

Práctica 2: Trabajo de investigación

Procesamiento de Imagen, Vídeo y Audio

- **Nombre:** Andrés Lires Saborido
- **Correo electrónico:** andres.lires@udc.es
- **Grupo:** 1.1
- **Curso:** 2024/2025

Este *notebook* contiene la resolución de ambos ejercicios de la práctica. Se combinan celdas de código ejecutable y celdas de texto para comentar metodologías y resultados. Estos bloques *markdown* son complementarios al documento PDF de memoria incluido en la entrega.

Importamos las librerías de procesamiento Numpy, SciPy, Scikit-image y Scikit-learn.

```
In [1]: import numpy as np
import scipy as sp
import skimage
import sklearn as sk

print(f"NumPy: {np.__version__}")
print(f"SciPy: {sp.__version__}")
print(f"scikit-image: {skimage.__version__}")
print(f"scikit-learn: {sk.__version__}")

NumPy: 2.1.3
SciPy: 1.15.2
scikit-image: 0.25.2
scikit-learn: 1.6.1
```

Ejercicio 1: Segmentación de carreteras en imagen aérea de alta resolución

Desarrollar un método computacional que permita:

- Segmentar las carreteras existentes en una imagen de entrada. Es decir, dada una imagen aérea, proporcionar una máscara binaria con los píxeles de carretera a 1 y el resto a 01.
- Para un conjunto de imágenes de prueba, proporcionar un conjunto de métricas de evaluación cuantitativa de la segmentación apropiadas para la aplicación objetivo, usando el ground truth correspondiente.

Esquema del trabajo

El objetivo es construir un modelo de aprendizaje automático capaz de segmentar carreteras, identificando pixel a pixel su pertenencia a esta clase a partir de diversas características extraídas de las imágenes satelitales

1. Cargar las imágenes de entrada y cada *ground truth*.
2. Experimentación de posibles **operaciones sobre las imágenes originales** que nos permitan obtener características relevantes de los puntos.
 - Se probarán diferentes filtros y técnicas de procesamiento de imagen para encontrar aquellas que mejor se adapte a la segmentación de carreteras.
3. Creación de un **vector de características** por cada pixel de la imagen aérea.
 - Los valores obtenidos con las operaciones del paso anterior caracterizaran a cada pixel de la imagen.
4. División en el conjunto de entrenamiento y de test.
 - Se divide el 80% de las imágenes para el entrenamiento y el 20% restante para la evaluación del modelo.
 - Se utilizarán las imágenes de *ground truth* para obtener las etiquetas de cada pixel.
5. Creación y entrenamiento de un **clasificador** con los vectores de características y etiquetas obtenidas.
 - Se define la arquitectura del clasificador, se prueban diferentes modelos.
 - El vector de características funcionará como entrada y la etiqueta del *ground truth* como salida a predecir.

6. Evaluación del clasificador con el conjunto de imágenes de prueba y se obtiene la segmentación de carreteras.

- Se obtienen las métricas de evaluación del clasificador con el conjunto de test.
- Se predicen las etiquetas de cada píxel de la imagen de entrada y se obtiene la segmentación de carreteras.
- Se visualiza la segmentación obtenida y se compara con el *ground truth* correspondiente.

7. Las mejores predicciones son sometidas a un post-procesado

- **Experimentación extra:** ¿Como de importante es el procesamiento de imagen?

1.1. Cargar la imagen y el ground truth

Importamos librerías necesarias para este ejercicio.

```
In [ ]: import matplotlib.pyplot as plt
import scipy.ndimage as ndi
from skimage import exposure
import tensorflow as tf
from tensorflow.keras import layers, models, optimizers, losses, metrics
from skimage.morphology import closing, footprint_rectangle, disk
from tensorflow.keras.models import load_model
from skimage.measure import label, regionprops

## Version de TensorFlow
print(f"TensorFlow: {tf.__version__}")

TensorFlow: 2.19.0
```

Cargamos las imágenes de las carreteras, tanto el conjunto de imágenes de satélite como el *ground truth* y las convertimos a tipo de dato float. Visualizamos dos pares de imágenes satélite y sus *ground truth*.

```
In [3]: ## Cargamos imágenes de satélite
sat = skimage.io.imread_collection("materiales/roads/sat/*.tiff")

## Cargamos imágenes máscaras
gt = skimage.io.imread_collection("materiales/roads/gt/*.tif")

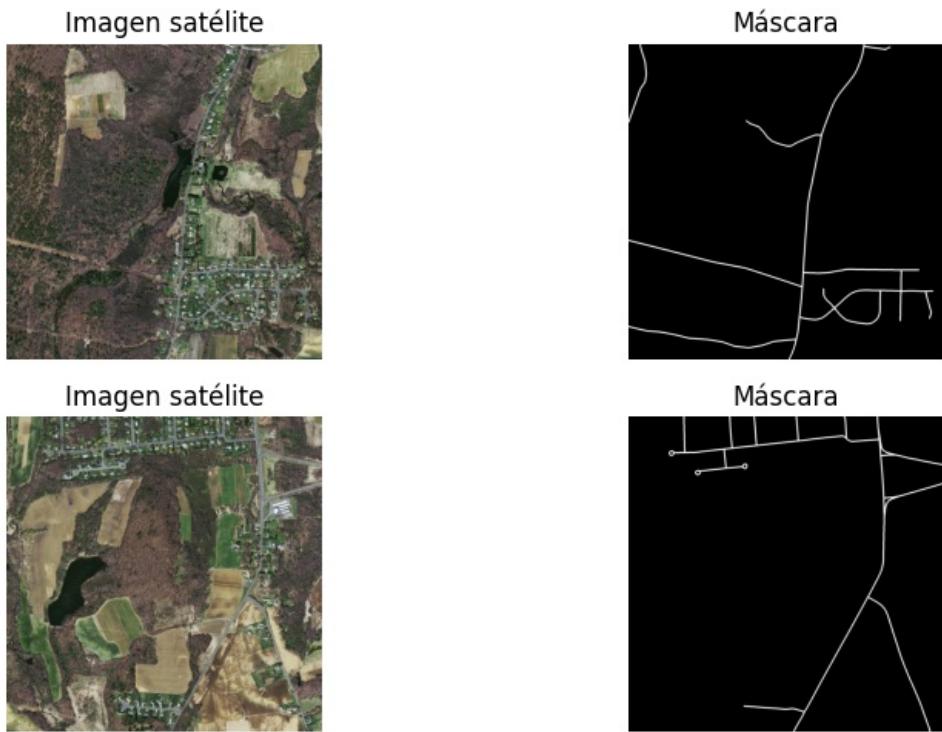
## Comprobamos que se cargó correctamente
print(f"Imágenes satélite: {len(sat)}")
print(f"Imágenes máscara: {len(gt)}")
print(f"Dimensiones imagen satélite: {sat[0].shape}")
print(f"Dimensiones imagen máscara: {gt[0].shape}")

## Imprimimos el tipo de dato de las imágenes
print(f"Tipo de dato imagen satélite antes de convertir: {sat[0].dtype} con valores entre {sat[0].min()} y {sat[0].max()}")
print(f"Tipo de dato imagen máscara antes de convertir: {gt[0].dtype} con valores entre {gt[0].min()} y {gt[0].max()}

## Convertimos cada imagen a float
sat = [skimage.img_as_float(image) for image in sat]
gt = [skimage.img_as_float(image) for image in gt]

## Visualizamos dos imágenes de satélite y su máscara en un subplot
fig, ax = plt.subplots(2, 2, figsize=(10, 5))
ax[0, 0].imshow(sat[0])
ax[0, 0].set_title("Imagen satélite")
ax[0, 0].axis("off")
ax[0, 1].imshow(gt[0], cmap="gray")
ax[0, 1].set_title("Máscara")
ax[0, 1].axis("off")
ax[1, 0].imshow(sat[1])
ax[1, 0].set_title("Imagen satélite")
ax[1, 0].axis("off")
ax[1, 1].imshow(gt[1], cmap="gray")
ax[1, 1].set_title("Máscara")
ax[1, 1].axis("off")
plt.tight_layout()
plt.show()

Imágenes satélite: 20
Imágenes máscara: 20
Dimensiones imagen satélite: (1500, 1500, 3)
Dimensiones imagen máscara: (1500, 1500)
Tipo de dato imagen satélite antes de convertir: uint8 con valores entre 0 y 255
Tipo de dato imagen máscara antes de convertir: uint8 con valores entre 0 y 255
```



- **Conclusión:**

Comprobamos que las imágenes de satélite y de *groud truth* están alineadas, es decir la primera imagen de satélite corresponde a la primera imagen de *ground truth* y así sucesivamente.

Trabajaremos con imágenes de 1500x1500 píxeles, la imagen de satélite tiene 3 canales (RGB) y el ground truth tiene un único canal (0 para el fondo y 1 para la carretera). Ambas imágenes fueron convertidas al tipo *float*.

- **Nota**

Inicialmente se había planteado la opción de **reducir la resolución de las imágenes**, pero al final, visto que el entrenamiento ofrecía un buen rendimiento en un número bajo de épocas, se decidió mantener la resolución original.

1.2. Experimentación para la extracción de características

Una vez cargadas las imágenes, el siguiente paso es el de extracción de características de la imagen satélite.

En los siguientes bloques de código se **experimentará con diferentes filtros y técnicas de procesamiento de imágenes** para obtener características relevantes que nos ayuden a segmentar las carreteras.

Trabajaremos generalmente sobre la primera imagen, pero a la hora de la creación del vector de características se aplicarán las mismas operaciones a todas las imágenes del conjunto de entrenamiento y test.

```
In [21]: img_sat = sat[0]
img_gt = gt[0]
```

Convertimos la imagen a **diferentes espacios de color** y visualizamos los diferentes canales (en escala de grises).

