EJERCICIOS, PROBLEMAS, CASOS PRÁCTICOS

Universidad de Jaén

Escuela Politécnica Superior de Jaén

Procesamiento de información visual: problemas resueltos mediante Python y OpenCV

Francisco Jesús Martínez Mimbrera

12/01/2026

Procesamiento de información visual

Universidad de Jaén

Procesamiento de Información Visual: Problemas resueltos mediante Python y OpenCV

3º Grado en Ingeniería Informática

Francisco J. Martínez Mimbrera

Curso 2025-2026

Título: Procesamiento de Información Visual: Problemas resueltos mediante Python y OpenCV

Autor: Francisco J. Martínez Mimbrera

Versión: 1

Fecha: 12 de enero de 2026

Este manual ha sido desarrollado y maquetado utilizando LaTeX 2_ε.

Índice general

Tema 1

Introducción

La visión artificial (*Computer Vision*) ha pasado de ser un campo exclusivo de la investigación académica a una herramienta fundamental en la industria moderna. Desde el reconocimiento facial en nuestros teléfonos hasta la navegación de vehículos autónomos, la capacidad de las máquinas para "ver" e interpretar el mundo es hoy una realidad cotidiana [1]. En el corazón de esta revolución se encuentra una biblioteca: **OpenCV**.

En este capítulo, abordaremos un poco de historia de la relación de Python con la librería OpenCV y cómo utilizarla en un proyecto.

1.1. Un poco de historia entre Python y OpenCV: De Intel al Open Source

El lenguaje de programación Python ha ido evolucionando a lo largo de los años desde su aparición a finales de la década de 1980. Fue diseñado por Guido van Rossum [2] en 1989 como sucesor del lenguaje ABC para el sistema operativo Amoeba [3]). El nombre «Python» no tiene relación con reptiles, sino que fue elegido por la afición de Van Rossum al grupo de comedia británico Monty Python's Flying Circus [2]. Python 1.0 fue lanzada en enero de 1994 [4] y marcó un hito fundamental en su desarrollo. Python es gestionado por la Python Software Foundation (PSF) a través de los Python Enhancement Proposals (PEP). En la actualidad, Python 3.14 es la versión estable de referencia (lanzada a finales de 2025) [5].

OpenCV (*Open Source Computer Vision Library*) [6] nació en 1999 en los laboratorios de Intel. El proyecto fue liderado inicialmente por Gary Bradski con un objetivo claro: proporcionar una infraestructura de software común y optimizada que acelerara la investigación en visión artificial y permitiera sacar partido a la creciente potencia de los procesadores de la época [6] [7].

La primera versión alfa se lanzó al público en el año 2000 en la conferencia CVPR, y desde entonces su crecimiento ha sido exponencial. Aunque el núcleo de la biblioteca está escrito en C/C++ para garantizar la máxima eficiencia computacional, su evolución ha sido constante [7]:

- **Versión 1.x:** Basada principalmente en C.
- **Versión 2.x:** Introdujo una interfaz C++ más moderna.
- **Era actual (3.x, 4.x y 5.x):** Se consolidó el soporte para Python, Deep Learning y aceleración por GPU (CUDA/OpenCL).

Hoy en día, OpenCV es un proyecto de código abierto (bajo licencia Apache 2), mantenido por una comunidad global y utilizado por gigantes tecnológicos como Google, Microsoft, Intel y miles de startups.

1.2. ¿Por qué OpenCV con Python?

Aunque OpenCV está escrito en C++, tal y como comentamos anteriormente, su unión con Python es probablemente la más popular en la comunidad educativa y de prototipado rápido. Esto se debe a que la interfaz de Python para OpenCV actúa como un "envoltorio" (*wrapper*).

Cuando llamamos a una función en Python, el cálculo pesado se realiza en el código nativo de C++ en segundo plano. Además, en Python, las imágenes en OpenCV se representan como arreglos de **NumPy** (matrices multidimensionales) [8], lo que permite integrar operaciones matemáticas complejas de forma intuitiva y eficiente.

1.3. Uso en los ejercicios

Para seguir los problemas resueltos de este manual, necesitarás tener instalado Python (versión 3.10 o superior recomendada). La forma más sencilla de obtener OpenCV es a través del gestor de paquetes `pip`. Asimismo, se recomienda la lectura de libro-curso sobre Python de los profesores Dr. Arturo Montejo Ráez y Dra. Salud Jiménez Zafra [9] para el conocimiento de manejo básico del lenguaje Python. En nuestro caso, hemos utilizado la versión 3.12 por estabilidad.

Existen dos versiones principales de OpenCV que puedes instalar:

- **Versión estándar** (Recomendada para producción):

  ```
  pip install opencv-python
  ```

- **Versión completa** (Recomendada para aprendizaje):

  ```
  pip install opencv-contrib-python
  ```

  Incluye módulos adicionales ("contrib") y algoritmos patentados o en desarrollo que a menudo se usan en ejercicios académicos (como ciertos detectores de características).

Para el trabajo con este manual, se ha creado un entorno virtual conda con la versión Python 3.12

```
conda create -n ''ejerciciosPIV'' python=3.12
```

y se le han añadido las siguientes librerías.

```
pip install opencv-contrib-python numpy matplotlib tensorflow scikit-image pillow-heif pillow
```

Nota: No es necesario utilizar conda para realizar estos ejercicios, pero es recomendable, sobre todo si se trabaja con otras asignaturas y se utiliza jupyter Notebook.

Una vez instalado, la convención universal para importarlo en tu código es:

```
import cv2
import numpy as np
from PIL import Image, ExifTags
```

Los ejercicios resueltos se encuentran en la siguiente dirección en 2 formatos: Archivos .py separados y jupyterNotebook.

https://github.com/franmartinezmimbrera/problemaspiv

A lo largo de las siguientes páginas, utilizaremos esta configuración para diseccionar imágenes, manipular píxeles, aplicar filtros y entender cómo las matemáticas se transforman en información visual.

Tema 2

Ejercicios resueltos de procesamiento básico de imágenes con OpenCV y Python

En este capítulo vamos a presentar una serie de ejercicios básicos sobre procesamiento de imágenes utilizando Python y la biblioteca OpenCV. El objetivo principal es que el alumnado pueda afianzar de forma progresiva los contenidos vistos en las clases teóricas mediante la resolución de problemas prácticos, claramente estructurados y con dificultad creciente.

A través de estos ejercicios, el estudiante aprenderá a realizar operaciones fundamentales como cargar, mostrar y analizar imágenes, así como aplicar transformaciones clásicas de preprocesamiento: conversión a escala de grises, filtrados, recortes, rotaciones, ajustes de brillo y contraste, entre otras. Además, se introducirán tareas esenciales para comprender el funcionamiento interno de las imágenes digitales, tales como la manipulación de canales de color, el trabajo con regiones de interés (ROI), la lectura de metadatos o la conversión entre distintos espacios de color.

La intención es que, mediante estas actividades, el alumno pueda "aterrizar" lo aprendido en teoría y desarrollar habilidades prácticas que le permitan abordar problemas reales de visión por computador. Cada ejercicio está diseñado para ser autónomo, claro y fácilmente extensible, fomentando así la experimentación y el aprendizaje activo. Este capítulo servirá como base para prácticas más complejas en capítulos posteriores, donde se aplicarán técnicas avanzadas o combinadas para resolver tareas de mayor nivel.

2.1. Leer, mostrar e imprimir información sobre una imagen

En este ejercicio se introduce el proceso fundamental de carga y visualización de imágenes utilizando OpenCV. Además, se muestra cómo acceder a información básica como dimensiones, tipo de datos y número de canales, lo cual es esencial para cualquier tarea posterior de procesamiento.

```
#leermostrar.py
import cv2
import matplotlib.pyplot as plt
# Leer la imagen
img = cv2.imread('imagen.jpg')
# Mostrar la imagen
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
# Imprimir información sobre la imagen
print(f'Tamaño de la imagen: {img.shape}')
```

2.2. Convertir una imagen a escala de grises

Aquí se practica la conversión de una imagen a escala de grises, una operación muy común que reduce la complejidad del procesamiento al trabajar con un único canal de intensidad en lugar de tres canales de color.

```
#convertiragrises.py
import cv2
import matplotlib.pyplot as plt

# Leer la imagen original
img = cv2.imread('imagen.jpg')

#  Convertir a escala de grises
gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Configurar el lienzo para 2 imágenes una al lado de la otra
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))

# Convertir BGR de OpenCV a RGB de Matplotlib
ax1.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax1.set_title('Original (Color)')
ax1.axis('off')

ax2.imshow(gray_img, cmap='gray')
ax2.set_title('Escala de Grises')
ax2.axis('off')

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

2.3. Aplicar un filtro Gaussiano

Este ejercicio enseña a suavizar una imagen mediante un filtro Gaussiano, técnica utilizada para reducir ruido y eliminar detalles finos, preparando la imagen para etapas posteriores como detección de bordes o segmentación. **Nota:** Habrá una sección dedicada para estos filtros en particular.

```
#aplicargaussiano.py
import cv2
import matplotlib.pyplot as plt

#  Leer la imagen
img = cv2.imread('imagen.jpg')

# 2. Aplicar filtro gaussiano. (35, 35) es el tamaño del kernel (debe
   ser impar). A mayor número, más desenfoque.
gaussian_blur = cv2.GaussianBlur(img, (35, 35), 0)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

ax1.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax1.set_title('Original (Nítida)')
ax1.axis('off')

ax2.imshow(cv2.cvtColor(gaussian_blur, cv2.COLOR_BGR2RGB))
ax2.set_title('Desenfoque Gaussiano (Kernel 35x35)')
ax2.axis('off')

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

2.4. Realizar un recorte en una imagen

En este apartado el alumno aprende a seleccionar y extraer una porción rectangular de una imagen. El recorte es útil para limitar el área de procesamiento o aislar regiones de interés.

```
#recortar.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg')
alto, ancho, _ = img.shape

# Calcular el punto central de la imagen
centro_y = alto // 2
centro_x = ancho // 2

# Definir el tamaño del recorte deseado (un cuadrado de lado X lado)
lado_recorte = 300
radio = lado_recorte // 2

# Calcular las coordenadas de inicio (arriba/izquierda) y fin
   (abajo/derecha)
inicio_y = int(centro_y - radio)
fin_y    = int(centro_y + radio)
inicio_x = int(centro_x - radio)
fin_x    = int(centro_x + radio)

# Las imágenes en OpenCV son matrices de NumPy.
```

```
# El recorte se hace así: imagen[filas_inicio:filas_fin,
   columnas_inicio:columnas_fin]
# Es decir: imagen[Y_inicio:Y_fin, X_inicio:X_fin]
cropped_img = img[inicio_y:fin_y, inicio_x:fin_x]

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

# Imagen 1: Original completa
ax1.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax1.set_title(f'Original ({ancho}x{alto})')

# Imagen 2: Recorte central
if cropped_img.size > 0:
    ax2.imshow(cv2.cvtColor(cropped_img, cv2.COLOR_BGR2RGB))
    ax2.set_title(f'Recorte Central ({lado_recorte}x{lado_recorte})')
    ax2.axis('off')
else:
    print("Error: La imagen original es más pequeña que el tamaño de
       recorte deseado.")
    ax2.set_title('Error al recortar')

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

2.5. Rotar una imagen

Este ejercicio muestra cómo aplicar una rotación simple a una imagen alrededor de su centro, empleando funciones de transformación geométrica de OpenCV.

```
#rotar.py
import cv2
import matplotlib.pyplot as plt

# Leer la imagen
img = cv2.imread('imagen.jpg')
# Obtener las dimensiones de la imagen
(h, w) = img.shape[:2]

# Calcular el centro de la imagen
center = (w // 2, h // 2)

# Generar la matriz de rotación
M = cv2.getRotationMatrix2D(center, 90, 1.0)

# Realizar la rotación
rotated_img = cv2.warpAffine(img, M, (w, h))

# Mostrar la imagen rotada
plt.imshow(cv2.cvtColor(rotated_img, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```

2.6. Invertir colores de una imagen

Aquí se realiza la inversión de los valores de color de cada píxel, produciendo un efecto negativo. Este tipo de manipulación permite comprender operaciones punto a punto en imágenes.

```
#invertircolores.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg')
# La función bitwise_not invierte cada bit de cada píxel.
img_invertida = cv2.bitwise_not(img)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

ax1.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax1.set_title('Original')
ax1.axis('off')

ax2.imshow(cv2.cvtColor(img_invertida, cv2.COLOR_BGR2RGB))
ax2.set_title('Invertida (Negativo)')
ax2.axis('off')

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

2.7. Ajustar el brillo y contraste de una imagen

En este ejercicio se modifican los niveles de brillo y contraste aplicando transformaciones lineales sobre los píxeles. Es una operación esencial para mejorar la calidad visual o preparar imágenes para análisis.

```
#ajustarbrillo.py
import cv2
import matplotlib.pyplot as plt
# Cargar la imagen
img = cv2.imread('imagen.jpg')

# Ajustar el brillo (beta) y contraste (alpha)
# Usando la fórmula: g(x) = alpha * f(x) + beta
alpha = 1.5
beta = 50
img_ajustada = cv2.convertScaleAbs(img, alpha=alpha, beta=beta)

# Creamos una figura con 1 fila y 2 columnas
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

# Imagen 1: Original
ax1.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax1.set_title('Original')
ax1.axis('off')

# Imagen 2: Ajustada
ax2.imshow(cv2.cvtColor(img_ajustada, cv2.COLOR_BGR2RGB))
ax2.set_title(f'Ajustada (alpha={alpha}, beta={beta})')
ax2.axis('off')

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

2.8. Recortar una región de interés (ROI) en una imagen

Se muestra cómo definir explícitamente una región de interés mediante coordenadas y extraerla. Este ejercicio refuerza el trabajo con índices y la selección precisa de áreas dentro de la imagen.

```
#recortarregioROI.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg')
#  Definir la región de interés (ROI)
x, y, w, h = 2000, 1400, 700, 700
# 3. Recortar la ROI
roi = img[y:y+h, x:x+w]

img_con_cuadro = img.copy()
cv2.rectangle(img_con_cuadro, (x, y), (x+w, y+h), (0, 255, 0), 5) #
    Cuadro verde

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

ax1.imshow(cv2.cvtColor(img_con_cuadro, cv2.COLOR_BGR2RGB))
ax1.set_title('Imagen Original (Zona marcada)')
ax1.axis('on')

ax2.imshow(cv2.cvtColor(roi, cv2.COLOR_BGR2RGB))
ax2.set_title(f'ROI: {w}x{h} píxeles')
ax2.axis('off')

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

2.9. Rotar una imagen en un ángulo arbitrario

A diferencia de la rotación simple, aquí se permite especificar cualquier ángulo, mostrando el uso de matrices de transformación e interpolación para obtener un resultado correcto.

```
#rotarimagenarbitrario.py
import cv2
import matplotlib.pyplot as plt
img = cv2.imread('imagen.jpg')
# Configurar la rotación
angulo = 45  # grados (sentido antihorario)
(h, w) = img.shape[:2]
(cX, cY) = (w // 2, h // 2) # Centro de la imagen
# Calcular la matriz de rotación 2D
M = cv2.getRotationMatrix2D((cX, cY), angulo, 1.0)
# Aplicar la transformación (Warp Affine)
img_rotada = cv2.warpAffine(img, M, (w, h))
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))
ax1.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax1.set_title('Original')
ax1.axis('off')
ax2.imshow(cv2.cvtColor(img_rotada, cv2.COLOR_BGR2RGB))
ax2.set_title(f'Rotada {angulo} grados')
ax2.axis('off')
plt.tight_layout()
plt.show()
```

2.10. Reflejar una imagen horizontal o verticalmente

Este ejercicio enseña a aplicar reflexiones o espejados horizontales y verticales, operaciones útiles en aumentación de datos y en análisis de simetrías.

```
#reflejarimagen.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg')

img_reflejada_h = cv2.flip(img, 1)
img_reflejada_v = cv2.flip(img, 0)

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))

ax1.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax1.set_title('Original')
ax1.axis('off')

ax2.imshow(cv2.cvtColor(img_reflejada_h, cv2.COLOR_BGR2RGB))
ax2.set_title('Horizontal (flipCode=1)')
ax2.axis('off')

ax3.imshow(cv2.cvtColor(img_reflejada_v, cv2.COLOR_BGR2RGB))
ax3.set_title('Vertical (flipCode=0)')
ax3.axis('off')

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

2.11. Cargar y mostrar información de metadatos de una imagen

En este apartado se muestra cómo cargar una imagen y extraer información EXIF y otros metadatos relevantes, tales como el modelo de cámara, fecha de captura o parámetros de disparo, cuando están disponibles.

```
from PIL import Image, ExifTags
import cv2
import matplotlib.pyplot as plt
# Usaremos PIL (necesario para metadatos EXIF)
imagen_pil = Image.open(ruta_imagen)
print(f"Formato: {imagen_pil.format}")
print(f"Tamaño (ancho x alto): {imagen_pil.size}")
print(f"Modo de color: {imagen_pil.mode}")
exif_data = imagen_pil._getexif()
print("Información mediante librería PIL ")
if exif_data is not None:
    for etiqueta, valor in exif_data.items():
            nombre = ExifTags.TAGS.get(etiqueta, etiqueta)
            print(f"{nombre}: {valor}")
else:
    print("La imagen no contiene metadatos EXIF.")

imagen_cv = cv2.imread(ruta_imagen)
if imagen_cv is not None:
    alto, ancho, canales = imagen_cv.shape
    print("Información mediante librería OpenCV")
```

```
    print(f"Resolución: {ancho} x {alto} píxeles")
    print(f"Número de canales: {canales}")
    print(f"Tipo de datos: {imagen_cv.dtype}")
else:
    print("\nOpenCV no pudo cargar la imagen.")
plt.imshow(cv2.cvtColor(imagen_cv, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```

2.12. Convertir una imagen a un espacio de color diferente

Aquí se explora la conversión entre distintos espacios de color (RGB, HSV, LAB, etc.), una operación fundamental para análisis avanzados como segmentación o detección basada en color.

