disminuye hasta valores cercanos a **0.10** para entrenamiento y **0.43** para validación.

Recordemos que el **valor ideal de la función de pérdida es 0**, ya que indicaría una coincidencia perfecta entre la máscara predicha y la máscara de referencia. En este ejemplo, a partir de aproximadamente la **época 35**, la pérdida de validación deja de disminuir y comienza a estabilizarse, motivo por el cual el

criterio de **EarlyStopping** detiene automáticamente el entrenamiento antes de completar las 150 épocas programadas.

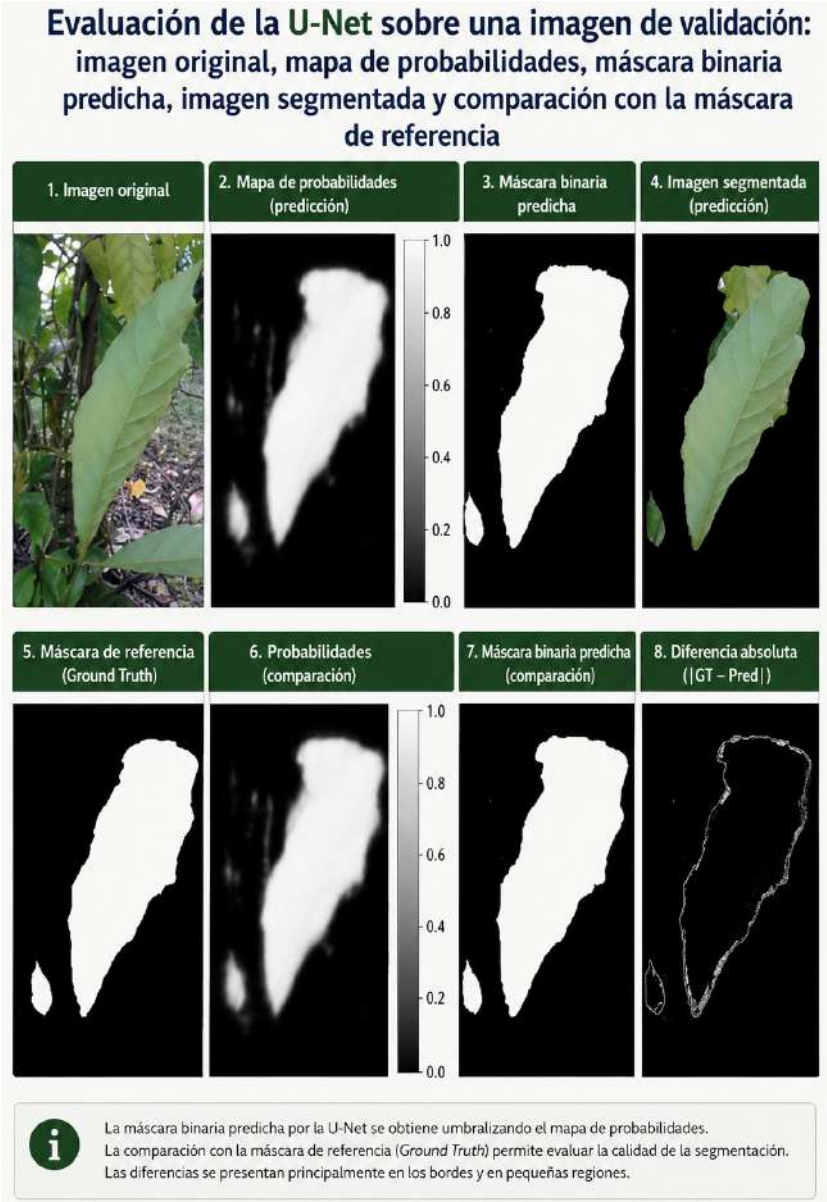


Figura 112. Evaluación de la U-Net sobre una imagen de validación: imagen original, mapa de probabilidades, máscara binaria predicha, imagen segmentada y comparación con la máscara de referencia.

La **Figura 112** presenta el resultado obtenido durante la etapa de evaluación, correspondiente a la **cuarta sección** del código: **Evaluación del Modelo**. Inicialmente, la U-Net recibe como entrada una imagen del *dataset* RoCoLe que no participó en el entrenamiento. Como salida, la red genera una **imagen de probabilidades**, donde los píxeles más claros indican una mayor probabilidad de pertenecer a la hoja principal y los más oscuros corresponden al fondo. Posteriormente, mediante la aplicación de un umbral, esta imagen se convierte en una **máscara binaria**, la cual se utiliza para eliminar el fondo y obtener la imagen segmentada.