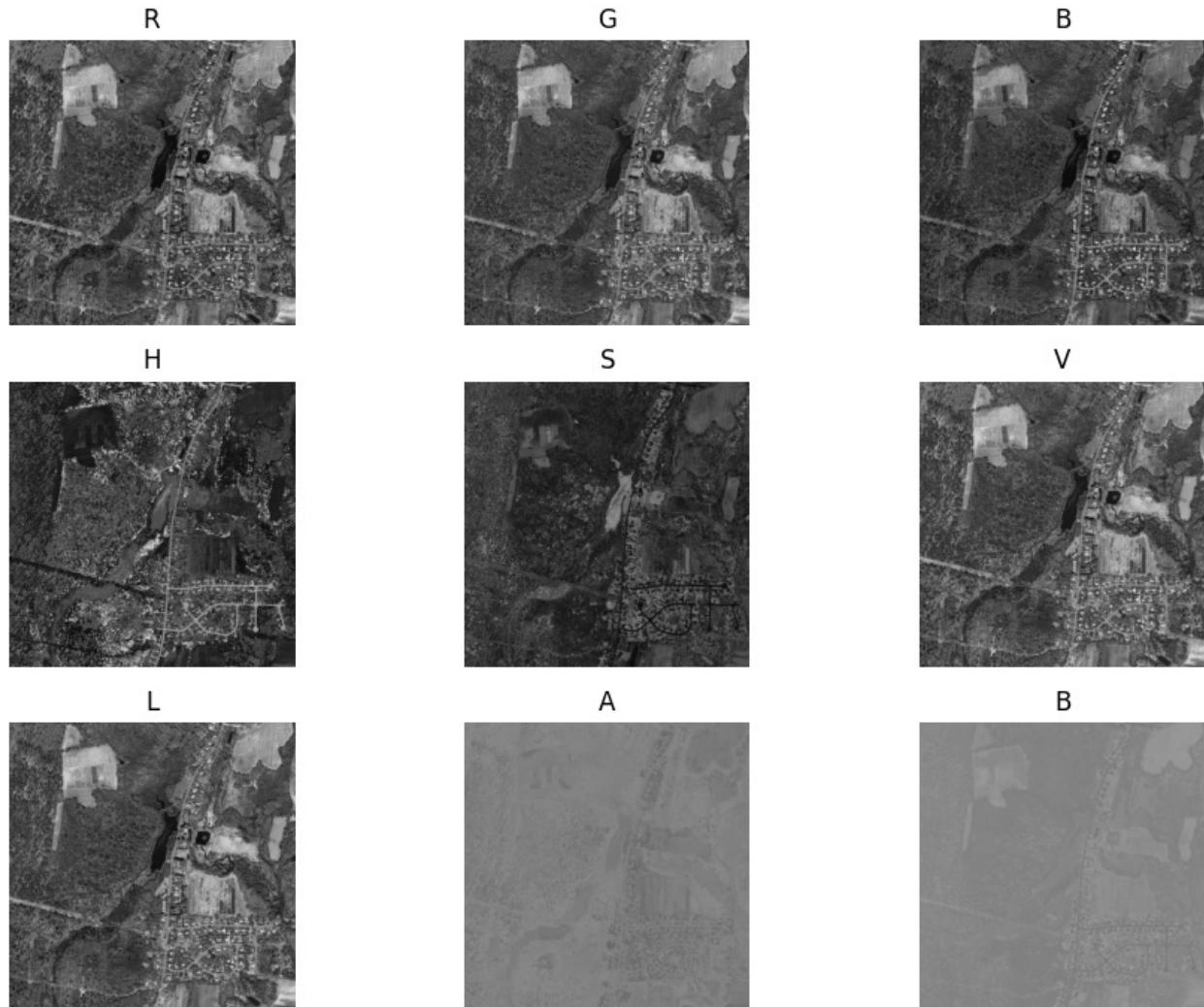
```
In [ ]: ## Convertir a HSV, LAB
img_hsv = skimage.color.rgb2hsv(img_sat)
img_lab = skimage.color.rgb2lab(img_sat)

## Visualizamos todos los canales de la imagen en RGB, HSV y LAB
fig, ax = plt.subplots(3, 3, figsize=(10, 7))
ax[0, 0].imshow(img_sat[:, :, 0], cmap="gray", vmin=0, vmax=1)
ax[0, 0].set_title("R")
ax[0, 0].axis("off")
ax[0, 1].imshow(img_sat[:, :, 1], cmap="gray", vmin=0, vmax=1)
ax[0, 1].set_title("G")
ax[0, 1].axis("off")
ax[0, 2].imshow(img_sat[:, :, 2], cmap="gray", vmin=0, vmax=1)
ax[0, 2].set_title("B")
```

```

ax[0, 2].axis("off")
ax[1, 0].imshow(img_hsv[:, :, 0], cmap="gray", vmin=0, vmax=1)
ax[1, 0].set_title("H")
ax[1, 0].axis("off")
ax[1, 1].imshow(img_hsv[:, :, 1], cmap="gray", vmin=0, vmax=1)
ax[1, 1].set_title("S")
ax[1, 1].axis("off")
ax[1, 2].imshow(img_hsv[:, :, 2], cmap="gray", vmin=0, vmax=1)
ax[1, 2].set_title("V")
ax[1, 2].axis("off")
ax[2, 0].imshow(img_lab[:, :, 0], cmap="gray", vmin=0, vmax=100) # El rango de L es [0, 100]
ax[2, 0].set_title("L")
ax[2, 0].axis("off")
ax[2, 1].imshow(img_lab[:, :, 1], cmap="gray", vmin=-128, vmax=127) # El rango de a es [-128, 127]
ax[2, 1].set_title("A")
ax[2, 1].axis("off")
ax[2, 2].imshow(img_lab[:, :, 2], cmap="gray", vmin=-128, vmax=127) # El rango de b es [-128, 127]
ax[2, 2].set_title("B")
ax[2, 2].axis("off")
plt.tight_layout()
plt.show()

```



- **Conclusión:**

Aunque los canales R, G y B de la imagen original no parecen ofrecer una buena diferencia entre lo que es carretera y lo que no, debemos tener en cuenta que la combinación de estos 3 es la que nos ofrece la imagen original en la cual si que se aprecian ciertas diferencias de color entre lo que queremos segmentar y el fondo.

Los canales H y S de la imagen en espacio HSV parecen ofrecer una buena diferencia entre lo que es carretera y lo que no.

- En el canal H la carretera es más clara que la gran mayoría de píxeles del fondo, mientras que en el canal S la carretera es más oscura que el fondo.

En el canal b de Lab*, la imagen tiene una escala de gris más uniforme, pero las carreteras aparecen significativamente más oscuras.

- Posteriormente podremos aplicar una expansión del histograma para que la diferencia sea más notable.

Estas diferencias de color podrán ser explotadas mediante una umbralización.

Mejoramos el contraste del canal b para que la diferencia entre carretera y fondo sea más notable.

1. Calculamos el histograma.
2. Expandimos el histograma.
3. Obtenemos la imagen resultante.

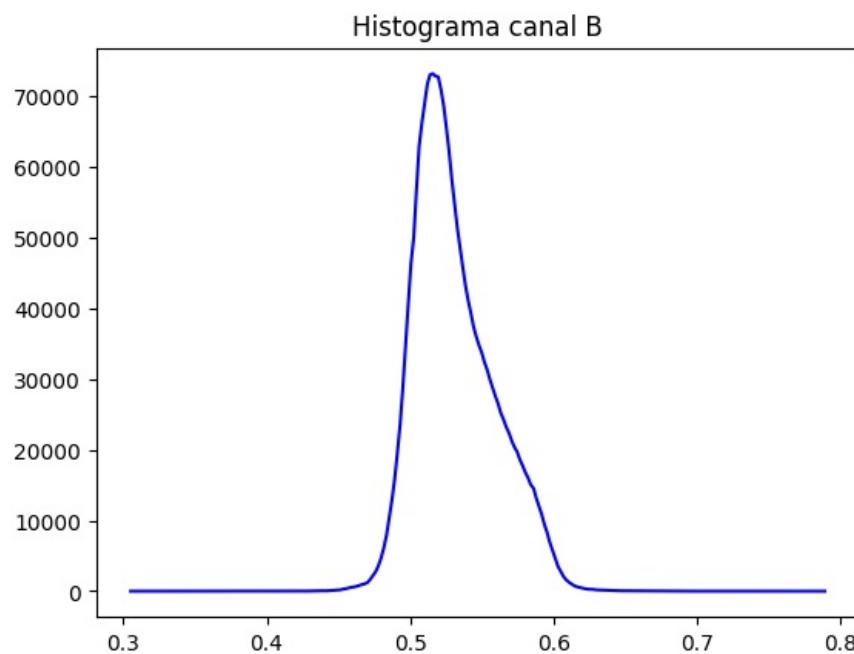
```
In [ ]: ## Seleccionamos el canal b de la imagen en LAB
img_b = img_lab[:, :, 2]

## Normalizamos la imagen al rango [0, 1]
img_b = (img_b + 128) / (127 + 128)
## Imprimimos el tipo de dato de la imagen
print(f"Tipo de dato imagen satélite después de convertir: {img_b.dtype} con valores entre {img_b.min()} y {img_b.max()}

## Calculamos el histograma de la imagen
hist_b, bins_b = exposure.histogram(img_b)

## Mostramos el histograma
plt.plot(bins_b, hist_b, color="blue")
plt.title("Histograma canal B")
plt.show()
```

Tipo de dato imagen satélite después de convertir: float64 con valores entre 0.30411355896544423 y 0.79048822166 08349



- **Nota:**

El canal b tiene un rango de valores de -128 a 127 y se convierte a un rango 0-1.

```
In [24]: ## Expansión del histograma
img_b_exp = exposure.rescale_intensity(img_b, in_range='image', out_range=(0, 1))

## Visualizamos la imagen original y la imagen expandida
fig, ax = plt.subplots(1, 2, figsize=(10, 5))
ax[0].imshow(img_b, cmap="gray", vmin=0, vmax=1)
ax[0].set_title("Imagen canal b")
ax[0].axis("off")
ax[1].imshow(img_b_exp, cmap="gray", vmin=0, vmax=1)
ax[1].set_title("Imagen canal b expandida")
ax[1].axis("off")
plt.tight_layout()
plt.show()
```

Imagen canal b



Imagen canal b expandida



- **Conclusión:**

La expansión del histograma mejora el contraste de la imagen, las carreteras son más oscuras que el fondo y la diferencia entre ambos es más notable.

A parte del propio valor del píxel en los diferentes canales puede ser interesante ya trabajar con una "**especie de segmentación obtenida por cada uno de estos canales**" considerado. Para esto, **aplicamos umbralización a la imagen expandida de b y a H y S**, poniendo a 1 aquellos píxeles que cumplan con cierta restricción de nivel de gris.

Probamos diferentes técnicas de umbralización sobre cada canal:

```
In [25]: ## Umbralización sobre el canal b expandido

## Otsu
thresh = skimage.filters.threshold_otsu(img_b_exp)
binary_otsu = img_b_exp < thresh
## Triángulo
thresh = skimage.filters.threshold_triangle(img_b_exp)
binary_triangle = img_b_exp < thresh
## Isodata
thresh = skimage.filters.threshold_isodata(img_b_exp)
binary_isodata = img_b_exp < thresh
## Manual
thresh = 0.4
binary_manual = img_b_exp < thresh

## Visualizamos los resultados
fig, ax = plt.subplots(2, 2, figsize=(10, 5))
for i, (binary, title) in enumerate(zip([binary_otsu, binary_triangle, binary_isodata, binary_manual],
                                         ["Otsu", "Triángulo", "Isodata", "Manual"])):
    ax[i // 2, i % 2].imshow(binary, cmap="gray")
    ax[i // 2, i % 2].set_title(title)
    ax[i // 2, i % 2].axis("off")
plt.tight_layout()
```

Otsu



Triángulo



Isodata



Manual



- **Aclaración:**

Utilizamos el operador `<` porque en el canal b los píxeles de carretera son más oscuros que el fondo. De esta forma convertiremos estos píxeles oscuros a color blanco en la imagen binaria (siguiendo el mismo formato que en los *ground truth*).

```
In [26]: ## Umbralización sobre H
img_h = img_hsv[:, :, 0]

## Otsu
thresh = skimage.filters.threshold_otsu(img_h)
binary_otsu = img_h > thresh
## Triángulo
thresh = skimage.filters.threshold_triangle(img_h)
binary_triangle = img_h > thresh
## Isodata
thresh = skimage.filters.threshold_isodata(img_h)
binary_isodata = img_h > thresh
## Manual
thresh = 0.3
binary_manual = img_h > thresh

## Visualizamos los resultados
fig, ax = plt.subplots(2, 2, figsize=(10, 5))
for i, (binary, title) in enumerate(zip([binary_otsu, binary_triangle, binary_isodata, binary_manual],
                                         ["Otsu", "Triángulo", "Isodata", "Manual"])):
    ax[i // 2, i % 2].imshow(binary, cmap="gray", vmin=0, vmax=1)
    ax[i // 2, i % 2].set_title(title)
    ax[i // 2, i % 2].axis("off")
plt.tight_layout()
plt.show()
```

Otsu



Triángulo



Isodata



Manual



- **Aclaración:**

En el canal H los píxeles de carretera son más claros que el fondo, por lo que utilizamos el operador `>` para convertir estos píxeles a blanco en la imagen binaria.

```
In [27]: ## Umbralización sobre S
img_s = img_hsv[:, :, 1]

## Otsu
thresh = skimage.filters.threshold_otsu(img_s)
## Triángulo
thresh = skimage.filters.threshold_triangle(img_s)
binary_triangle = img_s < thresh
## Isodata
thresh = skimage.filters.threshold_isodata(img_s)
binary_isodata = img_s < thresh
## Manual
thresh = 0.2
binary_manual = img_s < thresh

## Visualizamos los resultados
fig, ax = plt.subplots(2, 2, figsize=(10, 5))
for i, (binary, title) in enumerate(zip([binary_otsu, binary_triangle, binary_isodata, binary_manual],
                                         ["Otsu", "Triángulo", "Isodata", "Manual"])):
    ax[i // 2, i % 2].imshow(binary, cmap="gray")
    ax[i // 2, i % 2].set_title(title)
    ax[i // 2, i % 2].axis("off")
plt.tight_layout()
plt.show()
```

Otsu



Triángulo



Isodata



Manual



- **Aclaración:**

En el canal S los píxeles de carretera son más oscuros que el fondo, por lo que, al igual que en b, utilizamos el operador `<` para.