```
#convertirespacio.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg')
img_hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

ax1.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax1.set_title('Original (Espacio RGB)')
ax1.axis('off')

ax2.imshow(img_hsv)
ax2.set_title('Representación HSV (Colores falsos)')
ax2.axis('off')

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

2.13. Aplicar un efecto de desenfoque a una imagen

En este ejercicio se aplican técnicas de desenfoque básico, permitiendo comprender cómo la convolución suaviza la imagen y reduce detalles o ruido.

```
#desenfoque.py
import cv2
import matplotlib.pyplot as plt
img = cv2.imread('imagen.jpg')
kernel_size = (5, 5)
sigma = 1.4
img_desenfocada = cv2.GaussianBlur(img, kernel_size, sigma)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))
ax1.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax1.set_title('Original (Nítida)')
ax1.axis('off')
ax2.imshow(cv2.cvtColor(img_desenfocada, cv2.COLOR_BGR2RGB))
ax2.set_title(f'Desenfoque (Sigma={sigma})')
ax2.axis('off')
plt.tight_layout()
plt.show()
```

2.14. Crear una imagen vacía y dibujar formas geométricas básicas

Se enseña a generar una imagen desde cero y dibujar sobre ella líneas, círculos o rectángulos, introduciendo las funciones de dibujo de OpenCV.

```
#imagenvacia.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

# Crear una imagen vacía
img = np.zeros((512, 512, 3), dtype=np.uint8)

# Dibujar un círculo en la imagen
cv2.circle(img, (256, 256), 100, (255, 0, 0), -1)

# Dibujar una línea en la imagen
cv2.line(img, (100, 100), (400, 400), (0, 255, 0), 5)

# Dibujar un rectángulo en la imagen
cv2.rectangle(img, (200, 200), (300, 300), (0, 0, 255), -1)

# Mostrar la imagen
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```

2.15. Dividir una imagen en canales de color individuales

Este ejercicio permite visualizar los canales individuales de una imagen (por ejemplo, B, G y R), lo cual es útil para entender cómo está compuesta y para realizar operaciones sobre canales específicos.

```
#dividir.py
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg')

b, g, r = cv2.split(img)

plt.figure(figsize=(12,4))
plt.subplot(131); plt.imshow(b, cmap='Blues'); plt.axis('off')
plt.subplot(132); plt.imshow(g, cmap='Greens'); plt.axis('off')
plt.subplot(133); plt.imshow(r, cmap='Reds'); plt.axis('off')
plt.show()
```

Tema 3

Ejercicios resueltos de cálculo y visualización de histogramas

En este capítulo se abordará el estudio de los histogramas, una de las herramientas fundamentales en el procesamiento digital de imágenes. Un **histograma** es una representación gráfica que muestra la distribución de las intensidades de los píxeles, permitiendo analizar de manera visual y cuantitativa aspectos críticos como el contraste, el brillo y el rango dinámico [1].

A lo largo de este bloque práctico, exploraremos desde la generación básica de histogramas en escala de grises y canales RGB, hasta técnicas avanzadas que incluyen el uso de diversos espacios de color (HSV, CIELAB), el cálculo de descriptores estadísticos y el análisis dinámico en tiempo real. Asimismo, se introducirán métodos para la comparación de imágenes mediante métricas de distancia y la inspección de regiones de interés (ROI) mediante máscaras, proporcionando una base sólida para el diagnóstico y análisis de imágenes antes de proceder a técnicas de filtrado o mejora.

3.1. Calcular e imprimir el histograma en escala de grises.

El primer paso para comprender la composición de una imagen es analizar su distribución de luminancia. En esta sección se presenta un script que transforma una imagen a escala de grises y genera un histograma unidimensional. Este gráfico permite observar la frecuencia de aparición de cada nivel de intensidad, desde el negro absoluto (0) hasta el blanco puro (255).

```
#histogramagris.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg')
gris = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
#  Calcular el histograma
hist = cv2.calcHist([gris], [0], None, [256], [0, 256])

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

ax1.imshow(gris, cmap='gray')
ax1.set_title('Imagen en Grises')
ax1.axis('off')

ax2.plot(hist, color='black')
ax2.set_title('Histograma de Intensidad')
ax2.set_xlim([0, 256])
ax2.grid(True)

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

3.2. Calcular e imprimir los histogramas de cada canal de color

En las imágenes en color, la información se divide habitualmente en tres canales: Rojo, Verde y Azul (RGB). En este apartado, desarrollaremos un código capaz de descomponer la imagen original en sus componentes básicos para visualizar sus respectivos histogramas. Esta técnica es vital para identificar sesgos cromáticos o dominantes de color en una fotografía.

```
#histogramacolores.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg')
# Separar en canales
chans = cv2.split(img)

colors = ("b", "g", "r")
plt.figure()
plt.title("Histograma de Color")
plt.xlabel("Bins")
plt.ylabel("# of Pixels")
# Crear un histograma por cada canal
for (chan, color) in zip(chans, colors):
    hist = cv2.calcHist([chan], [0], None, [256], [0, 256])
    plt.plot(hist, color=color)
    plt.xlim([0, 256])

plt.show()
```

3.3. Calcular y mostrar el histograma acumulado

A diferencia del histograma convencional, el histograma acumulado suma las frecuencias de los niveles de intensidad inferiores o iguales al valor actual. Esta representación es especialmente útil para comprender la distribución global del brillo y constituye la base matemática para operaciones de mejora de imagen, como la ecualización de histogramas.

```
#histogramacumulado.py
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Leer la imagen en escala de grises
img = cv2.imread('imagen.jpg', cv2.IMREAD_GRAYSCALE)

# Calcular el histograma
hist = cv2.calcHist([img], [0], None, [256], [0, 256])

# Convertir a histograma acumulado
cum_hist = hist.cumsum()

# Normalizar el histograma acumulado
cum_hist_normalized = cum_hist / float(cum_hist.max())

# Mostrar el histograma acumulado
plt.figure()
plt.title("Histograma Acumulado")
plt.xlabel("Bins")
plt.ylabel("# of Pixels")
plt.plot(cum_hist_normalized)
plt.xlim([0, 256])
plt.show()
```

3.4. Calcular y mostrar el histograma de un área seleccionada

En muchas aplicaciones de visión artificial, nos interesa analizar únicamente una región de interés (ROI) en lugar de la imagen completa. En esta sección se muestra cómo aplicar una máscara para aislar un área específica y generar un histograma que describa únicamente el contenido de dicha zona, evitando que el fondo o elementos irrelevantes distorsionen los datos.

```
#histogramaareasele.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg', cv2.IMREAD_GRAYSCALE)
# Definir una región de interés (ROI)
roi = img[100:300, 150:400]
# Calcular el histograma del ROI
hist_roi = cv2.calcHist([roi], [0], None, [256], [0, 256])

plt.figure()
plt.title("Histograma de la ROI")
plt.xlabel("Bins")
plt.ylabel("# of Pixels")
plt.plot(hist_roi)
plt.xlim([0, 256])
plt.show()
```

3.5. Comparar dos histogramas

La comparación de histogramas permite medir el grado de similitud entre dos imágenes basándose en su distribución de color o intensidad. Aquí exploraremos cómo utilizar métricas estadísticas (como la correlación o la distancia Chi-cuadrado) para determinar qué tan parecidas son dos capturas, una técnica fundamental en sistemas de búsqueda y reconocimiento de imágenes.

```
#comparahistograma.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

img1 = cv2.imread('imagen.jpg', cv2.IMREAD_GRAYSCALE)
img2 = cv2.imread('imagenbordes.jpg', cv2.IMREAD_GRAYSCALE)

# Calcular los histogramas
hist1 = cv2.calcHist([img1], [0], None, [256], [0, 256])
hist2 = cv2.calcHist([img2], [0], None, [256], [0, 256])
# Normalizar los histogramas
cv2.normalize(hist1, hist1)
cv2.normalize(hist2, hist2)

# Comparar los histogramas usando Correlación
comparison = cv2.compareHist(hist1, hist2, cv2.HISTCMP_CORREL)
print(f'Correlación: {comparison}')
```

3.6. Histograma en el espacio de color HSV (Canal de Matiz)

A diferencia del espacio RGB, el modelo HSV separa la información cromática (Matiz o Hue) de la intensidad. En este ejercicio se calcula el histograma del canal H, lo cual es fundamental para tareas de segmentación basadas únicamente en el color, independientemente de la iluminación.

```
#histograma_hsv_hue.py
import cv2
import matplotlib.pyplot as plt

image = cv2.imread('imagen.jpg')
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
# El canal H (Hue) representa el color puro
hist_hue = cv2.calcHist([hsv], [0], None, [180], [0, 180])

plt.plot(hist_hue, color='orange')
plt.title('Histograma del Canal H (Matiz)')
plt.xlabel('Tono')
plt.ylabel('Frecuencia')
plt.show()
```

3.7. Visualización de histogramas RGB superpuestos

Para facilitar la comparación visual entre los canales Rojo, Verde y Azul, es común graficarlos en una sola ventana utilizando transparencias. Este ejercicio muestra cómo utilizar la librería Matplotlib para superponer las tres gráficas, permitiendo identificar rápidamente cuál es el color dominante en la escena.

```
#histograma_rgb_superpuesto.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg')
colores = ('b', 'g', 'r')

for i, col in enumerate(colores):
    hist = cv2.calcHist([img], [i], None, [256], [0, 256])
    plt.plot(hist, color=col, alpha=0.7, label=f'Canal {col.upper()}')
    plt.fill_between(range(256), hist.flatten(), color=col, alpha=0.1)

plt.title('Histogramas RGB Superpuestos')
plt.legend()
plt.show()
```

3.8. Histograma bidimensional (2D) de los canales H y S

Un histograma 2D analiza la relación entre dos canales simultáneamente. En este caso, relacionaremos el Matiz (Hue) y la Saturación (Saturation). El resultado es un mapa de calor que indica qué colores están presentes y con qué pureza, una herramienta poderosa para el seguimiento de objetos por color.

```
#histograma_2d_hs.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg')
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

# Calculamos histograma para H (0) y S (1) simultáneamente
hist2d = cv2.calcHist([hsv], [0, 1], None, [180, 256], [0, 180, 0, 256])

plt.imshow(hist2d, interpolation='nearest', origin='lower', extent=[0,
    256, 0, 180])
plt.colorbar(label='Frecuencia')
plt.xlabel('Saturación')
plt.ylabel('Matiz (Hue)')
plt.title('Histograma 2D H-S')
plt.show()
```

3.9. Cálculo de parámetros estadísticos del histograma

Más allá de la representación gráfica, el histograma puede resumirse en valores numéricos. En este ejercicio calcularemos la media, la desviación estándar y la varianza de las intensidades de los píxeles, lo que nos permite describir matemáticamente si una imagen es çlara", .ºscura.º si tiene "mucho o poco contraste".

```
#estadisticas_histograma.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

img = cv2.imread('imagen.jpg', cv2.IMREAD_GRAYSCALE)

media = np.mean(img)
desviacion = np.std(img)
varianza = np.var(img)
min_val, max_val, _, _ = cv2.minMaxLoc(img)

print(f"Estadísticas de la imagen:")
print(f"Media de intensidad: {media:.2f}")
print(f"Desviación Estándar: {desviacion:.2f}")
print(f"Varianza: {varianza:.2f}")
print(f"Rango de valores: [{min_val}, {max_val}]")
```

3.10. Histograma dinámico en tiempo real mediante Webcam

El análisis de imágenes no se limita a archivos estáticos. En esta sección se desarrolla un script que captura video en tiempo real desde una cámara y actualiza el histograma frame a frame, permitiendo observar cómo cambia la distribución de luz al mover la cámara o variar la iluminación ambiental.

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


cap = cv2.VideoCapture(0)
if not cap.isOpened():
    print("Error cámara")
    exit()

cv2.namedWindow("Cámara", cv2.WINDOW_NORMAL)

start = time.time()
hist_acum = np.zeros((256, 1), dtype=np.float32)

# CAPTURA 10 SEGUNDOS
while time.time() - start < 10:
    ret, frame = cap.read()
    if not ret:
        break

    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    hist = cv2.calcHist([gray], [0], None, [256], [0, 256])
    hist_acum += hist

    cv2.imshow("Cámara", frame)
```

```
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyWindow("Cámara")


hist_acum = cv2.normalize(hist_acum, None, 0, 300, cv2.NORM_MINMAX)
canvas = np.zeros((300, 256), dtype="uint8")

for i in range(1, 256):
    cv2.line(
        canvas,
        (i - 1, 300 - int(hist_acum[i - 1][0])),
        (i,     300 - int(hist_acum[i][0])),
        255, 1
    )

cv2.namedWindow("Histograma final (10s)", cv2.WINDOW_NORMAL)
cv2.imshow("Histograma final (10s)", canvas)

cv2.waitKey(0)
cv2.destroyAllWindows()

# Mostrar histograma final en el notebook
plt.figure(figsize=(8, 4))
plt.imshow(canvas, cmap="gray")
plt.title("Histograma final (10 segundos)")
plt.axis("off")
plt.show()
```

3.11. Histograma con máscara circular para enfoque central

Siguiendo con el análisis de regiones de interés (ROI), este ejercicio implementa una máscara circular. Esta técnica es común en aplicaciones de inspección donde el objeto de estudio es redondo o se encuentra ubicado específicamente en el centro del encuadre, ignorando las esquinas de la imagen.

```
#histograma_mascara_circular.py
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg', cv2.IMREAD_GRAYSCALE)
h, w = img.shape[:2]

# Crear máscara circular
mask = np.zeros(img.shape[:2], np.uint8)
cv2.circle(mask, (w//2, h//2), min(w, h)//4, 255, -1)

hist_mask = cv2.calcHist([img], [0], mask, [256], [0, 256])

plt.subplot(121), plt.imshow(cv2.bitwise_and(img, img, mask=mask),
   cmap='gray')
plt.subplot(122), plt.plot(hist_mask)
plt.show()
```

3.12. Detección de subexposición y sobreexposición mediante histograma

Una aplicación práctica del histograma es el control de calidad fotográfico. En este apartado se analiza si los píxeles se agrupan excesivamente en el valor 0 (subexposición) o en el 255 (sobreexposición o "quemado"), generando una alerta automática si la imagen ha perdido detalle en las sombras o en las luces.

```
#deteccion_exposicion.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

img = cv2.imread('imagen.jpg', cv2.IMREAD_GRAYSCALE)
hist = cv2.calcHist([img], [0], None, [256], [0, 256])

total_píxeles = img.shape[0] * img.shape[1]
subexpuestos = (hist[0] / total_píxeles) * 100
sobreexpuestos = (hist[255] / total_píxeles) * 100

print(f"Porcentaje píxeles negros: {subexpuestos[0]:.2f}%")
print(f"Porcentaje píxeles blancos: {sobreexpuestos[0]:.2f}%")

if subexpuestos > 5: print("Alerta: Posible imagen subexpuesta")
if sobreexpuestos > 5: print("Alerta: Posible imagen sobreexpuesta
    (quemada)")
```

3.13. Comparación de histogramas usando la distancia Bhattacharyya

Existen diversos algoritmos para comparar qué tan similares son dos histogramas. Mientras que la correlación es lineal, la distancia de Bhattacharyya es más robusta para medir el solapamiento entre distribuciones de probabilidad. Aquí compararemos dos imágenes para evaluar su similitud estadística.

```
#comparacion_bhattacharyya.py
import cv2
import matplotlib.pyplot as plt

img1 = cv2.imread('imagen.jpg')
img2 = cv2.imread('imagencopia.jpg') # O una imagen similar

# Convertir a HSV y calcular histogramas
h1 = cv2.calcHist([cv2.cvtColor(img1, cv2.COLOR_BGR2HSV)], [0, 1],
    None, [50, 60], [0, 180, 0, 256])
h2 = cv2.calcHist([cv2.cvtColor(img2, cv2.COLOR_BGR2HSV)], [0, 1],
    None, [50, 60], [0, 180, 0, 256])

cv2.normalize(h1, h1, 0, 1, cv2.NORM_MINMAX)
cv2.normalize(h2, h2, 0, 1, cv2.NORM_MINMAX)

# 0 indica coincidencia perfecta en Bhattacharyya
distancia = cv2.compareHist(h1, h2, cv2.HISTCMP_BHATTACHARYYA)
print(f"Distancia de Bhattacharyya: {distancia:.4f}")
```

3.14. Histograma en el espacio de color CIELAB (L*a*b*)

El espacio LAB es conocido por ser "perceptualmente uniforme". En este ejercicio, extraeremos el canal L (Luminosidad) y los canales a* y b* (oponencia de color). Este análisis es crucial en la industria donde se requiere una precisión de color que coincida con la visión humana.

```
#histograma_lab.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg')
lab = cv2.cvtColor(img, cv2.COLOR_BGR2Lab)
l, a, b = cv2.split(lab)

plt.figure(figsize=(10,4))
plt.subplot(131), plt.plot(cv2.calcHist([l],[0],None,[256],[0,256])),
    plt.title('L (Luminosidad)')
plt.subplot(132), plt.plot(cv2.calcHist([a],[0],None,[256],[0,256])),
    plt.title('a (Verde-Rojo)')
plt.subplot(133), plt.plot(cv2.calcHist([b],[0],None,[256],[0,256])),
    plt.title('b (Azul-Amarillo)')
plt.tight_layout()
plt.show()
```

3.15. Visualización de la línea de perfil (Profile Line) de intensidades

Aunque no es un histograma global, la línea de perfil muestra los cambios de intensidad a lo largo de un segmento lineal trazado sobre la imagen. Es una herramienta de diagnóstico útil para analizar bordes, transiciones de luz y ruido de una forma unidimensional y localizada.