En la parte inferior de la figura se compara la **máscara de referencia (*Ground Truth*)** con la máscara predicha por la U-Net. Aunque la segmentación no es perfecta, el modelo logra identificar correctamente la mayor parte de la hoja principal, conservando su forma general y eliminando gran parte del fondo. Considerando que la red fue entrenada únicamente con cuatro imágenes y evaluada sobre una quinta imagen no vista previamente, este resultado demuestra la capacidad de las arquitecturas U-Net para aprender la tarea de segmentación incluso a partir de conjuntos de entrenamiento muy reducidos.

6.5. Comparación entre la segmentación tradicional y la segmentación mediante U-Net

Para finalizar este capítulo, compararemos los resultados obtenidos mediante la mejor estrategia de segmentación tradicional desarrollada en el Capítulo 5 (Solución 3) con los obtenidos utilizando una U-Net entrenada con las mismas imágenes. Aunque ambos enfoques persiguen el mismo objetivo (aislar la hoja principal del fondo), la forma en que lo consiguen es completamente diferente.

La segmentación tradicional se basa en un conjunto de reglas definidas explícitamente por el programador, tales como umbrales de color, operaciones morfológicas, análisis de componentes conectados y criterios basados en el área y la posición de las regiones. En contraste, la U-Net no recibe estas reglas de manera explícita, sino que las aprende automáticamente a partir de ejemplos de entrenamiento (Ver Figura 113).