- **Conclusiones:**

Alguna de las umbralizaciones sobre todos los canales y, por tanto, las imágenes binarias obtenidas parecen ser bastante efectivas para la segmentación de carreteras. Obviamente, la imagen binaria no se corresponde exactamente con lo que buscamos segmentar, pero creemos que si que puede ser útil para la creación del vector de características.

- En el canal b utilizaremos un umbral manual de 0.4
 - Es importante tener en cuenta que el umbral manual está adaptado a la imagen 1 con la que se ha probado y esto podría dar problemas con otras imágenes (cuando posteriormente imprimimos las características que usamos para una imagen aleatoria se comprueba que este umbral funciona bien)
- Para los canales H y S, el método de obtención de umbral Otsu, parece ofrecer los mejores resultados.

Cálculo de magnitud y dirección del gradiente

Definimos las funciones para calcular la magnitud y dirección del gradiente de la imagen.

En cuanto a la dirección del gradiente, este **se ha discretizado a un número concreto de direcciones** para poder trabajar con ella posteriormente.

```
In [ ]: ## Función para calcular el gradiente
def magnitud_gradiente(imagen, sigma=1):
    """Calcula el gradiente de una imagen usando el operador Sobel.

    Args:
        imagen (ndarray): Imagen de entrada.

    Returns:
        ndarray: Gradiente de la imagen.
    """
    ## Calculamos el gradiente usando el operador Sobel
    grad_x = ndi.gaussian_filter(imagen, sigma=sigma, order=(0, 1))
    grad_y = ndi.gaussian_filter(imagen, sigma=sigma, order=(1, 0))
    grad = np.sqrt(grad_x**2 + grad_y**2)
    return grad

## Función para discretizar los ángulos
def orientacion_gradiente(imagen, sigma=1, angulos_discretos=36):
    """Calcula la orientación del gradiente de una imagen.

    Args:
        imagen (ndarray): Imagen de entrada.

    Returns:
        ndarray: Orientación del gradiente de la imagen.
    """
```

```

"""
## Calculamos el gradiente usando el operador Sobel
grad_x = ndi.gaussian_filter(imagen, sigma=sigma, order=(0, 1))
grad_y = ndi.gaussian_filter(imagen, sigma=sigma, order=(1, 0))
## Calculamos la dirección del gradiente
grad_orientation = np.arctan2(grad_y, grad_x)
## Convertimos a grados y discretizamos
grad_orientation = np.degrees(grad_orientation)
grad_orientation = np.round((grad_orientation - grad_orientation.min()) /
                             (grad_orientation.max() - grad_orientation.min()) * angulos_discretos)

return grad_orientation

```

Aplicamos las funciones definidas para calcular la magnitud y dirección del gradiente a la imagen de satélite (en escala de gris) y visualizamos los resultados.

Probamos con **diferentes valores de sigma** para el cálculo de la gaussiana y con la discretización a un **número diferente de orientaciones** para la dirección del gradiente.

```

In [ ]: ## Convertimos la imagen a escala de grises
img_sat_gray = skimage.color.rgb2gray(img_sat)

## Calculamos el gradiente y la orientación con diferentes sigmas
gradientes = []
orientaciones = []
for sigma in [1, 2, 5]:
    for angulos_discretos in [8, 16, 36]:
        grad = magnitud_gradiente(img_sat_gray, sigma=sigma)
        orient = orientacion_gradiente(img_sat_gray, sigma=sigma, angulos_discretos=angulos_discretos)
        gradientes.append(grad)
        orientaciones.append(orient)

## Visualizamos los gradientes y orientaciones con un bucle
fig, ax = plt.subplots(9, 3, figsize=(10, 20))
for i, (grad, orient) in enumerate(zip(gradi

```

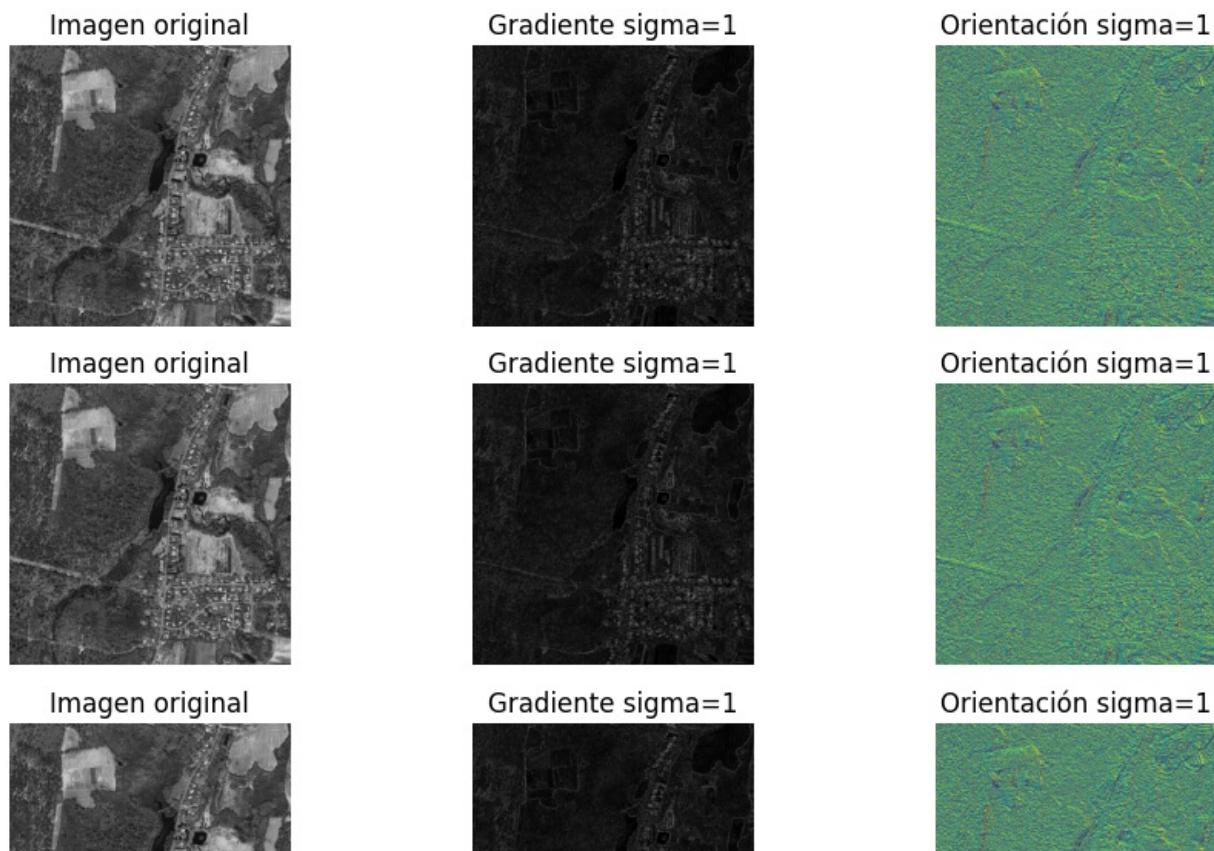




Imagen original



Gradiente sigma=2



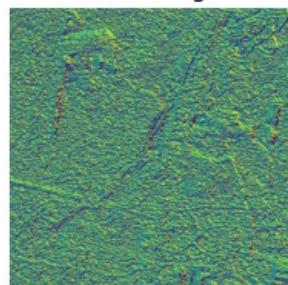
Orientación sigma=2



Imagen original



Gradiente sigma=2



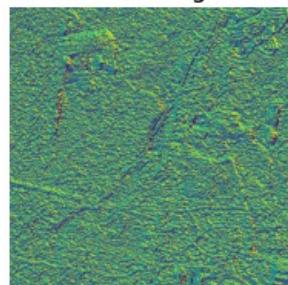
Orientación sigma=2



Imagen original



Gradiente sigma=2



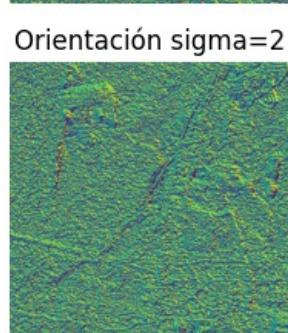
Orientación sigma=2



Imagen original



Gradiente sigma=2



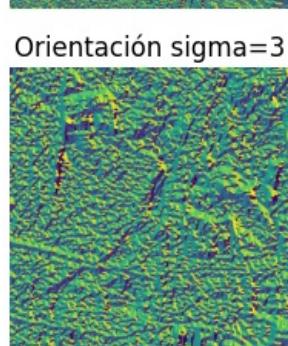
Orientación sigma=2



Imagen original



Gradiente sigma=3



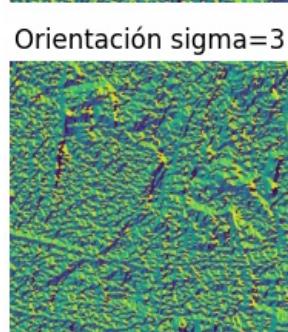
Orientación sigma=3



Imagen original



Gradiente sigma=3



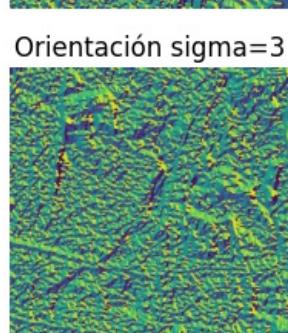
Orientación sigma=3



Imagen original



Gradiente sigma=3



Orientación sigma=3

Probamos a realizar el cálculo de la **magnitud y dirección del gradiente sobre los canales H y S** de la imagen en espacio HSV. Visualizamos los resultados y comparamos con el resultado obtenido sobre la imagen en escala de grises.

In [20]:

```
## Calculamos gradiente y orientación sobre los canales H y S de la imagen en HSV
img_h = img_hsv[:, :, 0]
img_s = img_hsv[:, :, 1]

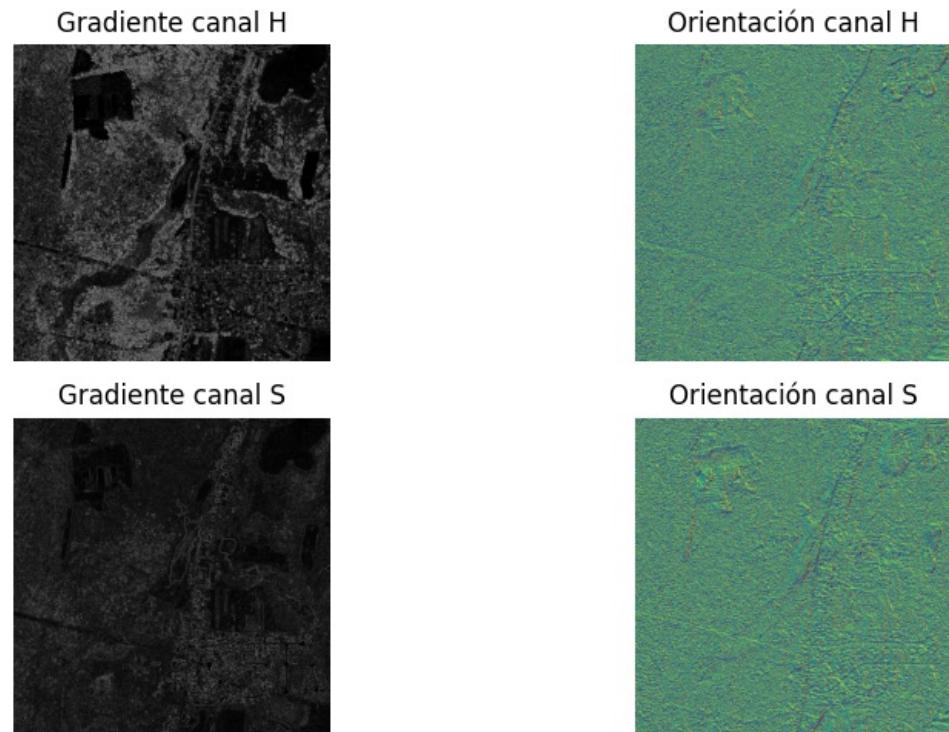
## Mostramos el tipo de dato de la imagen
print(f"Tipo de dato imagen H: {img_h.dtype} con valores entre {img_h.min()} y {img_h.max()}")
print(f"Tipo de dato imagen S: {img_s.dtype} con valores entre {img_s.min()} y {img_s.max()}")

## Calculamos el gradiente y la orientación de los canales H y S
grad_h = magnitud_gradiente(img_h, sigma=1)
grad_s = magnitud_gradiente(img_s, sigma=1)
orient_h = orientacion_gradiente(img_h, sigma=1, angulos_discretos=8)
orient_s = orientacion_gradiente(img_s, sigma=1, angulos_discretos=8)

## Visualizamos los gradientes y orientaciones de los canales H y S
fig, ax = plt.subplots(2, 2, figsize=(10, 5))
for i, (grad, orient) in enumerate(zip([grad_h, grad_s], [orient_h, orient_s])):
    ax[i, 0].imshow(grad, cmap="gray")
    ax[i, 0].set_title(f"Gradiente canal {[['H', 'S'][i]]}")
    ax[i, 0].axis("off")
    ax[i, 1].imshow(orient)
    ax[i, 1].set_title(f"Orientación canal {[['H', 'S'][i]]}")
    ax[i, 1].axis("off")
plt.tight_layout()
plt.show()
```

Tipo de dato imagen H: float64 con valores entre 0.0 y 0.9980276134122288

Tipo de dato imagen S: float64 con valores entre 0.0 y 1.0



- **Conclusión:**

Visualmente es complicado determinar si la magnitud y dirección del gradiente de los canales H y S ofrecen mejores resultados que el cálculo sobre la imagen en escala de grises.