```
#perfil_intensidad.py
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg', cv2.IMREAD_GRAYSCALE)
h, w = img.shape

# Definir una línea horizontal a la mitad de la imagen
fila = h // 2
perfil = img[fila, :]

plt.figure(figsize=(12, 4))
plt.subplot(121)
plt.imshow(img, cmap='gray')
plt.axhline(y=fila, color='r') # Dibujar línea de referencia
plt.title('Imagen con línea de perfil')

plt.subplot(122)
plt.plot(perfil)
plt.title('Perfil de intensidad (Fila Central)')
plt.xlabel('Columna')
plt.ylabel('Intensidad')
plt.show()
```

Tema 4

Ejercicios resueltos de Igualación de histogramas y convoluciones

En este capítulo se exploran técnicas fundamentales para la mejora y transformación de imágenes en el dominio espacial. El procesamiento se divide en dos grandes bloques: la modificación del contraste y el filtrado mediante núcleos. Iniciaremos con la **Igualación de Histogramas** [1], técnica que busca redistribuir las intensidades para maximizar el rango dinámico.

Posteriormente, estudiaremos la **Convolución** [1], una operación matemática que permite aplicar filtros lineales y no lineales. A través de estos ejercicios prácticos, aprenderemos a suavizar imágenes, eliminar diversos tipos de ruido, resaltar bordes mediante operadores de gradiente y aplicar efectos avanzados como el relieve o el desenfoque de movimiento. Estos métodos constituyen la base del preprocesamiento en sistemas modernos de visión artificial y reconocimiento de patrones.

4.1. Igualación de histograma global

La igualación global busca estirar el histograma de una imagen para que la distribución de intensidades sea uniforme. Es ideal para imágenes con poco contraste donde los píxeles están concentrados en un rango estrecho.

```
#igualacion_global.py
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg', cv2.IMREAD_GRAYSCALE)
if img is None:
    print("Error: No se pudo cargar la imagen. Revisa el nombre del
       archivo.")
else:
    # Aplicar la Ecualización de Histograma Global
    equ = cv2.equalizeHist(img)
    # Usamos np.hstack para pegar las imágenes horizontalmente
    comparativa = np.hstack((img, equ))

    plt.figure(figsize=(12, 6))
    plt.imshow(comparativa, cmap='gray')
    plt.title('Izquierda: Original | Derecha: Ecualización Global')
    plt.axis('off')

    print(f"Tamaño de la imagen: {img.shape}")
    print("Mostrando comparativa... Cierra la ventana para terminar.")
    plt.show()
```

4.2. Igualación Adaptativa de Histograma (CLAHE)

CLAHE opera sobre pequeñas regiones (bloques) de la imagen. Esto evita que áreas muy claras o oscuras dominen el proceso, logrando una mejora natural y evitando la amplificación excesiva del ruido.

```
#igualacion_clahe.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg', 0)

clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))

cl1 = clahe.apply(img)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Bajo Contraste)')
ax1.axis('off')

ax2.imshow(cl1, cmap='gray')
ax2.set_title('Resultado CLAHE (Adaptativo)')
ax2.axis('off')

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

4.3. Igualación de histogramas en imágenes a color (Espacio HSV)

Para evitar distorsiones cromáticas, convertimos la imagen al espacio HSV e igualamos únicamente el canal 'V' (Value), preservando el matiz original de la fotografía.

```
#igualacion_color_hsv.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg')

hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
h, s, v = cv2.split(hsv)
v_equ = cv2.equalizeHist(v)

final_hsv = cv2.merge((h, s, v_equ))
final_img = cv2.cvtColor(final_hsv, cv2.COLOR_HSV2BGR)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))
ax1.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax1.set_title('Original (Color)')
ax1.axis('off')
ax2.imshow(cv2.cvtColor(final_img, cv2.COLOR_BGR2RGB))
ax2.set_title('Resultado (Ecualización en V)')
ax2.axis('off')

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

4.4. Introducción a la convolución: Filtro de Identidad

Este ejercicio muestra cómo aplicar un kernel manual mediante la función `filter2D`. El kernel de identidad es la base para entender cómo los pesos de una matriz afectan al píxel central.

```
#convolucion_manual.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

img = cv2.imread('imagen.jpg')

kernel = np.array([[0, 0, 0],
                   [0, 1, 0],
                   [0, 0, 0]])

res = cv2.filter2D(img, -1, kernel)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

ax1.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax1.set_title('Imagen Original')
ax1.axis('off')
ax2.imshow(cv2.cvtColor(res, cv2.COLOR_BGR2RGB))
ax2.set_title('Resultado Convolución (Filtro Manual)')
ax2.axis('off')

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

4.5. Suavizado de imagen mediante Filtro de Media (Box Blur)

El filtrado de media reemplaza cada píxel por el promedio de sus vecinos, eliminando ruido de alta frecuencia a costa de un desenfoque uniforme.

```
#filtro_media.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg')
# El kernel (35,35) suma todos los píxeles en esa área y los divide por
   25
blur = cv2.blur(img, (35, 35))

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

ax1.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax1.set_title('Imagen Original')
ax1.axis('off')

ax2.imshow(cv2.cvtColor(blur, cv2.COLOR_BGR2RGB))
ax2.set_title('Suavizado (Filtro de Media 5x5)')
ax2.axis('off')

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

4.6. Reducción de ruido con Filtro Gaussiano

El filtro Gaussiano utiliza una campana de Gauss para asignar pesos, preservando mejor las estructuras que el filtro de media. Es el estándar para eliminar ruido antes de detectar bordes.

```
#filtro_gaussiano.py
#filtro_gaussiano.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg')

gauss = cv2.GaussianBlur(img, (5, 5), 0)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

ax1.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax1.set_title('Imagen Original (Nítida)')
ax1.axis('off')

ax2.imshow(cv2.cvtColor(gauss, cv2.COLOR_BGR2RGB))
ax2.set_title('Suavizado Gaussiano (5x5)')
ax2.axis('off')

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

4.7. Realce de detalles mediante Filtro de Afilado (Sharpening)

Aplicamos un kernel de paso alto que resalta las diferencias locales de intensidad, devolviendo nitidez a imágenes que han quedado borrosas tras un proceso de captura deficiente.

```
#filtro_afilado.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

img = cv2.imread('imagen.jpg')
kernel = np.array([[-1,-1,-1], [-1, 9,-1], [-1,-1,-1]])
sharp = cv2.filter2D(img, -1, kernel)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

ax1.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax1.set_title('Original')
ax1.axis('off')

ax2.imshow(cv2.cvtColor(sharp, cv2.COLOR_BGR2RGB))
ax2.set_title('Imagen Afilada')
ax2.axis('off')

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

4.8. Detección de bordes con el Operador Sobel

El operador Sobel calcula las derivadas de la imagen en los ejes X e Y. Es fundamental para encontrar contornos verticales y horizontales de forma independiente.

```
#filtro_sobel.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagenbordes.jpg', 0)

sobelx = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=5)
sobely = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=5)

sobel_combined = cv2.magnitude(sobelx, sobely)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')

ax2.imshow(sobel_combined, cmap='gray')
ax2.set_title('Detección de Bordes Sobel')
ax2.axis('off')

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

4.9. Filtro de Relieve (Emboss)

Este filtro resalta las transiciones de intensidad de forma direccional, creando un efecto visual de profundidad que simula el relieve en una superficie física.

```
#filtro_emboss.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

img = cv2.imread('imagen.jpg', 0)

# Definición del kernel de Emboss (Realce/Relieve)
kernel = np.array([[-2,-1,0],
                   [-1, 1,1],
                   [ 0, 1,2]])

emboss = cv2.filter2D(img, -1, kernel) + 128

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')

ax2.imshow(emboss, cmap='gray')
ax2.set_title('Efecto Emboss (Relieve)')
ax2.axis('off')

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

4.10. Filtro Bilateral: Suavizado con preservación de bordes

A diferencia de otros filtros, el bilateral considera la diferencia de intensidad cromática, permitiendo suavizar zonas planas mientras mantiene los bordes perfectamente nítidos.

```
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagenbordes.jpg', 0)

bilateral = cv2.bilateralFilter(img, 35, 75, 75)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')

ax2.imshow(bilateral, cmap='gray')
ax2.set_title('Bilateral (Bordes preservados)')
ax2.axis('off')

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

4.11. Eliminación de ruido "Sal y Pimienta"(Filtro de Mediana)

El filtro de mediana es un filtro no lineal excelente para eliminar ruido impulsivo. Selecciona el valor central de la vecindad tras ordenarlos, eliminando píxeles aberrantes sin emborronar la imagen.

```
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagenbordes.jpg',0)

median = cv2.medianBlur(img, 35)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

ax1.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax1.set_title('Original')
ax1.axis('off')

ax2.imshow(cv2.cvtColor(median, cv2.COLOR_BGR2RGB))
ax2.set_title('Filtro de Mediana (Sal y Pimienta)')
ax2.axis('off')

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

4.12. Detección de bordes mediante el Operador Laplaciano

El Laplaciano es un operador de segunda derivada que detecta regiones de cambio rápido de intensidad. A diferencia de Sobel, es isotrópico (detecta bordes en todas las direcciones simultáneamente).

```
#filtro_laplaciano.py
import cv2
import matplotlib.pyplot as plt

# Cargar en escala de grises
img = cv2.imread('imagenbordes.jpg', 0)

laplacian = cv2.Laplacian(img, cv2.CV_64F)
laplacian = cv2.convertScaleAbs(laplacian)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')

ax2.imshow(laplacian, cmap='gray')
ax2.set_title('Filtro Laplaciano (Bordes)')
ax2.axis('off')

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

4.13. Simulación de Desenfoque de Movimiento (Motion Blur)

Mediante un kernel diagonal personalizado, podemos simular el efecto de una cámara en movimiento. Este ejercicio profundiza en la creación de kernels específicos para efectos artísticos o forenses.

```
#filtro_motion_blur.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

img = cv2.imread('imagenbordes.jpg', 0)

size = 15
kernel_motion = np.zeros((size, size))
kernel_motion[int((size-1)/2), :] = np.ones(size)
kernel_motion = kernel_motion / size

res = cv2.filter2D(img, -1, kernel_motion)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')
ax2.imshow(res, cmap='gray')
ax2.set_title(f'Motion Blur Horizontal ({size}px)')
ax2.axis('off')

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

4.14. Técnica de Unsharp Masking (Máscara de desenfoque)

Esta técnica profesional consiste en restar una versión desenfocada de la imagen a la original para obtener los detalles finos y sumarlos de nuevo, logrando un realce de bordes muy superior al sharpening simple.

```
#unsharp_masking.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagenbordes.jpg', 0)

gaussian = cv2.GaussianBlur(img, (9,9), 10.0)

unsharp = cv2.addWeighted(img, 1.5, gaussian, -0.5, 0)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')

ax2.imshow(unsharp, cmap='gray')
ax2.set_title('Unsharp Masking (Nitidez)')
ax2.axis('off')

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

4.15. Operador de Prewitt para detección de gradientes

Similar al Sobel pero con pesos uniformes, el filtro de Prewitt es una alternativa clásica para el cálculo de gradientes, permitiendo comparar la sensibilidad de diferentes máscaras de derivación.

```
#filtro_prewitt.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

img = cv2.imread('imagenbordes.jpg', 0)

kernelx = np.array([[1, 1, 1], [0, 0, 0], [-1, -1, -1]])
kernely = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])

prewitt_x = cv2.filter2D(img, -1, kernelx)
prewitt_y = cv2.filter2D(img, -1, kernely)

res = cv2.addWeighted(prewitt_x, 0.5, prewitt_y, 0.5, 0)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')

ax2.imshow(res, cmap='gray')
ax2.set_title('Filtro Prewitt (Bordes)')
ax2.axis('off')

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

Tema 5

Ejercicios resueltos de Procesamiento en el dominio frecuencial

En este capítulo abandonamos el dominio espacial para analizar las imágenes desde su composición frecuencial. Mediante la Transformada Rápida de Fourier (FFT) [10] [11] [1], es posible descomponer una imagen en sus componentes de baja frecuencia (formas suaves y fondos) y alta frecuencia (bordes y detalles finos) [1].

A través de estos ejercicios, exploraremos el diseño de filtros en el plano complejo, aprendiendo a eliminar ruido periódico, suavizar texturas o resaltar contornos mediante filtros Ideales, de Butterworth y Gaussianos [1]. Este enfoque es crucial para entender procesos de compresión de imagen y restauración avanzada que serían imposibles de realizar mediante convoluciones simples.

5.1. Cálculo y visualización de la magnitud del espectro de Fourier

Para visualizar el espectro, aplicamos la FFT y desplazamos el componente de frecuencia cero al centro. Debido al alto rango dinámico, utilizamos una transformación logarítmica para poder interpretar los datos.

```
#fft_magnitud.py
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg', 0)

# Cálculo de la FFT 2D
f = np.fft.fft2(img)
fshift = np.fft.fftshift(f)
magnitude_spectrum = 20*np.log(np.abs(fshift))

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')

ax2.imshow(magnitude_spectrum, cmap='gray')
ax2.set_title('Espectro de Magnitud (Frecuencias)')
ax2.axis('off')

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

5.2. Visualización de la fase de la imagen

Aunque la magnitud contiene la distribución de frecuencias, la fase guarda la información sobre la estructura y bordes. En este ejercicio extraeremos y visualizaremos la fase del espectro.

```
#fft_fase.py
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg', 0)

f = np.fft.fft2(img)
fshift = np.fft.fftshift(f)
phase_spectrum = np.angle(fshift)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')
ax2.imshow(phase_spectrum, cmap='gray')
ax2.set_title('Espectro de Fase')
ax2.axis('off')

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

5.3. Reconstrucción mediante la Transformada Inversa (IFFT)

Comprobaremos la integridad del proceso transformando la imagen al dominio frecuencial y regresando al dominio espacial sin aplicar filtros, asegurando que no hay pérdida de información.

```
#ifft_reconstruccion.py
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg', 0)
# Calcular FFT e ir al dominio de la frecuencia
f = np.fft.fft2(img)
fshift = np.fft.fftshift(f)
# Calcular el espectro de magnitud (logarítmico para verlo bien)
magnitude_spectrum = 20 * np.log(np.abs(fshift))
# Calcular la IFFT para volver al dominio de la imagen
f_ishift = np.fft.ifftshift(fshift)
img_back = np.fft.ifft2(f_ishift)
img_back = np.abs(img_back)

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original')
ax1.axis('off')
ax2.imshow(magnitude_spectrum, cmap='gray')
ax2.set_title('Espectro Magnitud (log)')
ax2.axis('off')
ax3.imshow(img_back, cmap='gray')
ax3.set_title('Reconstrucción IFFT')
ax3.axis('off')

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

5.4. Filtro Paso Bajo Ideal (ILPF)

El filtro ideal corta todas las frecuencias superiores a una distancia D_0 del origen. Veremos cómo este corte abrupto genera el fenómeno de "ringing" o anillos en la imagen resultante.

```
#filtro_ideal_bajo.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

img = cv2.imread('imagen.jpg', 0)
rows, cols = img.shape
crow, ccol = rows // 2 , cols // 2
# Ir al dominio de la frecuencia
f = np.fft.fft2(img)
fshift = np.fft.fftshift(f)
# Crear la Máscara Ideal de Paso Bajo (Círculo blanco en fondo negro)
mask = np.zeros((rows, cols), np.uint8)
d0 = 30
cv2.circle(mask, (ccol, crow), d0, 1, -1)
# Aplicar máscara y volver al dominio del espacio
fshift_filtrado = fshift * mask
```

```
img_back = np.abs(np.fft.ifft2(np.fft.ifftshift(fshift_filtrado)))

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original')
ax1.axis('off')
ax2.imshow(mask, cmap='gray')
ax2.set_title(f'Máscara Paso Bajo (D0={d0})')
ax2.axis('off')
ax3.imshow(img_back, cmap='gray')
ax3.set_title('Imagen Filtrada (Suavizada)')
ax3.axis('off')

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

5.5. Filtro Paso Alto Ideal (IHPF)

Opuesto al anterior, este filtro atenúa las bajas frecuencias y deja pasar las altas, resultando en una imagen donde solo son visibles los bordes y las transiciones rápidas.

```
#filtro_ideal_alto.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

img = cv2.imread('imagenbordes.jpg', 0)
rows, cols = img.shape
crow, ccol = rows // 2 , cols // 2

# Ir al dominio de la frecuencia
f = np.fft.fft2(img)
fshift = np.fft.fftshift(f)
#  Crear la Máscara Ideal de Paso ALTO
mask = np.ones((rows, cols), np.uint8)
d0 = 30
# Dibujamos un círculo de CEROS en el centro (negro/bloquea el centro)
cv2.circle(mask, (ccol, crow), d0, 0, -1)
# Aplicar máscara y volver al dominio del espacio
fshift_filtrado = fshift * mask
img_back = np.abs(np.fft.ifft2(np.fft.ifftshift(fshift_filtrado)))

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original')
ax1.axis('off')
ax2.imshow(mask, cmap='gray')
ax2.set_title(f'Máscara Paso Alto (D0={d0})')
ax2.axis('off')
ax3.imshow(img_back, cmap='gray')
ax3.set_title('Imagen Filtrada (Solo Bordes)')
ax3.axis('off')

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

5.6. Filtro Paso Bajo de Butterworth (BLPF)

A diferencia del ideal, el filtro de Butterworth ofrece una transición más suave controlada por su orden n, lo que reduce significativamente el efecto de oscilación (ringing).

```
#filtro_butterworth_bajo.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

img = cv2.imread('imagen.jpg', 0)
rows, cols = img.shape

# Crear la malla de frecuencias y calcular distancias al centro
x, y = np.meshgrid(np.linspace(-cols/2, cols/2, cols),
    np.linspace(-rows/2, rows/2, rows))
d = np.sqrt(x**2 + y**2)

d0 = 30 # Radio de corte
n = 2   # Orden del filtro (a mayor n, más parecido al filtro ideal)

# Ecuación del filtro Butterworth de Paso Bajo:
# $$H(u,v) = \frac{1}{1 + [D(u,v)/D_0]^{2n}}$$
mask = 1 / (1 + (d/d0)**(2*n))

# Procesar en el dominio de la frecuencia
f = np.fft.fft2(img)
fshift = np.fft.fftshift(f)
f_filtrado = fshift * mask

img_back = np.abs(np.fft.ifft2(np.fft.ifftshift(f_filtrado)))

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original')
ax1.axis('off')

ax2.imshow(mask, cmap='gray')
ax2.set_title(f'Máscara Butterworth (n={n})')
ax2.axis('off')

ax3.imshow(img_back, cmap='gray')
ax3.set_title('Resultado (Paso Bajo)')
ax3.axis('off')

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

5.7. Filtro Paso Alto de Butterworth (BHPF)

Utilizado para el realce de bordes, este filtro permite un control preciso sobre la nitidez de la transición frecuencial, siendo muy útil en preprocesamiento de visión artificial.