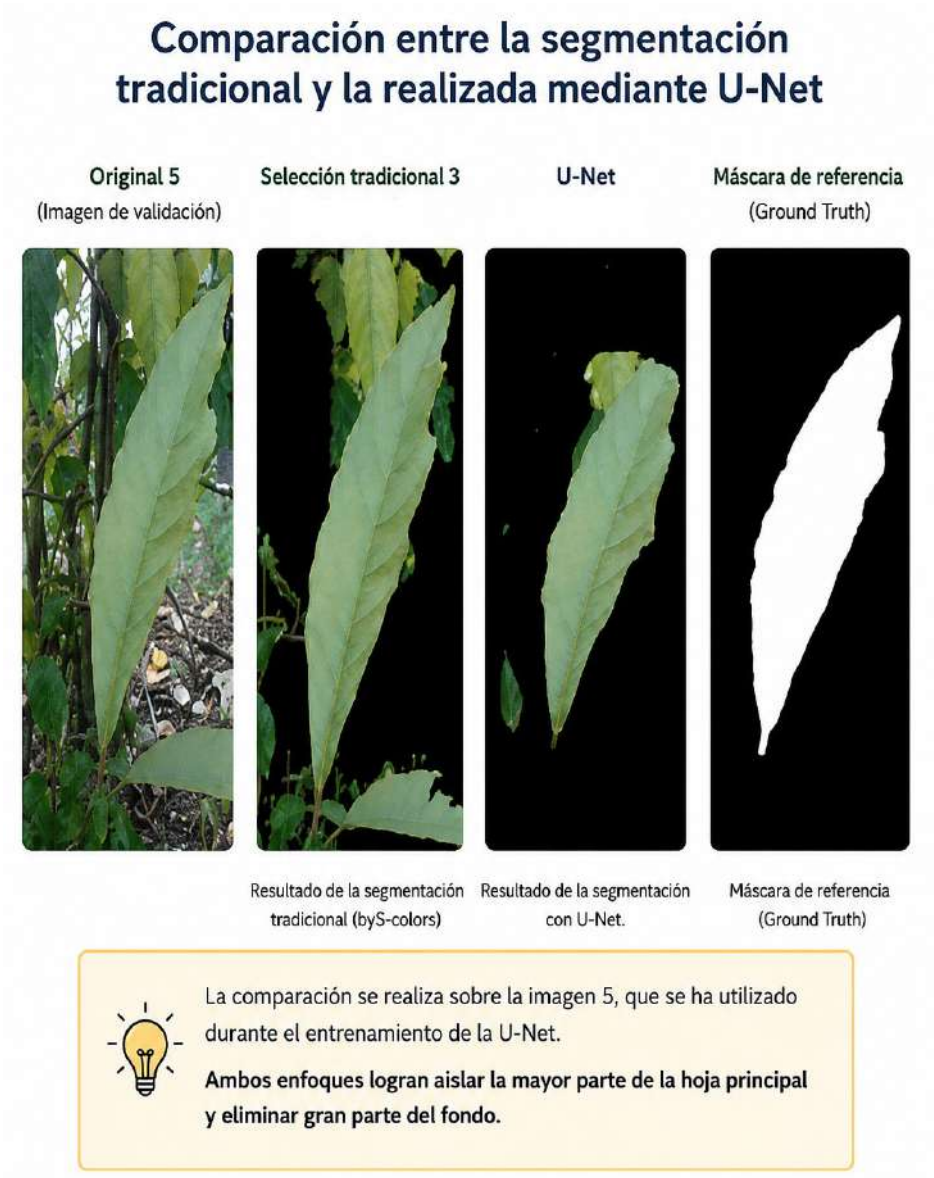


Figura 113. Comparación entre la segmentación tradicional y la realizada mediante U-Net. De izquierda a derecha: imagen original, imagen segmentada con la solución 3 del Capítulo 5, imagen segmentada con U-Net (con 4 imágenes de entrenamiento), y máscara de referencia para U-Net.

Vamos a comparar, en primer lugar, los resultados desde un punto de vista cualitativo. En el caso del enfoque tradicional, la imagen segmentada logró eliminar gran parte del fondo, generando un *background* negro; sin embargo, conservó algunas hojas secundarias cercanas a la hoja principal, especialmente en la parte superior central y en la esquina inferior derecha de la imagen. Por su parte, la U-Net eliminó una mayor proporción del fondo y únicamente conservó pequeños fragmentos de hojas secundarias. No obstante, también perdió una pequeña parte de la hoja principal, correspondiente a la punta ubicada en el extremo superior.

Ninguna de las dos soluciones produjo una segmentación perfecta. Sin embargo, es importante tener en cuenta que la U-Net fue entrenada únicamente con **cuatro parejas de imágenes** (imagen original–máscara de referencia), una cantidad extremadamente reducida para un modelo de aprendizaje profundo. Es de esperarse que, al incrementar el número de ejemplos de entrenamiento, por ejemplo, a **80 imágenes para entrenamiento y 20 para validación**, el modelo logre una mejor capacidad de generalización y, en consecuencia, produzca segmentaciones más precisas.

De esta forma, queda claro que el ejemplo desarrollado en este capítulo tiene un propósito principalmente didáctico. En un proyecto real sería necesario trabajar con un conjunto de entrenamiento mucho más amplio y diverso, que permitiera explotar plenamente el potencial de la U-Net y obtener modelos con una mayor capacidad de generalización. Asimismo, la comparación entre el enfoque tradicional y el basado en aprendizaje profundo no debería limitarse a una evaluación visual, sino complementarse con métricas objetivas, como el **coeficiente Dice** y el **índice IoU**, calculadas sobre un conjunto suficientemente representativo de imágenes de prueba.

6.6. ¿Qué aprendimos en esta sección?

Con este capítulo llegamos al final de nuestro recorrido por el procesamiento digital de imágenes. Durante estas páginas descubrimos que una imagen es mucho más que una fotografía: es una fuente de información que puede analizarse, transformarse y utilizarse para resolver problemas reales de ingeniería. Aprendimos que detrás de una buena solución no existe un único

algoritmo, sino una secuencia de decisiones fundamentadas, donde cada etapa influye en el resultado final.

También comprobamos que una misma situación puede abordarse desde perspectivas completamente diferentes. Primero construimos una solución utilizando técnicas clásicas de procesamiento digital de imágenes, diseñando explícitamente cada regla del algoritmo. Después resolvimos exactamente el mismo problema mediante Inteligencia Artificial, permitiendo que una arquitectura U-Net aprendiera automáticamente a segmentar la imagen a partir de ejemplos. Ninguno de estos enfoques es universalmente mejor que el otro; cada uno aporta fortalezas diferentes y, en muchos casos, la mejor solución surge precisamente de combinarlos.

Espero que, al terminar esta sección, no solo hayas aprendido nuevas técnicas de procesamiento digital de imágenes o Inteligencia Artificial. Espero, sobre todo, que hayas fortalecido una forma de pensar.

Porque las herramientas cambiarán, aparecerán nuevos algoritmos y nuevas arquitecturas, pero la capacidad de observar un problema, comprenderlo, plantear una estrategia, evaluar críticamente los resultados y seguir mejorándolos seguirá siendo una de las cualidades que distinguen a un buen ingeniero.