Por tanto, se probarán ambos métodos y se compararán los resultados obtenidos en el clasificador.

- **Nota:**

Finalmente se opta por trabajar con la imagen en escala de grises, ya que la umbralización con esta característica es más efectiva (entrenamiento guardado en *model.keras*)

Operador de Canny sobre la imagen en escala de grises y sobre los canales H y S de la imagen en espacio HSV para la detección de bordes.

In [21]:

```
## Aplicamos Canny sobre la imagen en escala de grises
img_gray = skimage.color.rgb2gray(img_sat)
edges = skimage.feature.canny(img_gray, sigma=3, low_threshold=0.01, high_threshold=0.25)
```

```

## Visualizamos el resultado
fig, ax = plt.subplots(1, 2, figsize=(10, 5))
ax[0].imshow(img_gray, cmap="gray")
ax[0].set_title("Imagen original")
ax[0].axis("off")
ax[1].imshow(edges, cmap="gray")
ax[1].set_title("Canny")
ax[1].axis("off")
plt.tight_layout()
plt.show()

## Aplicamos Canny sobre el canal H
img_h = img_hsv[:, :, 0]
edges = skimage.feature.canny(img_h, sigma=3, low_threshold=0.01, high_threshold=0.2)

## Visualizamos el resultado
fig, ax = plt.subplots(1, 2, figsize=(10, 5))
ax[0].imshow(img_h, cmap="gray")
ax[0].set_title("Canal H")
ax[0].axis("off")
ax[1].imshow(edges, cmap="gray")
ax[1].set_title("Canny canal H")
ax[1].axis("off")
plt.tight_layout()
plt.show()

## Aplicamos Canny sobre el canal S
img_s = img_hsv[:, :, 1]
edges = skimage.feature.canny(img_s, sigma=3, low_threshold=0.05, high_threshold=0.3)

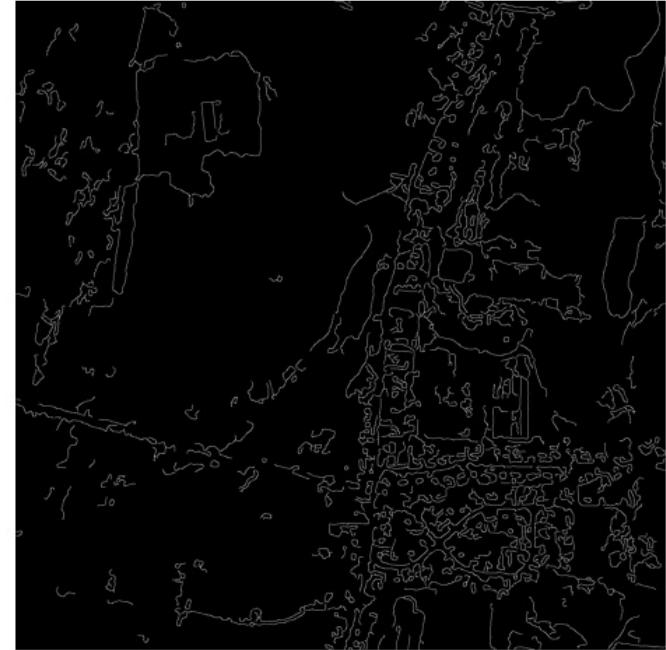
## Visualizamos el resultado
fig, ax = plt.subplots(1, 2, figsize=(10, 5))
ax[0].imshow(img_s, cmap="gray")
ax[0].set_title("Canal S")
ax[0].axis("off")
ax[1].imshow(edges, cmap="gray")
ax[1].set_title("Canny canal S")
ax[1].axis("off")
plt.tight_layout()
plt.show()

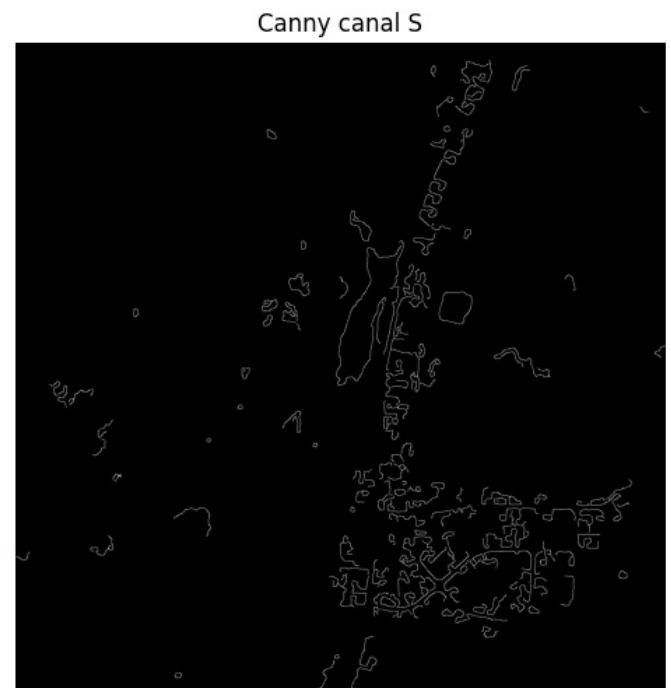
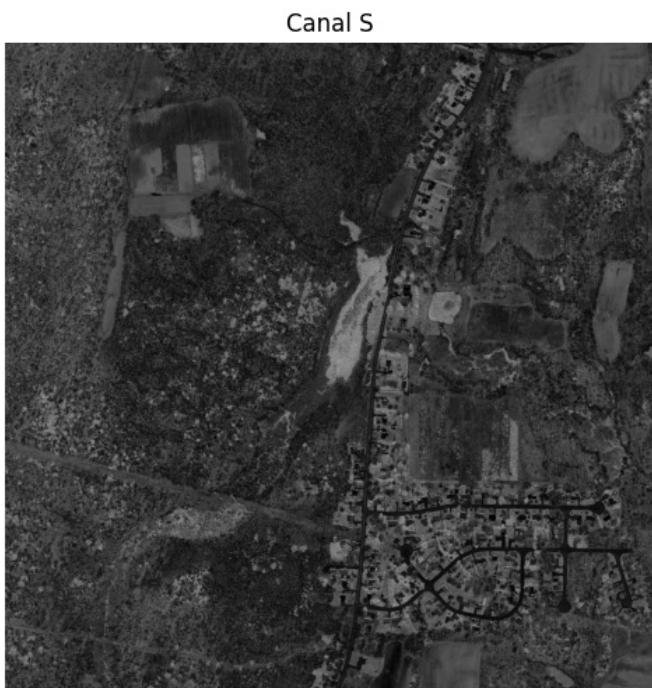
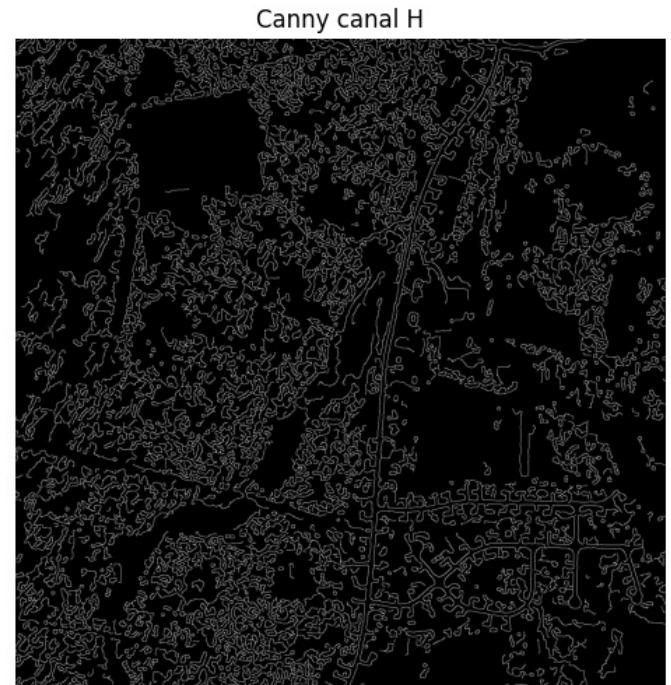
```

Imagen original



Canny





- **Conclusión:**

El detector de bordes de Canny no parece ofrecer unos resultados óptimos para la detección de carreteras. La dificultad para la elección de los parámetros de sigma y umbrales de histeresis es un problema a la hora de utilizar este operador.

1.3. Vector de características

Tras la experimentación con diferentes filtros y técnicas de procesamiento de imágenes, se ha decidido que el vector de características estará formado por los siguientes elementos:

- Canales **R, G y B** de la imagen original para obtener información de color.
- Canales **H y S** de la imagen en espacio HSV para obtener información de color.
- **Umbralización manual sobre el canal b** (previamente expandido)
- **Umbralización de Otsu sobre los canales H y S**
- **Dirección y magnitud del gradiente** de la imagen en escala de grises.

Se define entonces la **función para la creación del vector de características**. Esta función recibe como entrada la imagen de satélite y devuelve un vector de características para cada píxel de la imagen.

La función devuelve un array de tamaño (1500, 1500, 10) donde cada píxel tiene un vector de características de tamaño 10.

```
In [ ]: def caracteristicas(imagen):
    """
        Calcula las características de una imagen de satélite para la detección de carreteras.

    Args:
        imagen (ndarray): Imagen de entrada.

    Returns:
        array: Características de la imagen.
        - Canales R, G y B de la imagen original para obtener información de color.
        - Umbralización manual sobre el canal b (previamente expandido)
        - Umbralización de Otsu sobre el canal H y S
        - Dirección y magnitud del gradiente de la imagen original con un sigma de 1 y discretización a 36 direcciones
    """

    ## Convertimos la imagen a escala de grises
    imagen_gray = skimage.color.rgb2gray(imagen)

    ## Calculamos el gradiente y la orientación con un sigma de 1 y discretización a 36 direcciones
    grad = magnitud_gradiente(imagen_gray, sigma=1)
    orient = orientacion_gradiente(imagen_gray, sigma=1, angulos_discretos=8)

    ## Convertimos la imagen a HSV
    imagen_hsv = skimage.color.rgb2hsv(imagen)

    ## Obtenemos los canales H y S
    canal_h = imagen_hsv[:, :, 0]
    canal_s = imagen_hsv[:, :, 1]

    ## Calculamos el gradiente y la orientación del canal H
    # grad_h = magnitud_gradiente(canal_h, sigma=1)
    # orient_h = orientacion_gradiente(canal_h, sigma=1, angulos_discretos=8)

    ## Umbralizamos los canales H y S usando Otsu
    umbral_h = skimage.filters.threshold_otsu(canal_h)
    umbral_s = skimage.filters.threshold_otsu(canal_s)

    ## Convertimos la imagen a LAB
    imagen_lab = skimage.color.rgb2lab(imagen)
    # Obtenemos el canal b
    canal_b = imagen_lab[:, :, 2]
    # Normalizamos el canal b al rango [0, 1]
    canal_b = (canal_b + 128) / (127 + 128)
    # Expandimos el histograma del canal b
    canal_b = exposure.rescale_intensity(canal_b, in_range='image', out_range=(0, 1))
    # Umbralizamos el canal b usando un umbral manual
    umbral_b = 0.4

    ## Creamos una lista con las características
    caracteristicas = [imagen[:, :, i] for i in range(3)] + \
                      [canal_h] + \
                      [canal_s] + \
                      [canal_b < umbral_b] + \
                      [canal_h > umbral_h] + \
                      [canal_s > umbral_s] + \
                      [grad, orient]

    ## Reshape de las características para que tengan la misma forma
    caracteristicas = np.array(caracteristicas)
    caracteristicas = np.transpose(caracteristicas, (1, 2, 0))

    return caracteristicas
```