```
# filtro_butterworth_alto.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

img = cv2.imread('imagen.jpg', 0)
rows, cols = img.shape

# Crear la malla de frecuencias y calcular distancias al centro
x, y = np.meshgrid(np.linspace(-cols/2, cols/2, cols),
    np.linspace(-rows/2, rows/2, rows))
d = np.sqrt(x**2 + y**2)

#  Definir parámetros del filtro
d0 = 30 # Radio de corte
n = 2   # Orden del filtro

# Ecuación del filtro Butterworth de Paso Alto:
# Se añade 1e-5 para evitar la división por cero en el centro exacto
mask = 1 / (1 + (d0 / (d + 1e-5))**(2*n))

# Procesar en el dominio de la frecuencia
f = np.fft.fft2(img)
fshift = np.fft.fftshift(f)
f_filtrado = fshift * mask

# Volver al dominio del espacio (IFFT)
img_back = np.abs(np.fft.ifft2(np.fft.ifftshift(f_filtrado)))

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original')
ax1.axis('off')

ax2.imshow(mask, cmap='gray')
ax2.set_title(f'Máscara Paso Alto (n={n})')
ax2.axis('off')

ax3.imshow(img_back, cmap='gray')
ax3.set_title('Resultado (Paso Alto)')
ax3.axis('off')

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

5.8. Filtro Paso Bajo Gaussiano (GLPF)

El filtro Gaussiano es el más suave de todos. Su función de transferencia es una campana de Gauss, lo que garantiza una eliminación total del efecto de ringing en la reconstrucción espacial.

```
#filtro_gaussiano_bajo.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

img = cv2.imread('imagen.jpg', 0)
rows, cols = img.shape

# Crear la malla de frecuencias y calcular distancias al centro
x, y = np.meshgrid(np.linspace(-cols/2, cols/2, cols),
    np.linspace(-rows/2, rows/2, rows))
distancias_cuadrado = x**2 + y**2

# Definir la Máscara Gaussiana de Paso Bajo
d0 = 30 # Radio de corte (desviación estándar)
# Fórmula: H(u,v) = exp(-D^2(u,v) / (2 * D0^2))
mask = np.exp(-distancias_cuadrado / (2 * d0**2))

# Procesar en el dominio de la frecuencia
f = np.fft.fft2(img)
fshift = np.fft.fftshift(f)
f_filtrado = fshift * mask

# Volver al dominio del espacio (IFFT)
img_back = np.abs(np.fft.ifft2(np.fft.ifftshift(f_filtrado)))

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original')
ax1.axis('off')

ax2.imshow(mask, cmap='gray')
ax2.set_title(f'Máscara Gaussiana (D0={d0})')
ax2.axis('off')

ax3.imshow(img_back, cmap='gray')
ax3.set_title('Resultado (Paso Bajo)')
ax3.axis('off')

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

5.9. Filtro Paso Alto Gaussiano (GHPF)

Este filtro se emplea para obtener una detección de bordes extremadamente limpia y suave, ideal para resaltar microestructuras en imágenes médicas o satelitales.

```
# filtro_gaussiano_alto.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

img = cv2.imread('imagen.jpg', 0)
rows, cols = img.shape

# Crear la malla de frecuencias y calcular distancias al centro
x, y = np.meshgrid(np.linspace(-cols/2, cols/2, cols),
    np.linspace(-rows/2, rows/2, rows))
distancias_cuadrado = x**2 + y**2

# Definir la Máscara Gaussiana de Paso Alto
d0 = 30 # Radio de corte
# Fórmula: H(u,v) = 1 - exp(-D^2(u,v) / (2 * D0^2))
mask = 1 - np.exp(-distancias_cuadrado / (2 * d0**2))

# Procesar en el dominio de la frecuencia
f = np.fft.fft2(img)
fshift = np.fft.fftshift(f)
f_filtrado = fshift * mask

# Volver al dominio del espacio (IFFT)
img_back = np.abs(np.fft.ifft2(np.fft.ifftshift(f_filtrado)))

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original')
ax1.axis('off')

ax2.imshow(mask, cmap='gray')
ax2.set_title(f'Máscara Paso Alto (D0={d0})')
ax2.axis('off')

ax3.imshow(img_back, cmap='gray')
ax3.set_title('Resultado (Paso Alto)')
ax3.axis('off')

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

5.10. Eliminación de ruido periódico mediante Filtro de Muesca (Notch)

El ruido periódico (como interferencias eléctricas) aparece como puntos brillantes en el espectro. Con un filtro de muesca, podemos "anular" esos puntos específicos para limpiar la imagen.

```
#filtro_notch.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

img = cv2.imread('imagen.jpg', 0)
rows, cols = img.shape

# Ir al dominio de la frecuencia
f = np.fft.fft2(img)
fshift = np.fft.fftshift(f)

# Crear el Espectro de Magnitud para visualizar los picos de ruido
magnitude_spectrum = 20 * np.log(np.abs(fshift))

# Aplicar el Filtro de Muesca (Anular frecuencias específicas)
# Aquí estamos poniendo a cero dos pequeños cuadrados que representan
   el ruido
fshift[100:105, 100:105] = 0
fshift[rows-105:rows-100, cols-105:cols-100] = 0

#  Volver al dominio del espacio (IFFT)
f_ishift = np.fft.ifftshift(fshift)
img_back = np.abs(np.fft.ifft2(f_ishift))

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original')
ax1.axis('off')

ax2.imshow(magnitude_spectrum, cmap='gray')
ax2.set_title('Espectro (Notch aplicado)')
ax2.axis('off')

ax3.imshow(img_back, cmap='gray')
ax3.set_title('Reconstrucción IFFT')
ax3.axis('off')

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

5.11. Filtro de Rechazo de Banda (Band-Reject)

Este filtro elimina un rango específico de frecuencias situadas entre un radio mínimo y máximo. Es útil cuando el ruido no está en una sola frecuencia, sino en una banda determinada.

```
#filtro_rechazo_banda.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

img = cv2.imread('imagen.jpg', 0)
rows, cols = img.shape

# Crear malla de distancias al centro
x, y = np.meshgrid(np.linspace(-cols/2, cols/2, cols),
    np.linspace(-rows/2, rows/2, rows))
d = np.sqrt(x**2 + y**2)

# Definir máscara de Rechazo de Banda
d0, w = 50, 10 # Radio central y ancho de la banda
# Deja pasar lo que está fuera del anillo (menor que d0-w/2 o mayor que
    d0+w/2)
mask = np.logical_or(d < (d0-w/2), d > (d0+w/2)).astype(float)

# Procesar en frecuencia
f = np.fft.fft2(img)
fshift = np.fft.fftshift(f)
f_filtrado = fshift * mask

#  Reconstruir (IFFT)
img_back = np.abs(np.fft.ifft2(np.fft.ifftshift(f_filtrado)))

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original')
ax1.axis('off')

ax2.imshow(mask, cmap='gray')
ax2.set_title(f'Máscara Rechazo (D0={d0}, W={w})')
ax2.axis('off')

ax3.imshow(img_back, cmap='gray')
ax3.set_title('Resultado (Filtro Rechazo)')
ax3.axis('off')

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

5.12. Filtro Paso Banda (Band-Pass)

El filtro paso banda aísla un rango específico de frecuencias, descartando tanto las muy bajas como las muy altas. Se utiliza para segmentar texturas con una periodicidad conocida.

```
# filtro_paso_banda.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

img = cv2.imread('imagen.jpg', 0)
rows, cols = img.shape

# Crear la malla de frecuencias y calcular distancias al centro
x, y = np.meshgrid(np.linspace(-cols/2, cols/2, cols),
    np.linspace(-rows/2, rows/2, rows))
d = np.sqrt(x**2 + y**2)

# Definir parámetros y Máscara de Paso de Banda
d0, w = 50, 20  # d0 es el radio central, w es el ancho de la banda
# La máscara es True solo dentro del anillo definido por w
mask = np.logical_and(d >= (d0 - w/2), d <= (d0 + w/2)).astype(float)

# Procesar en el dominio de la frecuencia
f = np.fft.fft2(img)
fshift = np.fft.fftshift(f)
f_filtrado = fshift * mask

#  Volver al dominio del espacio (IFFT)
f_ishift = np.fft.ifftshift(f_filtrado)
img_back = np.abs(np.fft.ifft2(f_ishift))

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')

ax2.imshow(mask, cmap='gray')
ax2.set_title(f'Máscara Paso Banda (D0={d0}, W={w})')
ax2.axis('off')

ax3.imshow(img_back, cmap='gray')
ax3.set_title('Resultado (Paso de Banda)')
ax3.axis('off')

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

5.13. Filtrado Homomórfico

Esta técnica avanzada se utiliza para normalizar la iluminación y mejorar el contraste simultáneamente. Opera sobre el logaritmo de la imagen para separar la iluminancia de la reflectancia.

```
# filtrado_homomorfico.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

img = cv2.imread('imagen.jpg', 0)
# Evitamos log(0) y trabajamos en float64
img_float = np.float64(img) + 1.0
img_log = np.log(img_float)

rows, cols = img.shape
x, y = np.meshgrid(np.linspace(-cols/2, cols/2, cols),
                   np.linspace(-rows/2, rows/2, rows))
d2 = x**2 + y**2

#  Filtro Homomórfico (Parámetros ajustados)
rh, rl, c, d0 = 1.5, 0.5, 1.0, 50  # d0 un poco más grande
mask = (rh - rl) * (1 - np.exp(-c * (d2 / (d0**2)))) + rl

#  FFT y Filtrado
f = np.fft.fftshift(np.fft.fft2(img_log))
res_f = f * mask

#  Inversa y Exponencial
res_ifft = np.fft.ifft2(np.fft.ifftshift(res_f))
# Usamos np.real para evitar problemas de fase
res_exp = np.exp(np.real(res_ifft)) - 1.0

#  Normalización ROBUSTA
# Eliminamos posibles valores infinitos o Nan
res_exp = np.nan_to_num(res_exp)
# Re-escalado manual para asegurar visibilidad
res_final = cv2.normalize(res_exp, None, 0, 255, cv2.NORM_MINMAX)
res_final = np.uint8(np.clip(res_final, 0, 255))

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original')
ax1.axis('off')

ax2.imshow(mask, cmap='gray')
ax2.set_title('Máscara Filtro')
ax2.axis('off')

ax3.imshow(res_final, cmap='gray')
ax3.set_title('Resultado Corregido')
ax3.axis('off')

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

5.14. Énfasis de Alta Frecuencia (High-Frequency Emphasis)

Este filtro combina un filtro paso alto con una constante de desplazamiento para preservar los tonos bajos mientras se resaltan los detalles, evitando que la imagen quede excesivamente oscura.

```
#enfasis_alta_frecuencia.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

# Cargar imagen en gris
img = cv2.imread('imagen.jpg', 0)
rows, cols = img.shape

# Crear malla de frecuencias y Filtro Paso Alto Gaussiano (HP)
x, y = np.meshgrid(np.linspace(-cols/2, cols/2, cols),
    np.linspace(-rows/2, rows/2, rows))
d0 = 30
hp = 1 - np.exp(-(x**2 + y**2) / (2 * d0**2))

#  Máscara de Énfasis: a + b * HP
# a=0.5 (mantiene parte de la base), b=1.5 (potencia los bordes)
a, b = 0.5, 1.5
mask = a + b * hp

#  Procesar en frecuencia
f = np.fft.fftshift(np.fft.fft2(img))
f_filtrado = f * mask

#  Reconstruir (IFFT)
# Usamos np.real para mayor estabilidad en la reconstrucción
res = np.abs(np.fft.ifft2(np.fft.ifftshift(f_filtrado)))

#  Normalizar correctamente para visualización
res_norm = cv2.normalize(res, None, 0, 255, cv2.NORM_MINMAX)
res_final = np.uint8(res_norm)

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original')
ax1.axis('off')

ax2.imshow(mask, cmap='gray')
ax2.set_title(f'Máscara Énfasis ({a} + {b}*HP)')
ax2.axis('off')

ax3.imshow(res_final, cmap='gray')
ax3.set_title('Resultado (Nitidez aumentada)')
ax3.axis('off')

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

5.15. Análisis de la importancia de la fase frente a la magnitud

En este ejercicio final, intercambiaremos la fase de una imagen con la magnitud de otra para demostrar cómo la fase es la que realmente contiene la geometría de los objetos.

```
#importancia_fase.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

# 1. Cargar y redimensionar imágenes en gris
# Ambas deben tener el mismo tamaño para que sus espectros coincidan
img1 = cv2.resize(cv2.imread('imagen.jpg', 0), (300, 300))
img2 = cv2.resize(cv2.imread('imagenbordes.jpg', 0), (300, 300))

# 2. Calcular la FFT de ambas
f1 = np.fft.fft2(img1)
f2 = np.fft.fft2(img2)

# 3. Combinar: Magnitud de img1 + Fase de img2
# La fase (angle) determina dónde se colocan los componentes (la
    estructura)
combined = np.multiply(np.abs(f1), np.exp(1j * np.angle(f2)))

# 4. Reconstruir (IFFT)
# Usamos np.abs para obtener la intensidad final de los números
    complejos
res = np.abs(np.fft.ifft2(combined))

# Normalizar el resultado para que se vea correctamente (0-255)
res_final = cv2.normalize(res, None, 0, 255,
    cv2.NORM_MINMAX).astype(np.uint8)

# 5. Visualización de 3 paneles
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))

ax1.imshow(img1, cmap='gray')
ax1.set_title('Imagen 1 (Aporta Magnitud)')
ax1.axis('off')

ax2.imshow(img2, cmap='gray')
ax2.set_title('Imagen 2 (Aporta Fase)')
ax2.axis('off')

ax3.imshow(res_final, cmap='gray')
ax3.set_title('Resultado: Magnitud 1 + Fase 2')
ax3.axis('off')

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

Tema 6

Ejercicios resueltos de Umbralización global y local de imágenes

En este capítulo se aborda la técnica de umbralización, un paso crítico en la segmentación de imágenes. El objetivo principal es convertir una imagen en escala de grises en una imagen binaria, facilitando la extracción de objetos de interés.

Exploraremos la **Umbralización Global** [1], donde se aplica un único valor a toda la imagen, analizando métodos manuales y automáticos como el algoritmo de Otsu y el método del Triángulo. Posteriormente, resolveremos problemas de iluminación no uniforme mediante la Umbralización Adaptativa (Local), donde el umbral se calcula dinámicamente para cada región de la imagen. Finalmente, introduciremos métodos avanzados para el procesamiento de documentos degradados como los algoritmos de Niblack y Sauvola.

6.1. Umbralización global simple (Binarización)

La forma más básica de segmentación. Si el píxel supera un valor T, se asigna el máximo (255), de lo contrario, se asigna 0. Es ideal cuando el histograma es claramente bimodal.

```
# umbral_simple.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg', 0)

#  Aplicar umbral simple (Binary Thresholding)
# Los valores menores a 127 se vuelven negros (0) y los mayores blancos
    (255)
_, res = cv2.threshold(img, 127, 255, cv2.THRESH_BINARY)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')

ax2.imshow(res, cmap='gray')
ax2.set_title('Umbral Simple (127)')
ax2.axis('off')

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

6.2. Umbralización global invertida

Similar a la simple, pero invirtiendo los roles: los píxeles que superan el umbral se vuelven negros. Útil cuando los objetos de interés son más oscuros que el fondo.

```
# umbral_invertido.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg', 0)

#  Aplicar umbral invertido
# Los valores menores a 127 se vuelven negros (0) y los mayores blancos
    (255)
_, res = cv2.threshold(img, 127, 255, cv2.THRESH_BINARY_INV)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')

ax2.imshow(res, cmap='gray')
ax2.set_title('Umbral Invertido')
ax2.axis('off')

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

6.3. Umbralización Truncada (Trunc)

En este método, los píxeles que superan el umbral se igualan al valor del umbral, mientras que el resto permanece igual. Se utiliza para normalizar picos de intensidad.

```
# umbral_truncado.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg', 0)

#  Aplicar umbral truncado
# Los valores menores a 127 se vuelven negros (0) y los mayores blancos
   (255)
_, res = cv2.threshold(img, 127, 255, cv2.THRESH_TRUNC)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')

ax2.imshow(res, cmap='gray')
ax2.set_title('Umbral Truncado')
ax2.axis('off')

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

6.4. Umbralización a Cero (Tozero)

Los píxeles que no alcanzan el umbral se eliminan (se ponen a 0), manteniendo intactos aquellos que son lo suficientemente brillantes. Es una forma de filtrado de ruido de baja intensidad.

```
# umbral_tozero.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg', 0)

#  Aplicar umbral  TOZERO
# Los valores menores a 127 se vuelven negros (0) y los mayores blancos
   (255)
_, res = cv2.threshold(img, 127, 255, cv2.THRESH_TOZERO)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')

ax2.imshow(res, cmap='gray')
ax2.set_title('Umbral Tozero')
ax2.axis('off')

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

6.5. Binarización automática mediante el Método de Otsu

Otsu calcula automáticamente el umbral óptimo minimizando la varianza dentro de cada clase. Es el estándar de oro para imágenes con histogramas bimodales bien definidos.

```
# umbral_otsu.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg', 0)

#  Aplicar umbral simple (Binary Thresholding) + Otsu
val, res = cv2.threshold(img, 0, 255, cv2.THRESH_BINARY +
    cv2.THRESH_OTSU )

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')

ax2.imshow(res, cmap='gray')
ax2.set_title('Umbral Otsu')
ax2.axis('off')

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

6.6. Umbralización Adaptativa por Media (Local)

Cuando la iluminación es desigual, un umbral global falla. Este método calcula el umbral de cada píxel como el promedio de su vecindad, compensando sombras y reflejos.

```
# umbral_adaptativo_media.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg', 0)

t_adapt_mean = cv2.adaptiveThreshold(img, 255,
    cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 11, 2)


fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')

ax2.imshow(t_adapt_mean, cmap='gray')
ax2.set_title('Umbral adaptativo media')
ax2.axis('off')

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

6.7. Umbralización Adaptativa Gaussiana (Local)

A diferencia de la media, este método utiliza una suma ponderada gaussiana de la vecindad. Proporciona resultados más limpios en los bordes y es más robusto ante el ruido fino.

```
# umbral_adaptativo_media.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg', 0)

t_adapt_gauss = cv2.adaptiveThreshold(img, 255,
    cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11, 2)


fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')

ax2.imshow(t_adapt_gauss, cmap='gray')
ax2.set_title('Umbral adaptativo GAUSSIANA')
ax2.axis('off')

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

6.8. Otsu tras suavizado Gaussiano

El ruido puede engañar al algoritmo de Otsu. En este ejercicio aprenderemos a pre-procesar la imagen con un filtro Gaussiano para obtener un umbral automático mucho más estable.

```
# umbral_otsu.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg', 0)

#  Filtro Gaussiano + Otsu
blur = cv2.GaussianBlur(img, (35,35),0)
_, res = cv2.threshold(blur, 0, 255, cv2.THRESH_BINARY +
    cv2.THRESH_OTSU )

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')

ax2.imshow(res, cmap='gray')
ax2.set_title('Umbral Otsu')
ax2.axis('off')

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

6.9. Método del Triángulo para imágenes unimodales

Cuando los objetos ocupan una parte muy pequeña de la imagen, el histograma no es bimodal. El método del triángulo busca el punto más alejado de la recta principal del histograma para segmentar.

```
# umbral_simple.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg', 0)

#  Aplicar umbral  TRIANGULO
val, res = cv2.threshold(img, 127, 255, cv2.THRESH_BINARY +
    cv2.THRESH_TRIANGLE)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')

ax2.imshow(res, cmap='gray')
ax2.set_title('Método del triángulo')
ax2.axis('off')

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

6.10. Umbralización multinivel (Segmentación de tonos)

En ocasiones necesitamos separar más de dos regiones. Aquí utilizaremos múltiples umbrales para clasificar la imagen en diferentes niveles de intensidad (ej: sombras, tonos medios y luces).

```
#umbral_multinivel.py
import cv2
import matplotlib.pyplot as plt
import numpy as np
img = cv2.imread('imagen.jpg', 0)
res = np.zeros_like(img)
res[img > 85] = 127
res[img > 170] = 255


fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')

ax2.imshow(res, cmap='gray')
ax2.set_title('Método umbralización multinivel')
ax2.axis('off')

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

6.11. Segmentación por color en espacio HSV

La umbralización no se limita al gris. En este ejercicio aplicaremos umbrales sobre los canales de Matiz (H) y Saturación (S) para aislar objetos de un color específico.

```
#umbral_color_hsv.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

img = cv2.imread('imagen.jpg')

# Convertir a espacio de color HSV
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

# Definir el rango para detectar color azul
# Los valores HSV en OpenCV van: H(0-179), S(0-255), V(0-255)
bajo = np.array([100, 50, 50])
alto = np.array([130, 255, 255])

# Crear la máscara (los píxeles en rango serán blancos, el resto negros)
mask = cv2.inRange(hsv, bajo, alto)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

ax1.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax1.set_title('Imagen Original (Color)')
ax1.axis('off')

ax2.imshow(mask, cmap='gray')
ax2.set_title('Máscara: Color Azul Detectado')
ax2.axis('off')

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

6.12. Algoritmo de Niblack para análisis de texto

Niblack es un método local clásico que utiliza la media y la desviación estándar de la vecindad. Es extremadamente eficaz para binarizar documentos antiguos o con manchas.

```
# umbral_niblack.py
import cv2
import matplotlib.pyplot as plt
import numpy as np # IMPORTANTE: faltaba esta línea
from skimage.filters import threshold_niblack

# 1. Cargamos en color para el panel 1 y en gris para procesar
img_color = cv2.imread('imagentexto.png')
img_gray = cv2.cvtColor(img_color, cv2.COLOR_BGR2GRAY)

# 2. Aplicar el umbral de Niblack
# El cálculo se basa en la media y desviación estándar local:
# T = m + k * s
thresh = threshold_niblack(img_gray, window_size=25, k=0.8)
res = (img_gray > thresh).astype(np.uint8) * 255

# 3. Visualización de 2 paneles
```

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

# Panel 1: Original (Color real)
ax1.imshow(cv2.cvtColor(img_color, cv2.COLOR_BGR2RGB))
ax1.set_title('Imagen Original (Color)')
ax1.axis('off')

# Panel 2: Resultado de Niblack
# Cambiamos 'mask' por 'res' que es tu variable correcta
ax2.imshow(res, cmap='gray')
ax2.set_title('Umbralizado Niblack (Local)')
ax2.axis('off')

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

6.13. Algoritmo de Sauvola (Mejora de Niblack)

Sauvola mejora a Niblack reduciendo el ruido en las zonas donde el contraste es bajo. Es el algoritmo estándar en la industria para el OCR (Reconocimiento Óptico de Caracteres).

```
# umbral_sauvola.py
import cv2
import matplotlib.pyplot as plt
import numpy as np
from skimage.filters import threshold_sauvola # Importamos Sauvola

img_color = cv2.imread('imagentexto.png')
if img_color is None:
    print("Error: No se pudo cargar la imagen. Revisa el nombre y la
        extensión.")
else:
    img_gray = cv2.cvtColor(img_color, cv2.COLOR_BGR2GRAY)

    #  Aplicar el umbral de Sauvola
    # window_size: tamaño del bloque (debe ser mayor que el grosor del
        texto)
    # k: sensibilidad (valor típico 0.2 para Sauvola)
    window_size = 25
    thresh = threshold_sauvola(img_gray, window_size=window_size, k=0.2)
    res = (img_gray > thresh).astype(np.uint8) * 255

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

    ax1.imshow(cv2.cvtColor(img_color, cv2.COLOR_BGR2RGB))
    ax1.set_title('Imagen Original (Color)')
    ax1.axis('off')

    ax2.imshow(res, cmap='gray')
    ax2.set_title(f'Método Sauvola (k=0.2, ws={window_size})')
    ax2.axis('off')

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

6.14. Umbralización local por bloques (Manual)

Implementaremos nuestra propia versión de umbralización local dividiendo la imagen en bloques (grids) y calculando el umbral de Otsu de forma independiente para cada cuadro.

```
# umbral_bloques_manual.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

img = cv2.imread('imagen.jpg', 0)
if img is None:
    print("Error: No se encuentra 'imagen.jpg'")
else:
    h, w = img.shape
    res = np.zeros_like(img)
    bs = 50 # Tamaño del bloque (block size)

    #  Procesamiento manual por bloques
    for i in range(0, h, bs):
        for j in range(0, w, bs):
            # Extraer la región de interés (ROI)
            roi = img[i:i+bs, j:j+bs]

            # Aplicar Otsu localmente a este bloque
            # Otsu ignora el valor '0' y busca el umbral óptimo
                automáticamente
            _, thresh = cv2.threshold(roi, 0, 255, cv2.THRESH_BINARY +
                cv2.THRESH_OTSU)

            # Insertar el bloque procesado en la imagen de resultado
            res[i:i+bs, j:j+bs] = thresh

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

    ax1.imshow(img, cmap='gray')
    ax1.set_title('Imagen Original (Gris)')
    ax1.axis('off')

    ax2.imshow(res, cmap='gray')
    ax2.set_title(f'Umbralizado por Bloques ({bs}x{bs})')
    ax2.axis('off')

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

6.15. Comparativa de robustez: Global vs Adaptativo

En este ejercicio final compararemos visualmente cómo se comportan los métodos globales frente a los locales en una imagen con un fuerte gradiente de luz artificial.

```
# comparativa_umbrales.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

img = cv2.imread('imagen.jpg', 0)

if img is None:
    print("Error: No se pudo cargar 'imagen.jpg'.")
else:
    # Umbral Global Simple (valor fijo de 127)
    _, t1 = cv2.threshold(img, 127, 255, cv2.THRESH_BINARY)

    # Umbral Adaptativo Gaussiano
    # Calcula el umbral para áreas pequeñas de 11x11 píxeles.
    t2 = cv2.adaptiveThreshold(img, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                               cv2.THRESH_BINARY, 11, 2)

    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))

    ax1.imshow(img, cmap='gray')
    ax1.set_title('Original (Gris)')
    ax1.axis('off')


    ax2.imshow(t1, cmap='gray')
    ax2.set_title('Global (127)')
    ax2.axis('off')


    ax3.imshow(t2, cmap='gray')
    ax3.set_title('Adaptativo Gaussiano')
    ax3.axis('off')

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

Tema 7

Ejercicios resueltos de Práctica 7: Detección de objetos mediante cascadas de Haar.

En este capítulo exploraremos la detección de objetos basada en clasificadores de cascada [1]. Esta técnica utiliza características de Haar, que analizan las diferencias de intensidad en regiones adyacentes de la imagen, junto con una estructura de clasificadores en cascada para descartar rápidamente las regiones de la imagen que no contienen el objeto buscado.

A través de estos ejercicios, aprenderemos a configurar los parámetros críticos del método `detectMultiScale`, como el factor de escala y el número mínimo de vecinos, para detectar rostros, ojos, cuerpos y otros objetos pre-entrenados. También abordaremos la optimización mediante regiones de interés (ROI) y el procesamiento de video en tiempo real, proporcionando las herramientas necesarias para construir sistemas de vigilancia y monitorización básica.

7.1. Detección básica de rostros frontales

Utilizaremos el clasificador estándar de OpenCV para identificar rostros frontales. En este ejercicio aprenderemos a cargar el archivo XML y a dibujar cuadros delimitadores (bounding boxes) sobre las detecciones.

```
#deteccion_rostro_basica.py
import cv2
import matplotlib.pyplot as plt
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
    'haarcascade_frontalface_default.xml')
img = cv2.imread('personas.png')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
faces = face_cascade.detectMultiScale(gray, 1.3, 5)
for (x,y,w,h) in faces:
    cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```

7.2. Ajuste del parámetro scaleFactor

El parámetro `scaleFactor` compensa que algunos rostros estén más cerca de la cámara que otros. Veremos cómo un valor demasiado alto puede hacer que ignoremos rostros pequeños.

```
#ajuste_scale_factor.py
import cv2
import matplotlib.pyplot as plt
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
    'haarcascade_frontalface_default.xml')
img = cv2.imread('personas.png')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# scaleFactor=1.05 es mas lento pero detecta rostros mas pequeños
faces = face_cascade.detectMultiScale(gray, 1.05, 5)
for (x,y,w,h) in faces: cv2.rectangle(img,(x,y),(x+w,y+h),(0,255,0),2)
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```

7.3. Ajuste de minNeighbors para reducir falsos positivos

Este parámetro define cuántos "vecinos" debe tener un rectángulo candidato para ser retenido. Aprenderemos a equilibrar la sensibilidad del detector para eliminar detecciones erróneas en el fondo.

```
#ajuste_min_neighbors.py
import cv2
import matplotlib.pyplot as plt
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
    'haarcascade_frontalface_default.xml')
img = cv2.imread('personas.png')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# minNeighbors=10 es mas estricto, evita falsos positivos
faces = face_cascade.detectMultiScale(gray, 1.3, 10)
for (x,y,w,h) in faces: cv2.rectangle(img,(x,y),(x+w,y+h),(0,0,255),2)
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```

7.4. Detección de ojos dentro de un rostro

Para mejorar la precisión y velocidad, no buscaremos ojos en toda la imagen. Primero detectaremos el rostro y definiremos una Región de Interés (ROI) donde limitaremos la búsqueda de los ojos.

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

face_cas = cv2.CascadeClassifier(cv2.data.haarcascades +
   'haarcascade_frontalface_default.xml')
eye_cas = cv2.CascadeClassifier(cv2.data.haarcascades +
   'haarcascade_eye.xml')

img = cv2.imread('personas.png')
if img is None:
    print("Error: No se pudo cargar la imagen.")
else:
    # Convertir a gris para el procesamiento
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    img_out = img.copy()
    # Detectar Rostros
    faces = face_cas.detectMultiScale(gray, scaleFactor=1.1,
       minNeighbors=5, minSize=(100, 100))
    for (x, y, w, h) in faces:
        # Dibujar rectángulo en el rostro (Azul)
        cv2.rectangle(img_out, (x, y), (x+w, y+h), (255, 0, 0), 2)
        # Definir las Regiones de Interés (ROI)
        roi_gray = gray[y:y+h, x:x+w]
        roi_color = img_out[y:y+h, x:x+w]

        # RESTRICCIÓN PARA EVITAR LA BOCA
        # Cortamos el ROI de la cara para que solo busque en el 60%
           superior
        alto_ojos = int(h * 0.6)
        zona_ojos_gray = roi_gray[0:alto_ojos, :]

        #  Detectar Ojos en la zona restringida
        # minNeighbors=10 es lo bastante estricto para evitar falsos
           positivos
        eyes = eye_cas.detectMultiScale(
            zona_ojos_gray,
            scaleFactor=1.05,
            minNeighbors=10,
            minSize=(30, 30)
        )

        for (ex, ey, ew, eh) in eyes:
              cv2.rectangle(roi_color, (ex, ey), (ex+ew, ey+eh), (0,
                 255, 0), 2)


    plt.figure(figsize=(12, 8))
    plt.imshow(cv2.cvtColor(img_out, cv2.COLOR_BGR2RGB))
    plt.title('Detección Fina: Rostros y Ojos (Sin Boca)')
    plt.axis('off')
    plt.show()
```

7.5. Detección de sonrisas

El clasificador de sonrisas es más sensible que el de rostros. En este ejercicio implementaremos un detector que identifique si la persona en la imagen está sonriendo, ajustando los parámetros para evitar falsos positivos en los pliegues de la nariz.

```
#deteccion_sonrisa.py
import cv2
import matplotlib.pyplot as plt
face_cas = cv2.CascadeClassifier(cv2.data.haarcascades +
   'haarcascade_frontalface_default.xml')
smile_cas = cv2.CascadeClassifier(cv2.data.haarcascades +
   'haarcascade_smile.xml')
img = cv2.imread('personas.png')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
faces = face_cas.detectMultiScale(gray, 1.3, 5)
for (x,y,w,h) in faces:
    roi_gray = gray[y:y+h, x:x+w]
    #Detectar sonrisas
    smiles = smile_cas.detectMultiScale(roi_gray, 1.8, 20)
    for (sx,sy,sw,sh) in smiles: cv2.rectangle(img[y:y+h,
       x:x+w],(sx,sy),(sx+sw,sy+sh),(255,255,0),2)
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```

7.6. Detección de rostros de perfil

No siempre los sujetos miran de frente. OpenCV proporciona un clasificador específico para perfiles (haarcascade_profileface). Probaremos su eficacia frente al clasificador frontal estándar.

```
# deteccion_perfil.py
import cv2
import matplotlib.pyplot as plt

profile_cas = cv2.CascadeClassifier(cv2.data.haarcascades +
   'haarcascade_profileface.xml')
img = cv2.imread('personas.png')

if img is None:
    print("Error: No se encuentra la imagen.")
else:
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    ancho_img = img.shape[1]

    # Subimos minNeighbors a 12 o 15 para ser muy estrictos
    # Definimos minSize para ignorar detalles pequeños como una oreja
       sola
    params = {
        "scaleFactor": 1.1,
        "minNeighbors": 15,
        "minSize": (100, 100)
    }

    #  Detección Normal
    profiles_normal = profile_cas.detectMultiScale(gray, **params)

    #  Detección Espejo
```

```
    gray_flipped = cv2.flip(gray, 1)
    profiles_flipped = profile_cas.detectMultiScale(gray_flipped,
       **params)

    for (x, y, w, h) in profiles_normal:
        cv2.rectangle(img, (x, y), (x+w, y+h), (255, 0, 255), 3)

    for (x, y, w, h) in profiles_flipped:
        x_real = ancho_img - x - w
        cv2.rectangle(img, (x_real, y), (x_real+w, y+h), (0, 255, 255),
           3)

    plt.figure(figsize=(10, 8))
    plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
    plt.title('Detección Estricta de Perfiles')
    plt.axis('off')
    plt.show()
```

7.7. Detección de cuerpos completos (Peatones)

Extenderemos el uso de Haar a la detección de figuras humanas completas. Es una técnica común en sistemas de seguridad para contar personas que pasan por un área determinada.