Extraemos el vector de características de todas las imágenes haciendo uso de la función definida anteriormente.

```
In [6]: ## Calculamos los vectores de características para cada imagen
```

```

características_sat = [características(image) for image in sat]

## Convertimos a numpy array
características_sat = np.array(características_sat)
print(f"Dimensiones de las características: {características_sat.shape}")

```

Dimensiones de las características: (20, 1500, 1500, 10)

- **Conclusión:**

Las dimensiones de nuestro conjunto de datos son:

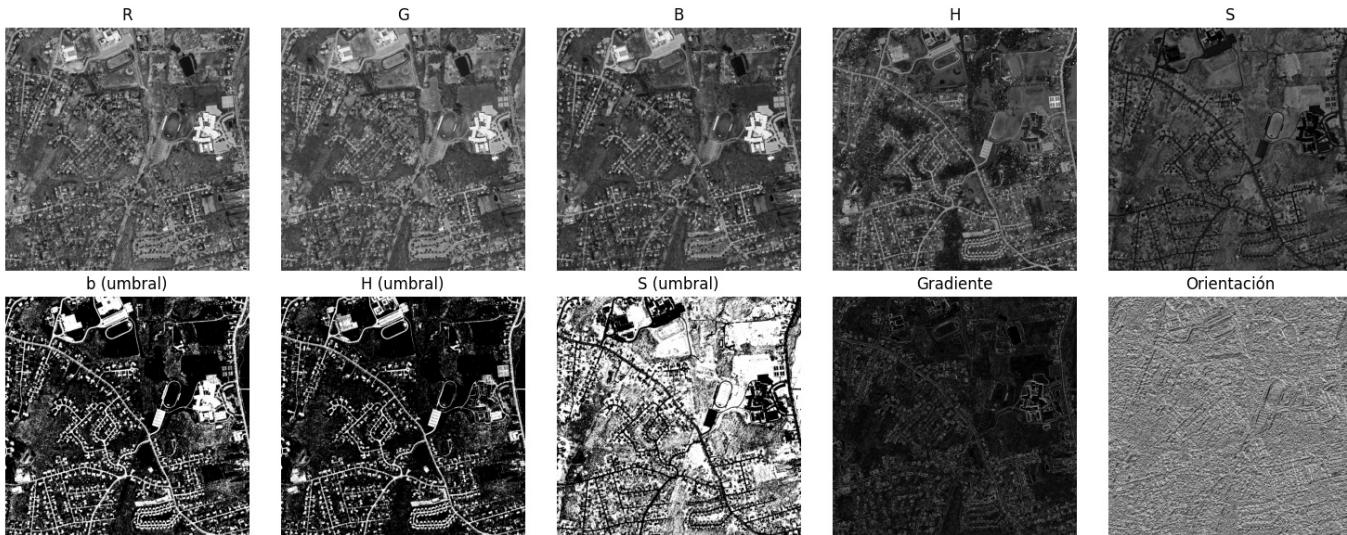
- 20 imágenes
- 1500x1500 píxeles cada imagen
- 10 características por píxel

Visualizamos las **imágenes representativas de cada una de las características** que componen el vector de características. Cada píxel tendrá como característica el nivel de gris mostrado en cada una de estas imágenes.

```
In [9]: ## Visualizamos las características de una imagen random
random_index = np.random.randint(0, len(características_sat))
print(f"Visualizando características de la imagen {random_index}")

canales = ["R", "G", "B", "H", "S", "b (umbral)", "H (umbral)", "S (umbral)", "Gradiente", "Orientación"]
fig, ax = plt.subplots(2, 5, figsize=(15, 6))
for i in range(10):
    ax[i // 5, i % 5].imshow(características_sat[random_index, :, :, i], cmap="gray")
    ax[i // 5, i % 5].set_title(f"{canales[i]}")
    ax[i // 5, i % 5].axis("off")
plt.tight_layout()
plt.show()
```

Visualizando características de la imagen 18



1.4. Creación de los conjuntos de entrenamiento y test

Dividimos las imágenes de satélite en un conjunto de entrenamiento (80%) y un conjunto de test (20%). Hemos prescindido del uso de funciones para la división de los conjuntos de entrenamiento y test, ya que el tamaño de las imágenes es pequeño pero dispar y no es necesario realizar una división aleatoria.

Realmente, si el objetivo solamente es segmentar las carreteras en las imágenes concretas, la división de datos no sería necesaria, sería suficiente con entrenar un modelo con todas las imágenes sin importar que este sobreentrene, es más sería incluso útil un sobreentrenamiento.

En el trabajo **decidimos dividir los datos para tratar de entrenar un modelo que pueda generalizar** a otras imágenes de carreteras, no solamente a las imágenes concretas que se nos ofrecen.

```
In [10]: ## Dividimos en entrenamiento y test las imágenes
x_train = características_sat[:int(len(características_sat)*0.8)]
x_test = características_sat[int(len(características_sat)*0.8):]

## Imprimimos las dimensiones de los datos
print(f"Dimensiones de x_train: {x_train.shape}")
print(f"Dimensiones de x_test: {x_test.shape}")
```

Dimensiones de x_train: (16, 1500, 1500, 10)
Dimensiones de x_test: (4, 1500, 1500, 10)

A partir del *groud truth* obtenemos un único **vector de etiquetas** para cada imagen. Los píxeles que son carretera se etiquetan como 1 y el resto como 0 (valores que se corresponden con la imagen binaria).

Dividimos las etiquetas que se corresponden a las imágenes de entrenamiento y test.

```
In [11]: ## Convertimos las etiquetas a numpy array de tamaño (20,1500,1500,1)
gt = np.array(gt)
gt = gt.reshape(gt.shape[0], gt.shape[1], gt.shape[2], 1)

## Dividimos en entrenamiento y test las etiquetas
y_train = gt[:int(len(gt)*0.8)]
y_test = gt[int(len(gt)*0.8):]

## Imprimimos las dimensiones de los datos
print(f"Dimensiones de y_train: {y_train.shape}")
print(f"Dimensiones de y_test: {y_test.shape}")

Dimensiones de y_train: (16, 1500, 1500, 1)
Dimensiones de y_test: (4, 1500, 1500, 1)
```

- **Aclaración:**

Las imágenes de satélite y el *groud truth* están ordenadas de la misma forma. La imagen 0 de satélite corresponde a la imagen 0 de *ground truth*, la imagen 1 de satélite corresponde a la imagen 1 de *ground truth*, etc.

Al dividir de forma que las 16 primeras imágenes de satélite y *ground truth* se corresponden con el conjunto de entrenamiento y las 4 últimas imágenes se corresponden con el conjunto de test, se mantiene la correspondencia entre las imágenes de satélite y *ground truth*.

1.5. Creación del clasificador

La función siguiente servirá para crear nuestro **clasificador** binario (carretera / no carretera) haciendo uso de una **red convolucional**.

```
In [ ]: def modelo1(input_shape):
    inputs = layers.Input(shape=input_shape)

    x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(inputs)
    x = layers.Conv2D(16, (3, 3), activation='relu', padding='same')(x)
    x = layers.Conv2D(1, (1, 1), activation='sigmoid', padding='same')(x)

    outputs = layers.Reshape((input_shape[0], input_shape[1]))(x)
    model = models.Model(inputs=inputs, outputs=outputs)

    model.compile(optimizer=optimizers.Adam(learning_rate=0.001),
                  loss=losses.BinaryCrossentropy(),
                  metrics=[metrics.BinaryAccuracy(name='accuracy'),
                           metrics.Precision(name='precision'),
                           metrics.Recall(name='recall'),
                           metrics.AUC(name='auc')])

    return model
```

- **Aclaración:**

Nuestra **métrica fundamental para evaluar el rendimiento será AUC** (Area Under Curve). Nuestro problema es un claro ejemplo de un problema de clasificación binaria en el que las clases están altamente desbalanceadas, la gran mayoría de los píxeles no son carretera.

Por este motivo, la *accuracy* no es una métrica adecuada ya que puede ser engañosa. Por ejemplo, si el clasificador predice que todos los píxeles son fondo, la *accuracy* será igualmente alta (habrá acertado en la mayoría de los píxeles) pero no habrá aprendido nada.

Sin embargo la métrica AUC es aquella que nos indica el ratio entre falsos positivos y verdaderos positivos, por lo que es una métrica más adecuada para nuestro problema. Será 1 si la clasificación es perfecta y 0.5 si la clasificación es aleatoria.

- **Nota:**

Hay que tener en cuenta que el uso de la métrica no evita el problema de desbalanceo, simplemente nos ayudará a evaluar el rendimiento del clasificador.

En caso de querer mitigar el problema de desbalanceo, se podrían aplicar técnicas como el oversampling, el uso de pesos dándole más importancia a la clase minoritaria (carretera) o la definición de funciones de pérdida específicas para este

problema.

En este caso, aunque se ha probado a modificar la función de perdida en alguno de los modelos creados (presente en la parte final del Notebook), hemos llegado a un entrenamiento satisfactorio sin esta técnica. De todas formas, debemos ser conscientes del problema existente.

Creamos el clasificador y lo entrenamos con el conjunto de entrenamiento. Este modelo recibe como entrada directamente la matriz de 1500x1500x10, no se realiza un *flatten* de la imagen. Cada una de las imágenes constituye un batch.

```
In [ ]: ## Creamos el modelo
INPUT_SHAPE = (1500, 1500, 10)
model = model1(input_shape=INPUT_SHAPE)

## Resumen del modelo
model.summary()

## Entrenamos el modelo
history = model.fit(x_train, y_train, epochs=5, batch_size=1)

## Guardamos el modelo
model.save("modelo1.keras")
```

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 1500, 1500, 10)	0
conv2d (Conv2D)	(None, 1500, 1500, 32)	2,912
conv2d_1 (Conv2D)	(None, 1500, 1500, 16)	4,624
conv2d_2 (Conv2D)	(None, 1500, 1500, 1)	17
reshape (Reshape)	(None, 1500, 1500)	0

Total params: 7,553 (29.50 KB)

Trainable params: 7,553 (29.50 KB)

Non-trainable params: 0 (0.00 B)

Epoch 1/5

16/16 16s 844ms/step - accuracy: 0.6376 - auc: 0.5315 - loss: 0.5849 - precision: 0.0437 - recall: 0.4353

Epoch 2/5

16/16 14s 892ms/step - accuracy: 0.9687 - auc: 0.5499 - loss: 0.1498 - precision: 0.0000e+00 - recall: 0.0000e+00

Epoch 3/5

16/16 15s 957ms/step - accuracy: 0.9655 - auc: 0.7226 - loss: 0.1408 - precision: 0.0000e+00 - recall: 0.0000e+00

Epoch 4/5

16/16 19s 1s/step - accuracy: 0.9727 - auc: 0.8282 - loss: 0.1059 - precision: 0.0000e+00 - recall: 0.0000e+00

Epoch 5/5

16/16 20s 1s/step - accuracy: 0.9647 - auc: 0.8762 - loss: 0.1142 - precision: 0.4706 - recall: 4.7830e-07

- **Aclaración:**

El entrenamiento aquí mostrado **no se corresponde con el entrenamiento del método final**. Esta celda anterior se ha ejecutado con diferentes datos de entrada (diferentes características), diferentes arquitecturas (número de capas, neuronas por capa, *batch normalization*), diferentes funciones de pérdida. Aquel que ha ofrecido mejores resultados, y equilibrio entre coste computacional y rendimiento, es el que se encuentra en la celda de abajo.