```
#deteccion_cuerpo.py
import cv2
import matplotlib.pyplot as plt
body_cas = cv2.CascadeClassifier(cv2.data.haarcascades +
   'haarcascade_fullbody.xml')
img = cv2.imread('personas.png')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
bodies = body_cas.detectMultiScale(gray, 1.1, 3)
for (x,y,w,h) in bodies:
   cv2.rectangle(img,(x,y),(x+w,y+h),(0,255,255),2)
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```

7.8. Detección de parte superior del cuerpo

En entornos de oficina o videoconferencias, a menudo solo es visible el torso. Utilizaremos el clasificador upperbody para identificar personas sentadas frente a un escritorio.

```
#deteccion_torso.py
import cv2
import matplotlib.pyplot as plt
upper_cas = cv2.CascadeClassifier(cv2.data.haarcascades +
   'haarcascade_upperbody.xml')
img = cv2.imread('personas.png')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
uppers = upper_cas.detectMultiScale(gray, 1.1, 3)
for (x,y,w,h) in uppers:
   cv2.rectangle(img,(x,y),(x+w,y+h),(100,100,255),2)
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```

7.9. Detección de matrículas (LBP vs Haar)

Aunque las matrículas suelen requerir OCR, las cascadas de Haar pueden localizarlas rápidamente en una imagen de tráfico. Veremos cómo aislar la placa del resto del vehículo.

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

plate_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
    'haarcascade_russian_plate_number.xml')
img = cv2.imread('coches.jpg')
if img is None:
    print("Error: No se encuentra la imagen.")
else:
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    # Detección inicial
    # Subimos minNeighbors para ser más exigentes con el patrón de texto
    raw_plates = plate_cascade.detectMultiScale(gray, scaleFactor=1.1,
        minNeighbors=12, minSize=(50, 20))
    img_res = img.copy()
    count = 0
    #  Filtrado por Aspect Ratio (Proporción)
    for (x, y, w, h) in raw_plates:
        aspect_ratio = w / float(h)

        # Una matrícula suele tener un ratio entre 2.5 y 5.5
        if 2.5 < aspect_ratio < 5.5:
            cv2.rectangle(img_res, (x, y), (x + w, y + h), (0, 255, 0),
                3)
            cv2.putText(img_res, f'Placa {aspect_ratio:.1f}', (x, y-10),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)
            count += 1

    plt.figure(figsize=(15, 10))
    plt.imshow(cv2.cvtColor(img_res, cv2.COLOR_BGR2RGB))
    plt.title(f'Detección Filtrada: {count} objetos encontrados')
    plt.axis('off')
    plt.show()
```

7.10. Detector de rostros de gatos

OpenCV incluye clasificadores curiosos como el de rostros felinos. Veremos cómo la tecnología Haar se adapta a diferentes estructuras biológicas manteniendo la misma lógica de entrenamiento.

```
#deteccion_gatos.py
import cv2
import matplotlib.pyplot as plt
cat_cas = cv2.CascadeClassifier(cv2.data.haarcascades +
    'haarcascade_frontalcatface.xml')
img = cv2.imread('perrogato.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
cats = cat_cas.detectMultiScale(gray, 1.1, 5)
for (x,y,w,h) in cats: cv2.rectangle(img,(x,y),(x+w,y+h),(0,128,255),2)
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```

7.11. Detección en tiempo real mediante Webcam

Llevaremos los clasificadores al flujo de video. En este ejercicio optimizaremos el código para que el procesamiento de cada fotograma permita una visualización fluida sin retardos.

```
# haar_video_realtime.py
import cv2

face_cas = cv2.CascadeClassifier(cv2.data.haarcascades +
    'haarcascade_frontalface_default.xml')

#  Iniciar la captura de video (0 suele ser la webcam integrada)
cap = cv2.VideoCapture(0)

# Comprobar si la cámara se abrió correctamente
if not cap.isOpened():
    print("Error: No se pudo acceder a la cámara.")
else:
    print("Presiona 'q' para salir.")

    while True:
        # Capturar frame a frame
        ret, frame = cap.read()
        # Si el frame no se capturó bien, saltar
        if not ret:
            break
        # Pre-procesamiento: Escala de grises
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        #  Detección de rostros
        # Ajustamos a 1.1 para que sea más sensible que 1.3
        faces = face_cas.detectMultiScale(gray, scaleFactor=1.1,
            minNeighbors=5, minSize=(30, 30))

        # Dibujar rectángulos sobre el frame original
        for (x, y, w, h) in faces:
            cv2.rectangle(frame, (x, y), (x+w, y+h), (255, 0, 0), 2)

        #  MOSTRAR VIDEO (Función nativa de OpenCV)
        cv2.imshow('Deteccion Realtime', frame)

        #  Condición de salida: presionar la tecla 'q'
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    cap.release()
    cv2.destroyAllWindows()
```

7.12. Contador automático de personas

Combinando la detección de cuerpos con una variable contadora, crearemos un script que analice una imagen o video y nos devuelva el número total de sujetos detectados en la escena.

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

face_cas = cv2.CascadeClassifier(cv2.data.haarcascades +
    'haarcascade_frontalface_default.xml')
profile_cas = cv2.CascadeClassifier(cv2.data.haarcascades +
    'haarcascade_profileface.xml')

img = cv2.imread('personas.png')
if img is None:
    print("Error: No se encuentra la imagen 'taller_alfareria.jpg'")
else:
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    img_out = img.copy()

    frontales = face_cas.detectMultiScale(gray, 1.1, 5, minSize=(40,
        40))
    perfiles_normal = profile_cas.detectMultiScale(gray, 1.1, 4,
        minSize=(40, 40))

    gray_flipped = cv2.flip(gray, 1)
    perfiles_flipped = profile_cas.detectMultiScale(gray_flipped, 1.1,
        4, minSize=(40, 40))

    for (x, y, w, h) in frontales:
        cv2.rectangle(img_out, (x, y), (x+w, y+h), (255, 0, 0), 3)

    for (x, y, w, h) in perfiles_normal:
        cv2.rectangle(img_out, (x, y), (x+w, y+h), (255, 0, 255), 3)

    ancho_img = img.shape[1]
    for (x, y, w, h) in perfiles_flipped:
        x_real = ancho_img - x - w
        cv2.rectangle(img_out, (x_real, y), (x_real+w, y+h), (0, 255,
            255), 3)

    plt.figure(figsize=(15, 10))
    plt.imshow(cv2.cvtColor(img_out, cv2.COLOR_BGR2RGB))
    plt.title('Detección Completa de Personas (Frontal + Perfiles)')
    plt.axis('off')
    plt.show()
```

7.13. Privacidad: Difuminado automático de rostros

Una aplicación ética de las cascadas de Haar es el anonimato. Detectaremos los rostros y aplicaremos un filtro Gaussiano intenso o un efecto de pixelado únicamente en el área detectada.

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

img_path = 'personas.png'
img = cv2.imread(img_path)
face_cas = cv2.CascadeClassifier(cv2.data.haarcascades +
    'haarcascade_frontalface_default.xml')
profile_cas = cv2.CascadeClassifier(cv2.data.haarcascades +
    'haarcascade_profileface.xml')

if img is None:
    print(f"Error: No se encuentra la imagen {img_path}")
else:
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    # Detección Frontal
    rostros = list(face_cas.detectMultiScale(gray, 1.1, 4, minSize=(30,
        30)))
    #  Detección Perfil (Normal)
    perfiles = list(profile_cas.detectMultiScale(gray, 1.1, 4,
        minSize=(30, 30)))
    #  Detección Perfil (Espejo)
    gray_flipped = cv2.flip(gray, 1)
    perfiles_flipped = list(profile_cas.detectMultiScale(gray_flipped,
        1.1, 4, minSize=(30, 30)))
    # Ajustar coordenadas de los perfiles invertidos y añadirlos a la
        lista
    ancho_img = img.shape[1]
    for (x, y, w, h) in perfiles_flipped:
        x_real = ancho_img - x - w
        rostros.append((x_real, y, w, h))

    # Añadir los perfiles normales a la lista principal
    for rect in perfiles:
        rostros.append(rect)
    # ANONIMIZADO (Blur)
    img_anonima = img.copy()
    for (x, y, w, h) in rostros:
        # Extraer la región de interés (ROI) -> La cara
        roi_cara = img_anonima[y:y+h, x:x+w]
        # Aplicar desenfoque Gaussiano fuerte
        # (99, 99) es el tamaño del kernel (debe ser impar)
        # 30 es la desviación estándar (cuanto más alto, más borroso)
        roi_blur = cv2.GaussianBlur(roi_cara, (99, 99), 30)
        # Pegar la cara borrosa sobre la imagen original
        img_anonima[y:y+h, x:x+w] = roi_blur

    plt.figure(figsize=(12, 8))
    plt.imshow(cv2.cvtColor(img_anonima, cv2.COLOR_BGR2RGB))
    plt.title(f'Anonimizado completado: {len(rostros)} zonas
        censuradas')
    plt.axis('off')
    plt.show()
```

7.14. Filtros de realidad aumentada (Máscaras)

Detectaremos la posición de los ojos y la nariz para superponer una imagen (como unas gafas o un sombrero) de forma dinámica, ajustando el tamaño del recurso al tamaño de la detección.

```
#filtro_mascara_ar.py
import cv2
import matplotlib.pyplot as plt
face_cas = cv2.CascadeClassifier(cv2.data.haarcascades +
    'haarcascade_frontalface_default.xml')
img = cv2.imread('imagen.jpg')
# Simulamos un rectangulo de gafas
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
faces = face_cas.detectMultiScale(gray, 1.3, 5)
for (x,y,w,h) in faces:
    cv2.circle(img, (x + w//4, y + h//3), 10, (0,0,0), -1)
    cv2.circle(img, (x + 3*w//4, y + h//3), 10, (0,0,0), -1)
    cv2.line(img, (x + w//4, y+h//3), (x + 3*w//4, y+h//3), (0,0,0), 2)
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```

7.15. Detección de movimiento automático en cámara

En el ejercicio final, crearemos un script capaz de detectar movimiento automáticamente en la web cam.

```
import cv2

cap = cv2.VideoCapture(0)

ret, frame_anterior = cap.read()
if not ret:
    print("Error al iniciar cámara")
    exit()

# Pre-procesamiento inicial
gray_anterior = cv2.cvtColor(frame_anterior, cv2.COLOR_BGR2GRAY)
gray_anterior = cv2.GaussianBlur(gray_anterior, (21, 21), 0)


cv2.namedWindow("Camara de Seguridad", cv2.WINDOW_NORMAL)
cv2.namedWindow("Diferencia (Lo que ve la IA)", cv2.WINDOW_NORMAL)

cv2.moveWindow("Camara de Seguridad", 50, 50)
cv2.moveWindow("Diferencia (Lo que ve la IA)", 700, 50)


while True:
    # Leemos frame actual
    ret, frame_actual = cap.read()
    if not ret: break

    # Pre-procesamiento
    gray_actual = cv2.cvtColor(frame_actual, cv2.COLOR_BGR2GRAY)
    gray_actual = cv2.GaussianBlur(gray_actual, (21, 21), 0)

    # Calcular diferencia (Resta de imágenes)
    resta = cv2.absdiff(gray_anterior, gray_actual)
```

```
    # Umbralizar (Limpiar sombras leves)
    _, umbral = cv2.threshold(resta, 25, 255, cv2.THRESH_BINARY)
    umbral = cv2.dilate(umbral, None, iterations=2)

    # Buscar contornos del movimiento
    contornos, _ = cv2.findContours(umbral.copy(), cv2.RETR_EXTERNAL,
        cv2.CHAIN_APPROX_SIMPLE)

    for c in contornos:
        if cv2.contourArea(c) < 5000:
            continue

        (x, y, w, h) = cv2.boundingRect(c)
        cv2.rectangle(frame_actual, (x, y), (x + w, y + h), (0, 255,
            0), 2)
        cv2.putText(frame_actual, "MOVIMIENTO DETECTADO", (10, 20),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255), 2)


    cv2.imshow("Camara de Seguridad", frame_actual)
    cv2.imshow("Diferencia (Lo que ve la IA)", umbral)

    # Actualizar referencia
    gray_anterior = gray_actual.copy()

    # Salir con 'q'
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
```

Tema 8

Ejercicios resueltos de Reconocimiento con redes neuronales

En este capítulo final, nos adentramos en el paradigma del aprendizaje profundo **Deep Learning** [1]. A diferencia de los métodos anteriores donde los descriptores se diseñaban manualmente, las Redes Neuronales Artificiales son capaces de extraer automáticamente las características más relevantes de una imagen durante el proceso de entrenamiento.

A lo largo de estos ejercicios, exploraremos la arquitectura del **Perceptrón Multicapa (MLP)** [1] para problemas numéricos y las **Redes Neuronales Convolucionales (CNN)** [1] para el reconocimiento de patrones complejos y objetos. Abordaremos conceptos críticos como las funciones de activación, el entrenamiento, la prevención del sobreajuste mediante *Dropout* y el uso de modelos pre-entrenados mediante *Transfer Learning* [1], proporcionando una visión completa de cómo se construyen los motores de inteligencia artificial actuales.

8.1. Creación de un Perceptrón simple para puertas lógicas

Iniciaremos con la unidad básica de aprendizaje: la neurona. En este ejercicio entrenaremos un perceptrón para resolver funciones lógicas, entendiendo el papel de los pesos y el sesgo (bias).

```
#perceptron_basico.py
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.optimizers import Adam
# Puerta AND
X = np.array([[0,0], [0,1], [1,0], [1,1]])
y = np.array([0, 0, 0, 1])
model = Sequential([
    Input(shape=(2,)),
    Dense(1, activation='sigmoid')
])
model.compile(loss='binary_crossentropy',
              optimizer=Adam(learning_rate=0.1),
              metrics=['accuracy'])
model.fit(X, y, epochs=500, verbose=0)
test_data = np.array([[1,1]])
prediccion = model.predict(test_data, verbose=0)
print(f"Predicción para [1,1]: {prediccion[0][0]:.4f} (¿Es True?:
   {prediccion > 0.5})")
```

8.2. Clasificación de dígitos manuscritos (MNIST) con MLP

El conjunto MNIST es el "Hello World" de las redes neuronales. Implementaremos una red densa (Multi-Layer Perceptron) para reconocer números del 0 al 9 escritos a mano.

```
#mnist_mlp.py
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'  # Limpiar logs rojos
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Input
# Carga de datos
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
#  Normalización (0-255 -> 0-1)
x_train, x_test = x_train / 255.0, x_test / 255.0
#  Modelo
model = Sequential([
    Input(shape=(28, 28)),          # Capa de entrada explícita
    Flatten(),                  # Aplana la imagen 28x28 a vector de 784
    Dense(128, activation='relu'),  # Capa oculta
    Dense(10, activation='softmax')]) # Capa de salida (10 números)
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
print("Iniciando entrenamiento...")
model.fit(x_train, y_train, epochs=5)
print("\n Evaluando en datos de test ....")
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f"\nPrecisión final en Test: {test_acc*100:.2f}%")
```

8.3. Visualización de curvas de pérdida y precisión

Un modelo no es útil si no podemos evaluar su aprendizaje. Aprenderemos a graficar la evolución del error (loss) y la precisión (accuracy) durante las épocas de entrenamiento para detectar problemas de convergencia.

```
#curvas_aprendizaje.py
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Input
import matplotlib.pyplot as plt
# Carga y preparación
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
# Modelo
model = Sequential([
    Input(shape=(28, 28)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
# Entrenamiento
print(" Entrenando...")
history = model.fit(x_train, y_train, epochs=5, validation_split=0.2,
   verbose=1)
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs_range = range(1, len(acc) + 1)

plt.figure(figsize=(12, 4))
# Gráfica 1: Precisión (Accuracy)
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Entrenamiento')
plt.plot(epochs_range, val_acc, label='Validación', linestyle='--')
plt.title('Precisión (Accuracy)')
plt.legend(loc='lower right')
plt.xlabel('Épocas')
plt.ylabel('Precisión')
# Gráfica 2: Pérdida (Loss)
plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Entrenamiento')
plt.plot(epochs_range, val_loss, label='Validación', linestyle='--')
plt.title('Pérdida (Loss)')
plt.legend(loc='upper right')
plt.xlabel('Épocas')
plt.ylabel('Error')
plt.show()
```

8.4. Introducción a las Redes Convolucionales (CNN)

Las CNN son el estándar en visión artificial. En este ejercicio diseñaremos nuestra primera arquitectura convolucional, comprendiendo las capas de Convolution, ReLU y MaxPooling.

```
#cnn_basica.py
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2' # Silenciar logs de TensorFlow

import tensorflow as tf
from tensorflow.keras import layers, models, Input
import numpy as np

# Preparar datos (MNIST)
# Las CNN necesitan 3 dimensiones: (Alto, Ancho, Canales).
# MNIST viene como (28, 28), así que añadimos el "1" del canal (escala
   de grises).
(x_train, y_train), (x_test, y_test) =
   tf.keras.datasets.mnist.load_data()
x_train = x_train.reshape((60000, 28, 28, 1)).astype('float32') / 255
x_test = x_test.reshape((10000, 28, 28, 1)).astype('float32') / 255

# Definición del Modelo
model = models.Sequential([
    # Capa de Entrada Explícita: (Alto, Ancho, Canales de color)
    Input(shape=(28, 28, 1)),
    # Bloque de Convolución
    layers.Conv2D(32, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    # Aplanado y Clasificación
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])

#Resumen y Compilación
model.summary()

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Entrenar
print("\n-Entrenando CNN....")
model.fit(x_train, y_train, epochs=3, batch_size=64,
   validation_split=0.1)
```

8.5. Clasificación de ropa (Fashion-MNIST) con CNN

Aplicaremos una CNN para clasificar imágenes de prendas de vestir. Este ejercicio demuestra cómo las redes convolucionales capturan texturas y formas de manera más eficiente que las redes densas.

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2' # Limpieza de logs

import tensorflow as tf
from tensorflow.keras import layers, models, Input
import numpy as np
import matplotlib.pyplot as plt

# Nombres de las clases (Para que sea legible)
class_names = ['Camiseta/Top', 'Pantalón', 'Jersey', 'Vestido',
    'Abrigo',
               'Sandalia', 'Camisa', 'Zapatilla', 'Bolso', 'Botín']
# Carga y Preprocesamiento
print("Cargando Fashion MNIST...-")
fashion = tf.keras.datasets.fashion_mnist
(x_train, y_train), (x_test, y_test) = fashion.load_data()
# Normalizar (0-1) y Reshape (28, 28, 1)
x_train = x_train.reshape(-1, 28, 28, 1).astype('float32') / 255.0
x_test = x_test.reshape(-1, 28, 28, 1).astype('float32') / 255.0
# Definición del Modelo (CNN de dos niveles)
model = models.Sequential([
    Input(shape=(28, 28, 1)),

    # Bloque 1: Detecta bordes y formas básicas
    layers.Conv2D(32, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    # Bloque 2: Detecta patrones más complejos (texturas, formas de
       ropa)
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    # Clasificación
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

print(" Entrenando...")
model.fit(x_train, y_train, epochs=5, batch_size=64,
    validation_split=0.1)
# Elegimos una imagen aleatoria del test
idx = np.random.randint(0, len(x_test))
imagen = x_test[idx]
etiqueta_real = y_test[idx]
#Predicción
prediccion_raw = model.predict(np.expand_dims(imagen, axis=0),
    verbose=0)
clase_predicha = np.argmax(prediccion_raw)
print(f"\n Resultado de la prueba...")
```

```
print(f"Realidad:   {class_names[etiqueta_real]}")
print(f"Predicción: {class_names[clase_predicha]}")

plt.figure(figsize=(3,3))
plt.imshow(imagen.reshape(28, 28), cmap='gray')
plt.title(f"IA dice: {class_names[clase_predicha]}")
plt.axis('off')
plt.show()
```

8.6. Aumento de datos (Data Augmentation)

Para que una red sea robusta, debe aprender de imágenes con rotaciones o desplazamientos. Utilizaremos técnicas de generación de datos sintéticos para mejorar la generalización del modelo sin necesidad de capturar nuevas imágenes.

```
#data_augmentation.py
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
import numpy as np
# Cargamos una imagen de ejemplo (Fashion MNIST)
(x_train, y_train), _ = tf.keras.datasets.fashion_mnist.load_data()
# Cogemos una imagen (por ejemplo, la número 9, que suele ser un zapato
    o bolso)
imagen_original = x_train[9]
# Le damos forma: (1 imagen, 28 alto, 28 ancho, 1 canal)
imagen_input = imagen_original.reshape((1, 28, 28,
    1)).astype('float32') / 255.0

datagen = ImageDataGenerator(
    rotation_range=20,      # Rotar hasta 20 grados
    zoom_range=0.15,        # Zoom de hasta 15%
    horizontal_flip=True,   # Voltear horizontalmente (espejo)
    fill_mode='nearest'     # Cómo rellenar los huecos al rotar
)
# Vamos a generar 5 variantes de esa misma imagen
print("Generando imágenes aumentadas...")
iterador = datagen.flow(imagen_input, batch_size=1)

plt.figure(figsize=(10, 4))
# Mostramos la original
plt.subplot(1, 6, 1)
plt.title("Original")
plt.imshow(imagen_original, cmap='gray')
plt.axis('off')
# Mostramos 5 transformaciones generadas por datagen
for i in range(5):
    plt.subplot(1, 6, i + 2)
    batch = next(iterador)
    imagen_generada = batch[0].reshape(28, 28)
    plt.title(f"Gen {i+1}")
    plt.imshow(imagen_generada, cmap='gray')
    plt.axis('off')
plt.show()
```

8.7. Prevención del sobreajuste (Dropout y Regularización)

Cuando una red "memoriza" los datos en lugar de aprender, ocurre el overfitting. Implementaremos capas de Dropout para desactivar neuronas aleatoriamente y obligar a la red a encontrar caminos redundantes de aprendizaje.

```
#prevencion_overfitting.py
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2' # Limpiar logs

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Input, Flatten

# La entrada son imágenes aplanadas de 28x28 = 784 píxeles.
# Si fueran imágenes sin aplanar, usaríamos Input(shape=(28, 28)) y
   luego Flatten().

# Modelo
model = Sequential([
    #  Capa de entrada explícita
    Input(shape=(784,)),

    #  Primera capa densa
    Dense(512, activation='relu'),

    # 3. Capa de Dropout agen)
    # Durante el entrenamiento, "apaga" aleatoriamente el 50% de las
       neuronas
    # de la capa anterior en cada paso. Esto fuerza a la red a no
       depender
    # de ninguna neurona específica, reduciendo el sobreajuste.
    Dropout(0.5),

    # Capa de salida (como en la imagen)
    Dense(10, activation='softmax')
])

print("Resumen del Modelo con Dropout")
model.summary()
```

8.8. Reconocimiento de dígitos en tiempo real por Webcam

Integraremos nuestra red entrenada con el flujo de video. El sistema capturará el área de la cámara, la pre-procesará y mostrará la predicción del número detectado en tiempo real. Previamente hemos creado y entrenado el modelo utilizando el procesamiento del conjunto de números de mnist. Una vez guardado el modelo es utilizado para nuestro ejercicio de reconocimiento en tiempo real.

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
import cv2
import numpy as np
import tensorflow as tf
import sys
print("Cargando modelo...")
try:
    model = tf.keras.models.load_model('mi_red_mnist.keras')
    print(" Modelo cargado correctamente.")
```

```
except Exception as e:
    print(f" Error cargando el modelo: {e}")
    sys.exit(1)
# Iniciar Webcam con gestión de errores
cap = cv2.VideoCapture(0)
if not cap.isOpened():
    print(" Error: No se puede acceder a la webcam.")
    sys.exit(1)
cv2.namedWindow("Webcam en tiempo real", cv2.WINDOW_NORMAL)
cv2.namedWindow("Ojo de la IA", cv2.WINDOW_NORMAL)
cv2.resizeWindow("Ojo de la IA", 200, 200)
print("Cámara iniciada. Pulsa 'q' en la ventana para salir-")
try:
    while True:
        ret, frame = cap.read()
        if not ret:
            print("Error leyendo frame (¿cámara desconectada?)")
            break
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        # Definir ROI
        height, width = gray.shape
        # Hacemos el cuadro un poco más grande
        tamano_caja = 200
        x1 = int(width/2 - tamano_caja/2)
        y1 = int(height/2 - tamano_caja/2)
        x2 = int(width/2 + tamano_caja/2)
        y2 = int(height/2 + tamano_caja/2)
        # Recorte seguro (evita salirnos de la imagen)
        roi = gray[max(0, y1):min(height, y2), max(0, x1):min(width,
            x2)]
        if roi.size > 0:
            roi_resized = cv2.resize(roi, (28, 28))
            roi_resized = cv2.bitwise_not(roi_resized)
            roi_norm = roi_resized.reshape(1, 28, 28,
                1).astype('float32') / 255.0
            # Predicción
            prediccion = model.predict(roi_norm, verbose=0)
            clase = np.argmax(prediccion)
            confianza = np.max(prediccion)
            color = (0, 255, 0) if confianza > 0.7 else (0, 0, 255)
            cv2.rectangle(frame, (x1, y1), (x2, y2), color, 2)
            texto = f"Num: {clase} ({confianza*100:.0f}%)"
            cv2.putText(frame, texto, (x1, y1 - 10),
                cv2.FONT_HERSHEY_SIMPLEX, 0.8, color, 2)
            # Mostrar lo que ve la IA en la ventana pequeña
            cv2.imshow("Ojo de la IA", roi_resized)
        # Mostrar ventana principal
        cv2.imshow("Webcam en tiempo real", frame)
        # Salida controlada
        if cv2.waitKey(1) & 0xFF == ord('q'):
            print("Salir.")
            break
except KeyboardInterrupt:
    print("Interrupción de teclado detectada.")
finally:
    print("Liberando recursos")
    cap.release()
    cv2.destroyAllWindows()
```

8.9. Transfer Learning con MobileNetV2

No siempre es necesario entrenar una red desde cero. Utilizaremos una red pre-entrenada por Google (MobileNetV2) para reconocer más de 1.000 categorías de objetos comunes con una precisión profesional.

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
import cv2
import numpy as np
import tensorflow as tf
from tensorflow.keras.applications.mobilenet_v2 import MobileNetV2,
   preprocess_input, decode_predictions
print(" Cargando MobileNetV2...")
try:
    model = MobileNetV2(weights='imagenet')
    print(" Modelo cargado.")
except Exception as e:
    print(f" Error cargando modelo")
    exit()
cap = cv2.VideoCapture(0)
if not cap.isOpened():
    print(" No se detecta la cámara.")
    exit()
print("Iniciando. Pulsa 'q' o cierra la ventana para salir")
try:
    while True:
        ret, frame = cap.read()
        if not ret:
            print(" Error de lectura de cámara.")
            break
        # MobileNet espera imágenes de 224x224
        img_resized = cv2.resize(frame, (224, 224))
        # Preprocesamiento
        img_array = np.expand_dims(img_resized, axis=0)
        img_preprocessed = preprocess_input(img_array)
        # Predicción
        preds = model.predict(img_preprocessed, verbose=0)
        resultados = decode_predictions(preds, top=3)[0]
        texto_top = f"{resultados[0][1]}: {resultados[0][2]*100:.1f}%"
        cv2.putText(frame, texto_top, (10, 30),
           cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2)
        texto_segundo = f"o {resultados[1][1]}:
           {resultados[1][2]*100:.1f}%"
        cv2.putText(frame, texto_segundo, (10, 70),
           cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 255), 2)
        cv2.imshow('Detector de Objetos', frame)
        tecla = cv2.waitKey(1) & 0xFF
        if tecla == ord('q'):
            break
        if cv2.getWindowProperty('Detector de Objetos',
           cv2.WND_PROP_VISIBLE) < 1:
            break
except KeyboardInterrupt:
    print("\n Interrumpido por el usuario (Ctrl+C).")
finally:
    cap.release()
    cv2.destroyAllWindows()
```

8.10. Ajuste fino (Fine-tuning) para clasificación personalizada

Tomaremos una red experta y la re-entrenaremos para una tarea específica (ej: distinguir entre tipos de frutas). Solo ajustaremos las últimas capas para adaptar el conocimiento previo a nuestro dataset.

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

import tensorflow as tf
from tensorflow.keras import models, layers

# Supongamos que vamos a distinguir 3 cosas: "Piedra", "Papel", "Tijera"
NUM_CLASES = 3
IMG_SHAPE = (224, 224, 3)

print(" Cargando el cerebro experto (MobileNetV2) ---")
base_model = tf.keras.applications.MobileNetV2(input_shape=IMG_SHAPE,
                                               include_top=False,
                                               weights='imagenet')

# Congelar los pesos antiguos
# Esto evita que rompamos lo que la red ya ha aprendido durante años.
base_model.trainable = False

print(" Añadiendo capa personalizada ---")
model = models.Sequential([
    # El cerebro base
    base_model,
    # Esta capa convierte el mapa de características 7x7x1280 en un
       solo vector de 1280
    layers.GlobalAveragePooling2D(),
    # Capa intermedia para aprender mejor
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.2),
    # Capa de salida: Tantas neuronas como clases tengas
    layers.Dense(NUM_CLASES, activation='softmax')
])

# Compilar modelo
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

model.summary()

print("¡Listo!")
```

8.11. Evaluación mediante Matriz de Confusión

Para entender dónde falla nuestro modelo (ej: si confunde el "3" con el "8"), generaremos una matriz de confusión que desglose los aciertos y errores por cada clase.

```
#matriz_confusion.py
import matplotlib.pyplot as plt
from sklearn import datasets, svm
from sklearn.model_selection import train_test_split
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix

# Cargar datos (Dígitos 0-9)
digits = datasets.load_digits()
X = digits.images.reshape((len(digits.images), -1))
y = digits.target

# Separar entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.5, shuffle=False
)

# Entrenar modelo (SVM)
clf = svm.SVC(gamma=0.001)
clf.fit(X_train, y_train)

# Predecir
y_pred = clf.predict(X_test)

fig, ax = plt.subplots(figsize=(10, 10))

# 'from_predictions' hace todo el trabajo duro:
# calcula la matriz y la dibuja con el mapa de color.
ConfusionMatrixDisplay.from_predictions(
    y_test,
    y_pred,
    ax=ax,
    cmap='Blues',      # Mapa de color azul profesional
    colorbar=True,     # Barra lateral de intensidad
    values_format='d'  # Mostrar números enteros
)

plt.title("Matriz de Confusión: Clasificación de Dígitos")
plt.ylabel('Etiqueta Real (Verdadera)')
plt.xlabel('Etiqueta Predicha (Modelo)')

plt.show()


cm = confusion_matrix(y_test, y_pred)

errores_3_como_8 = cm[3, 8]

print("-" * 30)
print(f"ANÁLISIS DE ERROR ESPECÍFICO:")
print(f"Veces que el modelo vio un '3' pero predijo un '8':
   {errores_3_como_8}")
print("-" * 30)
```

8.12. Clasificación Binaria vs Multi-clase

Analizaremos la diferencia entre utilizar una neurona de salida con `sigmoid` (Perro vs Gato) frente a una capa `softmax` para múltiples categorías, ajustando las funciones de pérdida correspondientes.

```
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_circles, make_blobs
from sklearn.model_selection import train_test_split
import numpy as np

# BINARIO (Perro vs Gato)
# Generamos datos dummy: 2 clases (0 y 1)
X_bin, y_bin = make_circles(n_samples=100, noise=0.2, factor=0.5,
    random_state=1)
X_train_b, X_test_b, y_train_b, y_test_b = train_test_split(X_bin,
    y_bin, stratify=y_bin)

# Definición del modelo Binario
modelo_binario = MLPClassifier(
    hidden_layer_sizes=(10,),
    activation='relu',       # Activación oculta
    solver='adam',
    max_iter=500,
    random_state=1
)

modelo_binario.fit(X_train_b, y_train_b)

print(" MODELO BINARIO")
print(f"Función de Activación de Salida (Implícita): Logistic
    (Sigmoid)")
print(f"Función de Pérdida usada: Log-Loss (Binary Crossentropy)")
print(f"Predicción (probabilidad clase 1):
    {modelo_binario.predict_proba(X_test_b[:1])[0][1]:.4f}")

#  MULTI-CLASE (Dígitos 0-9)
# Generamos datos dummy: 3 clases (0, 1, 2)
X_multi, y_multi = make_blobs(n_samples=100, centers=3, random_state=1)
X_train_m, X_test_m, y_train_m, y_test_m = train_test_split(X_multi,
    y_multi)

# Definición del modelo Multi-clase
modelo_multiclase = MLPClassifier(
    hidden_layer_sizes=(10,),
    activation='relu',
    solver='adam',
    max_iter=500,
    random_state=1
)

modelo_multiclase.fit(X_train_m, y_train_m)

print("\nMODELO MULTI-CLASE")
print(f"Función de Activación de Salida (Implícita): Softmax")
print(f"Función de Pérdida usada: Cross-Entropy Loss")
probs = modelo_multiclase.predict_proba(X_test_m[:1])[0]
print(f"Predicción (vector de probs): {probs}")
print(f"Suma de probabilidades: {np.sum(probs):.2f}")
```

8.13. Extracción de características (Embeddings)

En lugar de predecir una clase, usaremos la penúltima capa de la red para obtener un vector numérico que represente la imagen. Esto es la base de los sistemas modernos de búsqueda de imágenes por similitud.

```
import tensorflow as tf
import numpy as np
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.mobilenet_v2 import preprocess_input

# Definir tamaño de imagen estándar para MobileNetV2
IMG_SHAPE = (224, 224, 3)

print(" Cargando base MobileNetV2")

#  Cargar el modelo BASE (Sin la cabeza de clasificación)
# include_top=False: Elimina la capa final que dice "Perro" o "Gato".
# weights='imagenet': Carga lo aprendido de millones de fotos.
base_model = tf.keras.applications.MobileNetV2(input_shape=IMG_SHAPE,
                                               include_top=False,
                                               weights='imagenet')

# Congelar la base
# Esto evita que se destruya lo aprendido si fueras a entrenar algo
   encima.
base_model.trainable = False

# Crear el modelo extractor final
# Añadimos Pooling para convertir el mapa 7x7x1280 en un vector limpio
   de 1280 números
model = tf.keras.Sequential([
    base_model,
    tf.keras.layers.GlobalAveragePooling2D()
])

model.summary()

def obtener_embedding(img_path):
    # Cargar imagen y redimensionar a 224x224
    img = image.load_img(img_path, target_size=IMG_SHAPE[:2])
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0) # Crear lote de 1 imagen
    x = preprocess_input(x)       # Preprocesamiento específico de
       MobileNetV2

    # El modelo devuelve un vector de 1280 números
    feature_vector = model.predict(x)
    return feature_vector.flatten()

vector = obtener_embedding('personas.png')
print(f"Vector característico generado: {vector.shape}")
```

Tema 9

Ejercicios resueltos de Cálculo del flujo óptico.

En este último capítulo, exploramos el análisis del movimiento mediante el flujo óptico. A diferencia de los capítulos anteriores que analizaban imágenes estáticas, aquí trabajamos con la dimensión temporal. El flujo óptico es fundamental para aplicaciones de seguimiento de objetos, navegación de robots, compresión de vídeo y estabilización de imagen.

A través de estos ejercicios, implementaremos los dos enfoques principales: el Flujo Óptico Disperso (algoritmo de Lucas-Kanade), que rastrea un conjunto selecto de puntos característicos, y el Flujo Óptico Denso (algoritmo de Farneback), que calcula el vector de movimiento para cada píxel de la imagen. Aprenderemos a visualizar estas trayectorias, detectar direcciones de movimiento y procesar flujos en tiempo real.