- **Conclusión**

Creamos una red convolucional con dos capas de 32 y 16 neuronas respectivamente y activación relu y otra capa de 1 neurona y activación sigmoid. Se ha probado a definir una arquitectura más compleja pero el aumento de la complejidad no ofrecía mejoras significativas y aumentaba considerablemente el coste computacional, resultando prácticamente imposible el entrenamiento. Se ha utilizado la función de pérdida `binary_crossentropy` y el optimizador Adam. El entrenamiento se ha realizado durante 5 épocas. La métrica utilizada para evaluar el rendimiento del clasificador es AUC (Area Under Curve).

La salida de la red es un valor entre 0 y 1 que indica la probabilidad de que el píxel sea carretera, más adelante se tratará con esto para obtener la segmentación binaria.

Cargamos el clasificador `model.keras` y lo evaluamos con el conjunto de test. Obtenemos las métricas de evaluación. Este modelo aquí cargado es el que se utilizará para la segmentación de carreteras en las imágenes de prueba.

```
In [12]: from tensorflow.keras.models import load_model
```

```
## Cargamos el modelo
model = load_model("modelo.keras")
model.summary()

## Evaluación en el conjunto de test
score = model.evaluate(x_test, y_test)
print(f"Test loss: {score[0]}")
print(f"Test accuracy: {score[1]}")
print(f"Test AUC: {score[4]}")
```

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 1500, 1500, 10)	0
conv2d (Conv2D)	(None, 1500, 1500, 32)	2,912
conv2d_1 (Conv2D)	(None, 1500, 1500, 16)	4,624
conv2d_2 (Conv2D)	(None, 1500, 1500, 1)	17
reshape (Reshape)	(None, 1500, 1500)	0

Total params: 22,661 (88.52 KB)

Trainable params: 7,553 (29.50 KB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 15,108 (59.02 KB)

```
1/1 ━━━━━━━━ 5s 5s/step - accuracy: 0.8603 - auc: 0.8877 - loss: 0.3162 - precision: 0.2290 - recall : 0.7631
Test loss: 0.3162466883659363
Test accuracy: 0.8602579832077026
Test AUC: 0.8877443671226501
```

- **Conclusión:**

Obtenemos un AUC de 0.89, lo que indica que el clasificador tiene un buen rendimiento. Se evaluará de forma cualitativa la segmentación obtenida en las imágenes de prueba.

Definimos una función para **visualizar los resultados**. Se mostrará la imagen de satélite, el *ground truth* y la segmentación obtenida por el clasificador. Cabe destacar que la salida de nuestro modelo, y por tanto de la predicción es una probabilidad entre 0 y 1 para cada píxel, por lo que se aplicará un umbral para convertir la salida a una imagen binaria.

```
In [ ]: def visualizar_prediccion_gt_og(vector, gt, prediccion, umbral=0.5):
```

```
    ...
    Función para visualizar la imagen original, la predicción binarizada y la gt
```

```
Args:
```

```
    vector (array): vector (1500, 1500, 8) que contiene las características de la imagen (las 3 primeras
    gt (array): imagen gt (1500, 1500) que contiene la máscara de la imagen
    prediccion (array): imagen predicción (1500, 1500) que contiene la probabilidad de que el pixel sea car
    umbral (int): umbral para binarizar la predicción (por defecto 0.5)
```

```
Returns:
```

```
    None
    ...
    ## Convertimos el vector a una imagen RGB
    img_rgb = np.zeros((1500, 1500, 3))
    img_rgb[:, :, 0] = vector[:, :, 0] # Canal R
    img_rgb[:, :, 1] = vector[:, :, 1] # Canal G
    img_rgb[:, :, 2] = vector[:, :, 2] # Canal B
```

```
    ## Binarizamos la predicción
    prediccion_binaria = np.where(prediccion > umbral, 1, 0)
```

```
    ## Visualizamos la imagen original, la predicción y la gt
```

```
    fig, ax = plt.subplots(1, 3, figsize=(15, 5))
```

```
    ax[0].imshow(img_rgb)
```

```
    ax[0].set_title("Imagen original")
```

```
    ax[0].axis("off")
```

```
    ax[1].imshow(gt, cmap="gray")
```

```
    ax[1].set_title("Ground Truth")
```

```
    ax[1].axis("off")
```

```

    ax[2].imshow(prediccion_binaria, cmap="gray")
    ax[2].set_title("Predicción binarizada")
    ax[2].axis("off")
    plt.tight_layout()
    plt.show()

    return prediccion_binaria

```

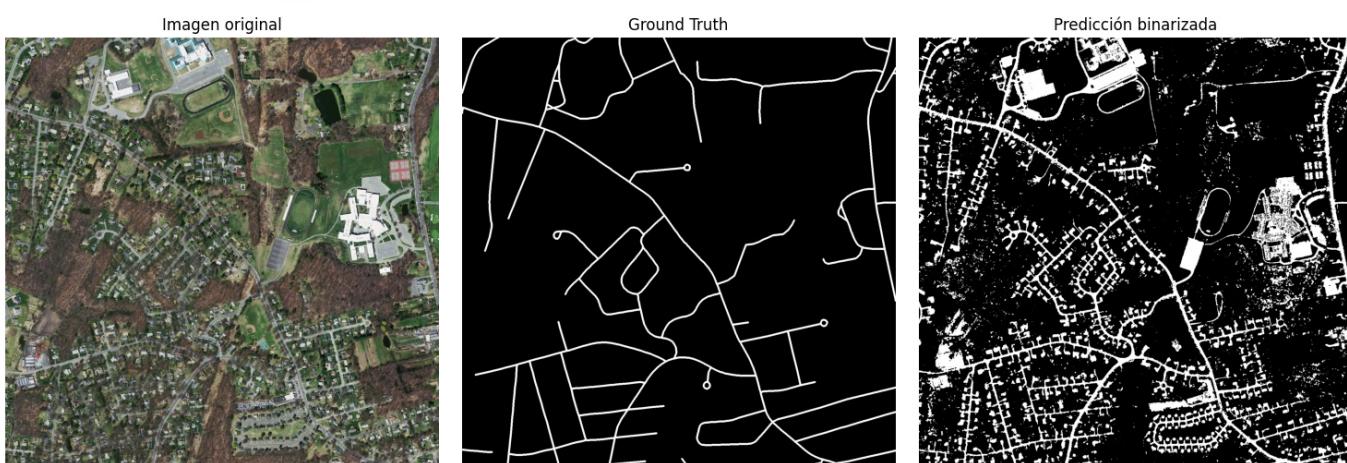
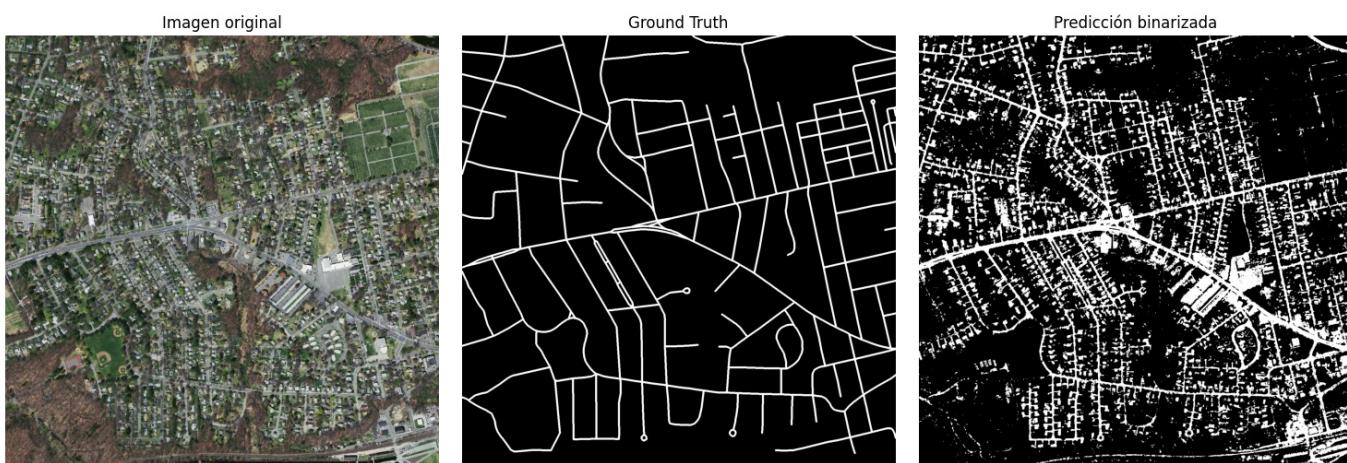
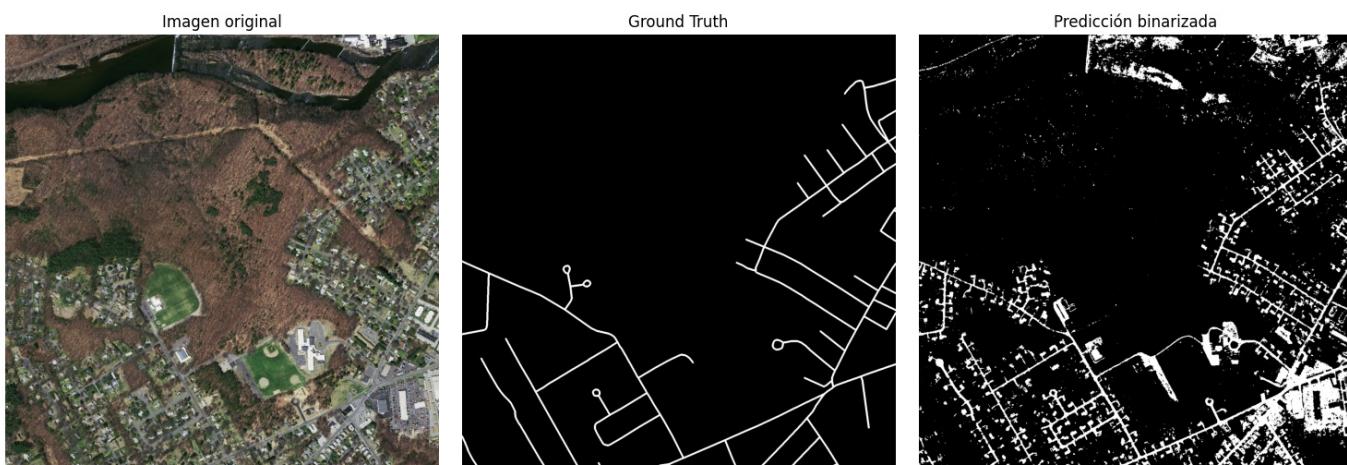
Predecimos la segmentación en todas nuestras imágenes de test.

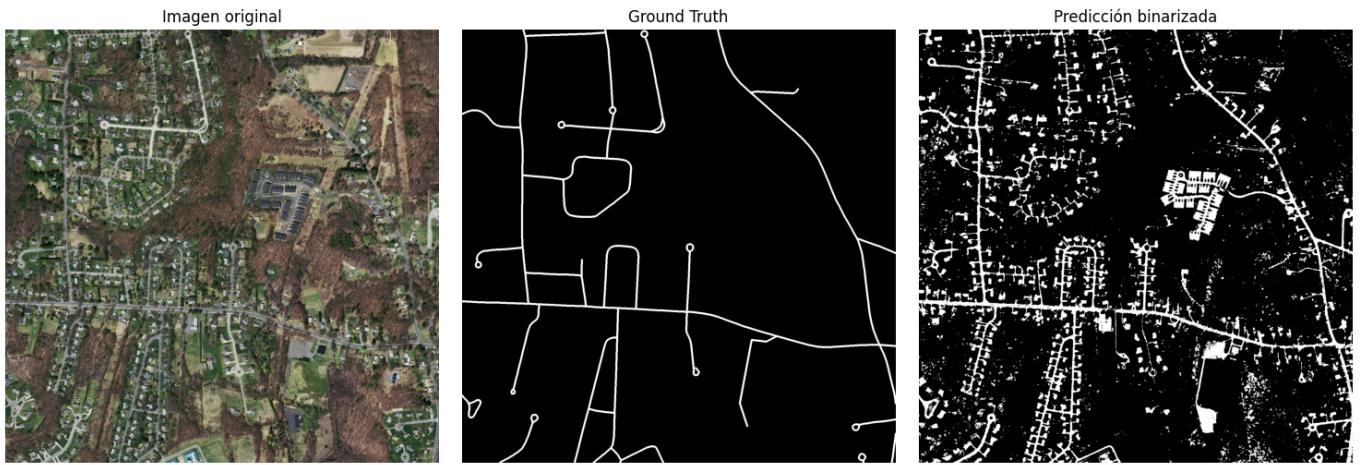
In [14]: `## Predicción sobre el conjunto de test
prediccion = model.predict(x_test)`

1/1 3s 3s/step

Visualizamos la segmentación obtenida por el clasificador, es decir, la predicción de la red convolucional. Se aplicará un umbral de 0.5 para convertir la salida a una imagen binaria.

In [15]: `for idx in range(len(x_test)):
 ## Visualizamos la imagen original, la predicción y la gt
 visualizar_prediccion_gt_og(x_test[idx], y_test[idx], prediccion[idx])`





- **Conclusión:**

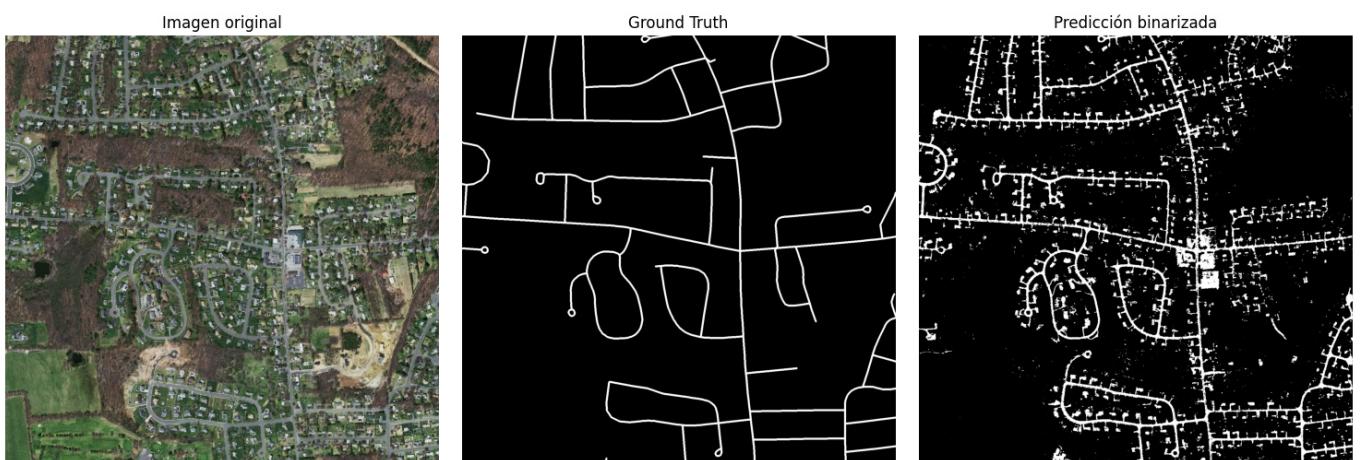
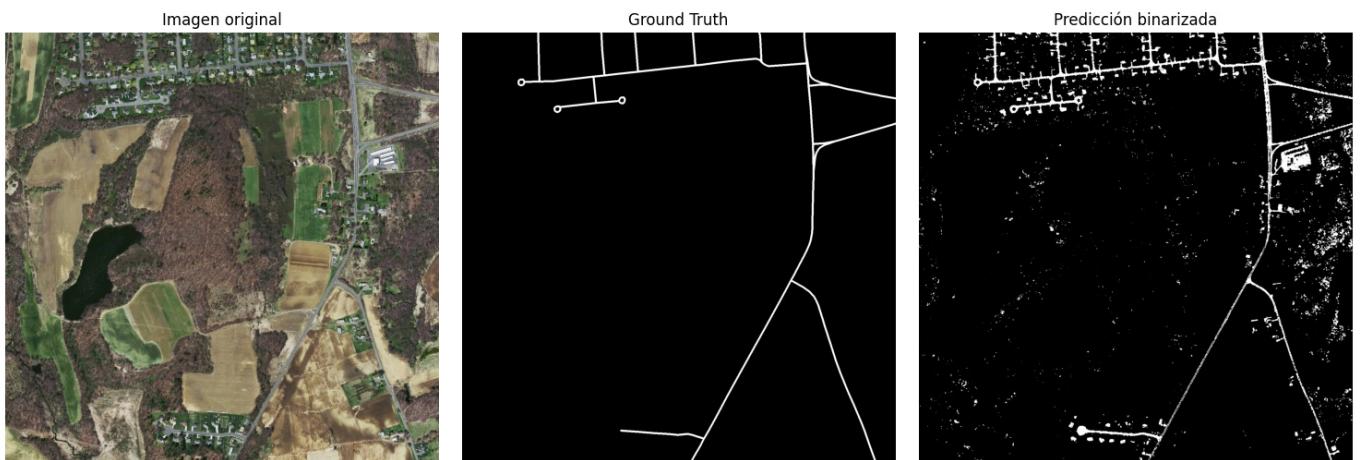
Estamos observando la segmentación en las imágenes de prueba, es decir aquellas de las cuales el modelo no conoce el *grand truth* y no ha utilizado para su aprendizaje. Teniendo esto en cuenta, la segmentación parece bastante buena, se discutirán más adelante los puntos débiles y puntos fuertes del modelo.

Probamos a **visualizar la segmentación obtenida** en algunas de las imágenes del **conjunto de entrenamiento**, aquellas cuyas características y *ground truth* han sido utilizadas para el entrenamiento del modelo.

```
In [ ]: predicción = model.predict(x_train[0:3, :, :, :])
```

1/1 ━━━━━━━━ 3s 3s/step

```
In [15]: _ = visualizar_prediccion_gt_og(x_train[1], y_train[1], predicción[1], umbral=0.6)
_ = visualizar_prediccion_gt_og(x_train[2], y_train[2], predicción[2], umbral=0.6)
```



- **Conclusión:**

La segmentación resulta satisfactoria, la carretera es claramente visible. Obviamente, no es idéntica a la del *ground truth*, pero la segmentación es bastante buena.

Tenemos un modelo que no ha sobreentrenado a las imágenes de entrenamiento y ofrece buenos resultados en ambos conjuntos de imágenes.

1.7. Post-procesado de la segmentación obtenida

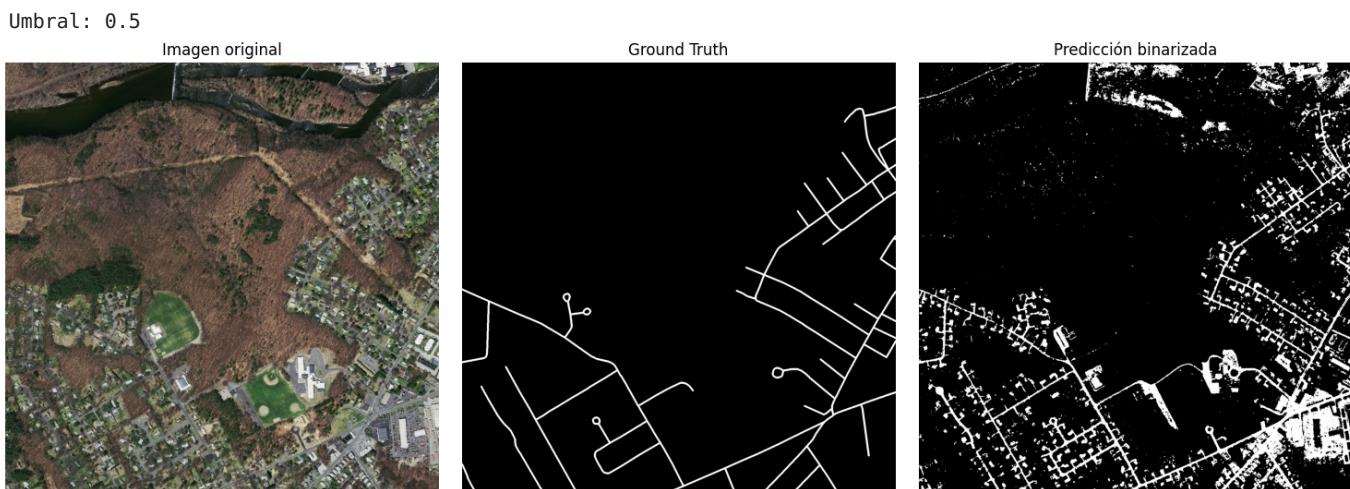
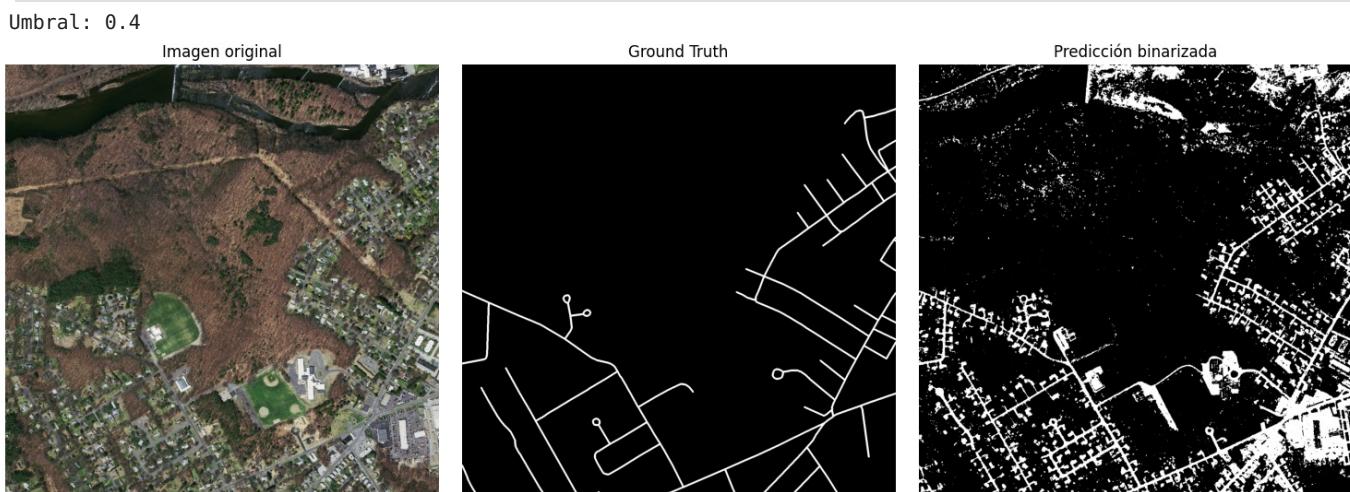
Trataremos de **mejorar la segmentación obtenida mediante un post-procesado**. Para ello, probaremos una serie de técnicas y elegiremos aquellas que ofrezcan un buen resultado.

Trabajaremos con la imagen de test 0 y su correspondiente *ground truth* para valorar el post-procesado de la segmentación obtenida.