9.1. Introducción al flujo óptico disperso (Lucas-Kanade)

El método de Lucas-Kanade asume que el flujo es constante en una vecindad local. En este ejercicio, utilizaremos `calcOpticalFlowPyrLK` para rastrear puntos clave detectados previamente entre dos fotogramas consecutivos. Usaremos la webcam del portátil para ir capturar los frames a analizar.

```
#lucas_kanade_basico.py
import cv2
import numpy as np
cap = cv2.VideoCapture(0)
ret, old_frame = cap.read()
old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)
p0 = cv2.goodFeaturesToTrack(old_gray, mask=None, maxCorners=100,
    qualityLevel=0.3, minDistance=7)
while True:
    ret, frame = cap.read()
    frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    p1, st, err = cv2.calcOpticalFlowPyrLK(old_gray, frame_gray, p0,
        None)
    for i,(new,old) in enumerate(zip(p1[st==1],p0[st==1])):
        a,b = new.ravel(); c,d = old.ravel()
        cv2.line(frame, (int(a),int(b)), (int(c),int(d)), (0,255,0), 2)
    cv2.imshow('Lucas-Kanade', frame)
    if cv2.waitKey(30) & 0xFF == ord('q'): break
    old_gray = frame_gray.copy(); p0 = p1.reshape(-1,1,2)
```

9.2. Detección y rastreo de esquinas Shi-Tomasi

Para que el flujo óptico sea fiable, necesitamos puntos con alta varianza en ambas direcciones. Implementaremos el detector de esquinas de Shi-Tomasi como paso previo al rastreo de movimiento.

```
#rastreo_shi_tomasi.py
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('img1.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
#  ALGORITMO SHI-TOMASI
# maxCorners: Queremos solo los 100 mejores puntos
# qualityLevel: Calidad mínima (0.01 es permisivo, 0.3 es estricto)
# minDistance: Distancia mínima entre esquinas (para no agruparlas
    todas)
corners = cv2.goodFeaturesToTrack(gray, maxCorners=100,
                                  qualityLevel=0.01,
                                  minDistance=10)
# Convertir a enteros para poder dibujar
corners = np.int64(corners)
# Dibujar los puntos encontrados
vis = img.copy()
for i in corners:
    x, y = i.ravel()
    # Dibuja un círculo pequeño verde en cada esquina detectada
    cv2.circle(vis, (x, y), 5, (0, 255, 0), -1)
plt.imshow(cv2.cvtColor(vis, cv2.COLOR_BGR2RGB))
plt.title('Shi-Tomasi: Detección de características rastreables')
plt.axis('off')
plt.show()
```

9.3. Visualización de trayectorias (Trails)

Rastrear puntos no es suficiente; a menudo necesitamos ver su recorrido histórico. Crearemos un efecto de "estela" o dibujo de líneas que muestre el camino recorrido por los puntos detectados a lo largo del vídeo.

```
#estelas.py
import cv2
import numpy as np
import matplotlib.pyplot as plt
import sys

# CARGA DE IMÁGENES
nombre_img1 = 'img1.jpg'
nombre_img2 = 'img2.jpg'

img1 = cv2.imread(nombre_img1)
img2 = cv2.imread(nombre_img2)

if img1 is None or img2 is None:
    print(f"Error: No se pudieron cargar {nombre_img1} o
       {nombre_img2}.")
    print("Verifica que los archivos estén en la carpeta correcta.")
    sys.exit()

if img1.shape != img2.shape:
     img2 = cv2.resize(img2, (img1.shape[1], img1.shape[0]))

gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

#  Detectar características en la primera imagen (Shi-Tomasi)
p0 = cv2.goodFeaturesToTrack(gray1, maxCorners=100, qualityLevel=0.3,
   minDistance=7)

# Calcular Flujo Óptico (Lucas-Kanade)
# Calcula la nueva posición (p1) en la segunda imagen
lk_params = dict(winSize=(15, 15),
                 maxLevel=2,
                 criteria=(cv2.TERM_CRITERIA_EPS |
                    cv2.TERM_CRITERIA_COUNT, 10, 0.03))

p1, st, err = cv2.calcOpticalFlowPyrLK(gray1, gray2, p0, None,
   **lk_params)

vis = img2.copy()

if p1 is not None:
    # Seleccionar solo los puntos encontrados con éxito (status = 1)
    good_new = p1[st == 1]
    good_old = p0[st == 1]

    for i, (new, old) in enumerate(zip(good_new, good_old)):
        a, b = new.ravel() # Posición actual (img2)
        c, d = old.ravel() # Posición anterior (img1)

        # Línea ROJA: Vector de desplazamiento (desde viejo a nuevo)
        cv2.line(vis, (int(a), int(b)), (int(c), int(d)), (0, 0, 255),
           2)
```

```
        # Punto VERDE: Ubicación actual en img2
        cv2.circle(vis, (int(a), int(b)), 5, (0, 255, 0), -1)
else:
    print("No se pudo calcular el flujo óptico para ningún punto.")

plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1) # (Filas, Columnas, Índice)
# Convertir BGR a RGB para matplotlib
plt.imshow(cv2.cvtColor(img1, cv2.COLOR_BGR2RGB))
plt.title('Frame 1 (Origen: t)')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(cv2.cvtColor(img2, cv2.COLOR_BGR2RGB))
plt.title('Frame 2 (Destino: t+1)')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(cv2.cvtColor(vis, cv2.COLOR_BGR2RGB))
plt.title('Resultado Lucas-Kanade')
plt.axis('off')

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

9.4. Flujo Óptico Denso: Método de Farneback

A diferencia del disperso, el método de Farneback estima el movimiento de todos los píxeles. Esto genera una imagen de flujo mucho más rica pero computacionalmente más costosa.

```
#farneback_denso.py
import cv2
import numpy as np
import matplotlib.pyplot as plt
import sys

img1 = cv2.imread('img1.jpg')
img2 = cv2.imread('img2.jpg')

if img1 is None or img2 is None:
    print("Error: No se encontraron las imágenes.")
    sys.exit()

# Redimensionar si es necesario (Farneback es pesado, mejor imágenes no
    gigantes)
if img1.shape != img2.shape:
    img2 = cv2.resize(img2, (img1.shape[1], img1.shape[0]))

# Convertir a escala de grises
prvs = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
next_img = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

# ALGORITMO FARNEBACK
# Calcula el flujo para TODOS los píxeles
# flow tendrá dimensiones (alto, ancho, 2). El '2' son los vectores
    (dx, dy)
```

```
flow = cv2.calcOpticalFlowFarneback(prvs, next_img, None,
                                    pyr_scale=0.5, levels=3, winsize=15,
                                    iterations=3, poly_n=5,
                                       poly_sigma=1.2, flags=0)

#  PREPARAR VISUALIZACIÓN HSV
# Creamos una imagen vacía con 3 canales (Hue, Saturation, Value)
hsv = np.zeros_like(img1)

# Ponemos la Saturación al máximo (255) para colores vivos
hsv[..., 1] = 255

# CONVERSIÓN CARTESIANA A POLAR
# Convertimos los vectores (x, y) a (Magnitud, Ángulo)
# mag: Qué tan rápido se mueve
# ang: Hacia dónde se mueve
mag, ang = cv2.cartToPolar(flow[..., 0], flow[..., 1])

#  El ÁNGULO determina el COLOR (Hue)
# Convertimos radianes a grados (OpenCV usa 0-180 para Hue)
hsv[..., 0] = ang * 180 / np.pi / 2

#  La MAGNITUD determina el BRILLO (Value)
# Normalizamos para que se vea bien (0 a 255)
hsv[..., 2] = cv2.normalize(mag, None, 0, 255, cv2.NORM_MINMAX)

# Convertir de HSV a BGR para mostrarlo
bgr_flow = cv2.cvtColor(hsv, cv2.COLOR_HSV2BGR)

plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(img1, cv2.COLOR_BGR2RGB))
plt.title("Frame 1 (Original)")
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(cv2.cvtColor(bgr_flow, cv2.COLOR_BGR2RGB))
plt.title("Flujo Denso (Farneback)\nColor=Dirección, Brillo=Velocidad")
plt.axis('off')

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

9.5. Implementación del Filtro de Kalman

El Filtro de Kalman es un estimador recursivo óptimo que minimiza el error cuadrático medio. En este ejercicio, modelamos el sistema como un objeto con velocidad constante. En concreto, vamos a utilizar el ratón para la implementación en tiempo real del movimiento en un lienzo negro del movimiento.

```
#filtro_kalman.py
import cv2
import numpy as np

# Lienzo negro
frame = np.zeros((600, 800, 3), np.uint8)

# CONFIGURACIÓN KALMAN
# 4 variables de estado (x, y, dx, dy), 2 de medición (x, y)
kalman = cv2.KalmanFilter(4, 2)
kalman.measurementMatrix = np.array([[1,0,0,0], [0,1,0,0]], np.float32)
   # Solo medimos posición
kalman.transitionMatrix = np.array([[1,0,1,0], [0,1,0,1], [0,0,1,0],
   [0,0,0,1]], np.float32) # Física básica
kalman.processNoiseCov = np.eye(4, dtype=np.float32) * 0.03

# Función para capturar mouse
last_measurement = current_measurement = np.array((2,1), np.float32)
def mousemove(event, x, y, flags, param):
    global current_measurement, last_measurement
    last_measurement = current_measurement
    current_measurement = np.array([[np.float32(x)], [np.float32(y)]])

cv2.namedWindow("Kalman Tracker")
cv2.setMouseCallback("Kalman Tracker", mousemove)

while True:
    #  PREDECIR (El cuadrado VERDE es donde la IA cree que estarás)
    prediction = kalman.predict()
    pred_x, pred_y = int(prediction[0]), int(prediction[1])

    #  CORREGIR (Usamos la posición real del mouse)
    kalman.correct(current_measurement)

    # Dibujar
    cv2.line(frame, (int(last_measurement[0]),
       int(last_measurement[1])),
            (int(current_measurement[0]),
               int(current_measurement[1])), (0,255,0), 1) # Ruta real

    cv2.rectangle(frame, (pred_x-10, pred_y-10), (pred_x+10,
       pred_y+10), (0,0,255), 2) # Predicción (Rojo)

    cv2.imshow("Kalman Tracker", frame)
    if cv2.waitKey(30) & 0xFF == 27: break # ESC para salir

cv2.destroyAllWindows()
```

9.6. Seguimiento mediante Mean Shift

El algoritmo Mean Shift es una técnica no paramétrica de escalada de gradiente hacia la moda de una distribución de probabilidad. Dada una Región de Interés (ROI) inicial, calculamos su histograma de color y realizamos una retroproyección (*Back Projection*) sobre los siguientes fotogramas. Lo vamos a utilizar para hacer seguimiento mediante una web cam.

```
#mean_shift.py
import cv2
import numpy as np

cap = cv2.VideoCapture(0) # Usa la webcam

# Tomar el primer frame
ret, frame = cap.read()
# Selecciona manualmente el objeto a rastrear (haz un recuadro con el
   mouse y pulsa ENTER)
r, h, c, w = cv2.selectROI("Selecciona Objeto", frame, False)
track_window = (c, r, w, h)

# Configurar la ROI para el rastreo
roi = frame[r:r+h, c:c+w]
hsv_roi = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)
# Creamos una máscara para ignorar colores con poca luz (ruido)
mask = cv2.inRange(hsv_roi, np.array((0., 60., 32.)), np.array((180.,
   255., 255.)))
roi_hist = cv2.calcHist([hsv_roi], [0], mask, [180], [0, 180])
cv2.normalize(roi_hist, roi_hist, 0, 255, cv2.NORM_MINMAX)

term_crit = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 1)

while(1):
    ret, frame = cap.read()
    if ret == True:
        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        # Retroproyección: Probabilidad de que cada píxel pertenezca al
           objeto
        dst = cv2.calcBackProject([hsv], [0], roi_hist, [0, 180], 1)
        # Aplicamos Mean Shift para mover la ventana hacia la máxima
           densidad
        ret, track_window = cv2.meanShift(dst, track_window, term_crit)

        # Dibujar resultado
        x,y,w,h = track_window
        img2 = cv2.rectangle(frame, (x,y), (x+w,y+h), 255,2)
        cv2.imshow('Seguimiento Mean Shift',img2)

        if cv2.waitKey(1) & 0xFF == 27: break
    else: break
cap.release()
cv2.destroyAllWindows()
```

9.7. Algoritmo CamShift

Una limitación del Mean Shift es que la ventana de búsqueda tiene un tamaño fijo. El *Continuously Adaptive Mean Shift* (CamShift) resuelve esto recalculando no solo la posición, sino también el tamaño y la orientación de la ventana en cada iteración.

```
#cam_shift.py
import cv2
import numpy as np

cap = cv2.VideoCapture(0) # Usa la webcam

# Tomar el primer frame
ret, frame = cap.read()
# Selecciona manualmente el objeto a rastrear (haz un recuadro con el
   mouse y pulsa ENTER)
r, h, c, w = cv2.selectROI("Selecciona Objeto", frame, False)
track_window = (c, r, w, h)

# Configurar la ROI para el rastreo
roi = frame[r:r+h, c:c+w]
hsv_roi = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)
# Creamos una máscara para ignorar colores con poca luz (ruido)
mask = cv2.inRange(hsv_roi, np.array((0., 60., 32.)), np.array((180.,
   255., 255.)))
roi_hist = cv2.calcHist([hsv_roi], [0], mask, [180], [0, 180])
cv2.normalize(roi_hist, roi_hist, 0, 255, cv2.NORM_MINMAX)

term_crit = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 1)

while(1):
    ret, frame = cap.read()
    if ret == True:
        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        dst = cv2.calcBackProject([hsv],[0],roi_hist,[0,180],1)

        # Usamos CamShift
        ret, track_window = cv2.CamShift(dst, track_window, term_crit)

        # Dibujar una caja rotada que se adapta al tamaño
        pts = cv2.boxPoints(ret)
        pts = np.int64(pts)
        img2 = cv2.polylines(frame, [pts], True, (0, 255, 255), 2)

        cv2.imshow('Seguimiento CamShift', img2)
        if cv2.waitKey(1) & 0xFF == 27: break
    else: break

cap.release()
cv2.destroyAllWindows()
```

9.8. Filtrado de Partículas (Algoritmo CONDENSATION)

El Filtro de Partículas permite estimar estados en sistemas no lineales y no gaussianos. En este ejercicio lo aplicamos a un punto que se va moviendo en un cuadrado negro.

```
#condensation.py
import numpy as np
import cv2

# Configuración
n_particles = 200
width, height = 800, 600
particles = np.random.rand(n_particles, 2) * [width, height] #
   Partículas aleatorias
# Mouse actúa como el objeto real a rastrear
target = np.array([width/2, height/2])
def mouse(event, x, y, flags, param):
    global target
    target = np.array([x, y])

cv2.namedWindow('Particle Filter')
cv2.setMouseCallback('Particle Filter', mouse)
while True:
    frame = np.zeros((height, width, 3), dtype=np.uint8)
    #  PREDICCIÓN (Mover partículas + Ruido)
    noise = np.random.randn(n_particles, 2) * 15
    particles += noise
    # EVALUACIÓN (Calcular pesos)
    # Las partículas cerca del mouse (target) pesan más
    dist = np.linalg.norm(particles - target, axis=1)
    # Función Gaussiana de similitud: más cerca = peso más alto (hasta
       1.0)
    weights = np.exp(-dist**2 / (2 * 50**2))
    weights /= np.sum(weights) # Normalizar para que sumen 1

    #  REMUESTREO (Supervivencia del más apto)
    # Seleccionamos índices basados en su peso
    indices = np.random.choice(np.arange(n_particles),
       size=n_particles, p=weights)
    particles = particles[indices]
    # Visualización
    for p in particles:
        cv2.circle(frame, (int(p[0]), int(p[1])), 2, (255, 0, 0), -1) #
           Partículas azules

    cv2.circle(frame, (int(target[0]), int(target[1])), 10, (0, 0,
       255), -1) # Objetivo Rojo
    # Centro de masa estimado (la "respuesta" del filtro)
    estimated = np.mean(particles, axis=0)
    cv2.circle(frame, (int(estimated[0]), int(estimated[1])), 5, (0,
       255, 255), 2) # Estimación Amarilla

    cv2.imshow('Particle Filter', frame)
    if cv2.waitKey(20) & 0xFF == 27: break

cv2.destroyAllWindows()
```

9.9. Seguimiento de Puntos mediante Lucas-Kanade

Aunque Lucas-Kanade es un método de flujo óptico diferencial, puede emplearse para seguimiento a largo plazo si se actualizan iterativamente las características. Utilizamos la función `calcOpticalFlowPyrLK` para propagar la ubicación de los puntos p_t a p_{t+1}. Para evitar la deriva (*drift*), es necesario regenerar periódicamente los puntos a rastrear utilizando el detector de Shi-Tomasi cuando la calidad del seguimiento disminuye.

```
#lucas_kanade_2.py
import numpy as np
import cv2

cap = cv2.VideoCapture(0)
# Parámetros ShiTomasi y LucasKanade
feature_params = dict(maxCorners=100, qualityLevel=0.3, minDistance=7,
    blockSize=7)
lk_params = dict(winSize=(15, 15), maxLevel=2,
    criteria=(cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 0.03))
# Color aleatorio para cada punto
color = np.random.randint(0, 255, (100, 3))
# Tomar primer frame y detectar esquinas
ret, old_frame = cap.read()
old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)
p0 = cv2.goodFeaturesToTrack(old_gray, mask=None, **feature_params)
# Crear máscara para dibujar rastros
mask = np.zeros_like(old_frame)

while(1):
    ret, frame = cap.read()
    if not ret: break
    frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    # CALCULAR FLUJO OPTICO
    p1, st, err = cv2.calcOpticalFlowPyrLK(old_gray, frame_gray, p0,
        None, **lk_params)
    # Seleccionar puntos buenos
    if p1 is not None:
        good_new = p1[st==1]
        good_old = p0[st==1]
    # Dibujar rastros
    for i, (new, old) in enumerate(zip(good_new, good_old)):
        a, b = new.ravel()
        c, d = old.ravel()
        mask = cv2.line(mask, (int(a), int(b)), (int(c), int(d)),
            color[i].tolist(), 2)
        frame = cv2.circle(frame, (int(a), int(b)), 5,
            color[i].tolist(), -1)

    img = cv2.add(frame, mask)
    cv2.imshow('Lucas-Kanade Object Tracker', img)

    # Actualizar frame anterior y puntos
    old_gray = frame_gray.copy()
    p0 = good_new.reshape(-1, 1, 2)

    if cv2.waitKey(1) & 0xFF == 27: break

cv2.destroyAllWindows()
```

Bibliografía

[1] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed. Pearson, 2018.

[2] G. Van Rossum, "Foreword for programming python," https://www.python.org/doc/essays/foreword/, 1996.

[3] V. R. Guido, "The making of python. a conversation with guido van rossum," *Artima Developer*, 2003.

[4] Python Software Foundation, "Python version 1.0 release," https://www.python.org/download/releases/1.0/, January 1994.

[5] P. S. Foundation, *Python 3.14.0 Documentation*, https://docs.python.org/3.14/, Python Software Foundation, 2025.

[6] G. Bradski, "The opencv library," *Dr. Dobb's Journal of Software Tools*, vol. 25, no. 11, pp. 120–125, 2000.

[7] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library*. .ºReilly Media, Inc.", 2008.

[8] C. R. Harris, K. J. Millman, S. J. van der Walt *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[9] S. Jiménez Zafra and A. Montejo Ráez, *Curso de Programación Python*, 2nd ed. Anaya, 2021.

[10] J. B. J. Fourier, *Théorie analytique de la chaleur*. Paris: Firmin Didot, 1822, reimpreso por Cambridge University Press en 2009.

[11] J. Fourier, *The analytical theory of heat*. London: Cambridge University Press, 1878.