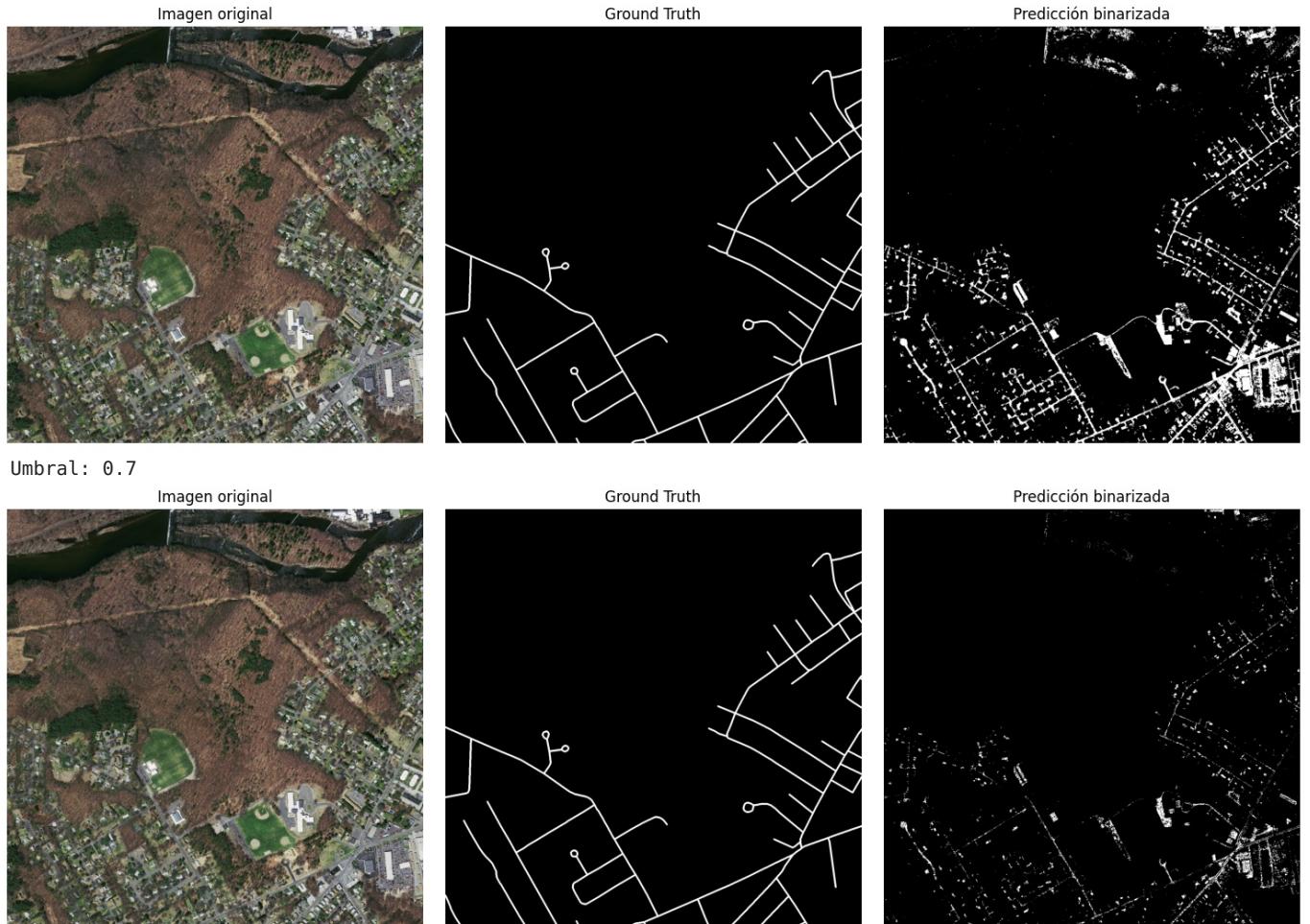
```
In [16]: pred_0 = prediccion[0]
gt_0 = y_test[0]
img_0 = x_test[0]
```

Experimentación de mejor **umbral de clasificación**. En principio, el umbral 0.5 es el acorde a la probabilidad de la red convolucional. Sin embargo, probamos si la modificación de este umbral para la binarización mejora la segmentación obtenida.

```
In [17]: for umbral in [0.4, 0.5, 0.6, 0.7]:
    print(f"Umbral: {umbral}")
    visualizar_prediccion_gt_og(img_0, gt_0, pred_0, umbral=umbral)
```



Umbral: 0.6



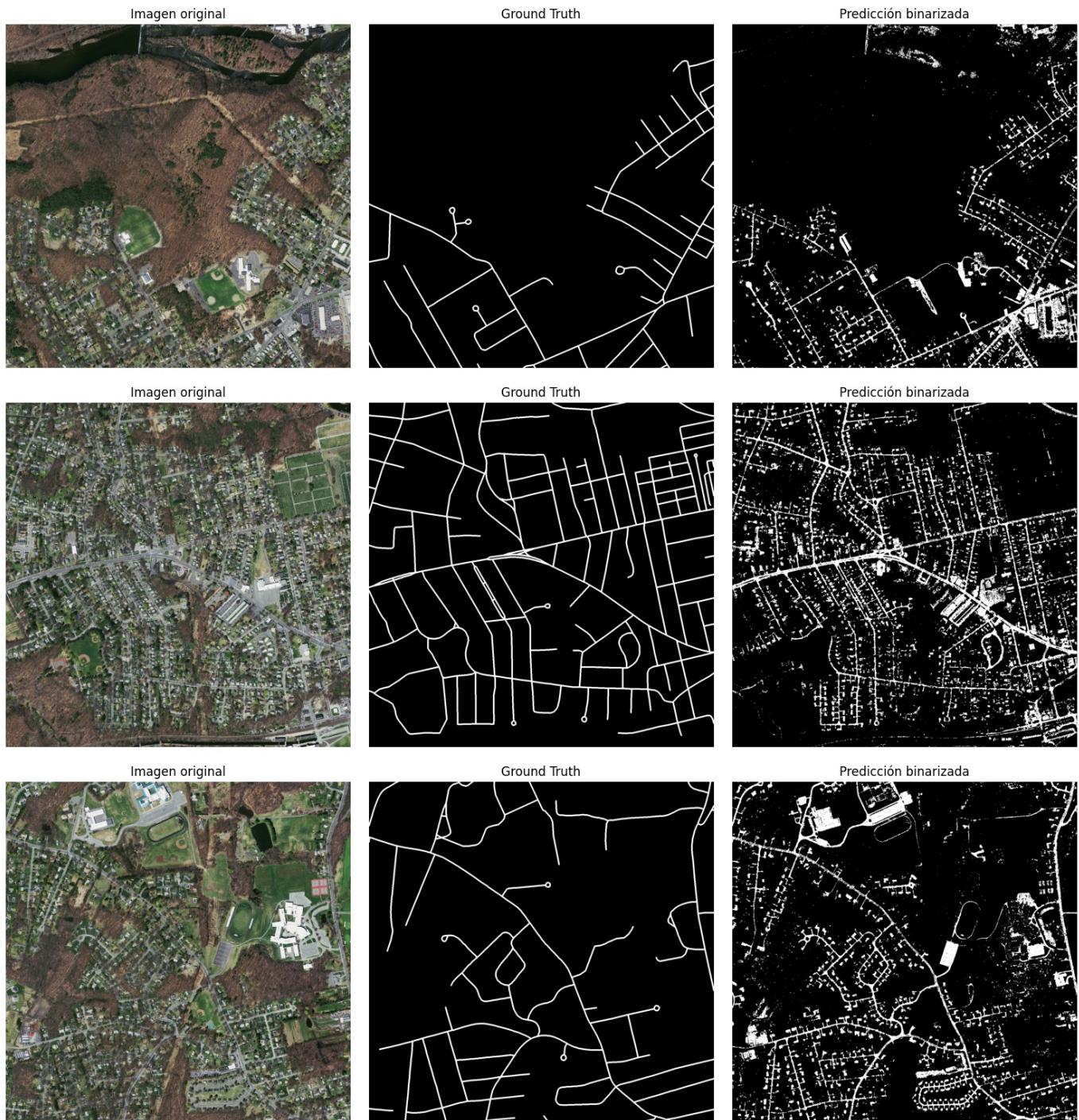
- **Conclusión:**

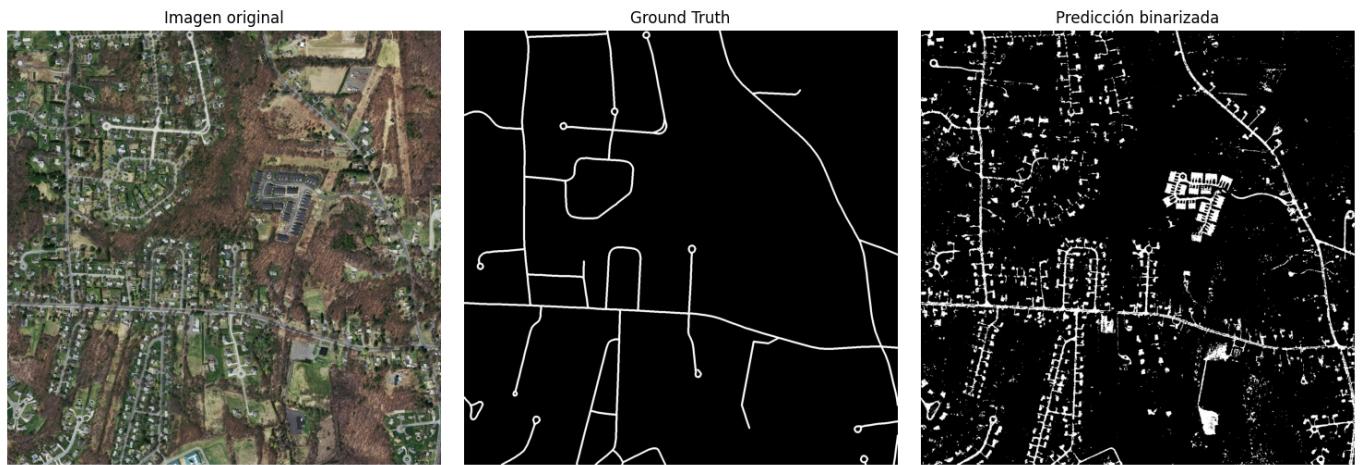
Los umbrales de 0.5 y 0.6 parecen ofrecer los mejores resultados. Con un umbral de 0.7 se pierden muchos píxeles de carretera y con un umbral de 0.4 se añaden muchos píxeles de ruido.

Decidimos quedarnos con el umbral 0.6 y aplicar más post-procesado a la imagen binaria obtenida.

```
In [18]: ## Visualizamos todas las imágenes con el umbral 0.6
print("Visualizando imágenes con umbral 0.6")
for idx in range(len(x_test)):
    ## Visualizamos la imagen original, la predicción y la gt
    visualizar_prediccion_gt_og(x_test[idx], y_test[idx], prediccion[idx], umbral=0.6)
```

Visualizando imágenes con umbral 0.6





- **Conclusión:**

A la vista del resto de imágenes, el umbral 0.6 si parece ofrecer un buen resultado para todas las predicciones.

El siguiente paso será la **identificación de componentes conexos**. Para ello, se aplicará un etiquetado de componentes conexos a la imagen binaria obtenida por el clasificador. A partir de este etiquetado, se eliminarán los componentes de tamaño menor a un umbral definido.

```
In [ ]: ## Identificación de componentes conexas con skimage
from skimage.measure import label, regionprops

def componentes_conexas(imagen, umbral=0.6):
    """
    Función para identificar componentes conexas en una imagen binarizada.

    Args:
        imagen (ndarray): Imagen binarizada de entrada.
        umbral (float): Umbral para binarizar la imagen.

    Returns:
        list: Lista de regiones conectadas.
    """
    ## Binarizamos la imagen
    imagen_binaria = np.where(imagen > umbral, 1, 0)

    ## Identificamos los componentes conexos
    etiquetas = label(imagen_binaria)

    ## Obtenemos las propiedades de las regiones conectadas
    regiones = regionprops(etiquetas)

    return regiones

## Eliminación de componentes pequeñas
def eliminar_componentes_pequeñas(imagen, umbral=0.6, area_minima=100):
    """
    Función para eliminar componentes pequeñas en una imagen binarizada.

    Args:
        imagen (ndarray): Imagen binarizada de entrada.
        umbral (float): Umbral para binarizar la imagen.
        area_minima (int): Área mínima para conservar un componente.

    Returns:
        ndarray: Imagen con componentes pequeñas eliminadas.
```

```

"""
## Binarizamos la imagen
imagen_binaria = np.where(imagen > umbral, 1, 0)

## Identificamos los componentes conexos
etiquetas = label(imagen_binaria)

## Obtenemos las propiedades de las regiones conectadas
regiones = regionprops(etiquetas)

## Creamos una nueva imagen para almacenar los componentes grandes
imagen_filtrada = np.zeros_like(imagen_binaria)

## Recorremos las regiones y conservamos solo las que cumplen el área mínima
for region in regiones:
    if region.area >= area_minima:
        imagen_filtrada[etiquetas == region.label] = 1

return imagen_filtrada

```

In [23]:

```

## Aplicamos la función de detección de componentes conexas
regiones = componentes_conexas(pred_0, umbral=0.6)
print(f"Cantidad de componentes conexas: {len(regiones)}")
print(f"Área mínima de los componentes: {min([region.area for region in regiones])}")
print(f"Área máxima de los componentes: {max([region.area for region in regiones])}")

## Comprobamos cuantos elementos de menos de 50 píxeles hay
print(f"Cantidad de componentes conexas menores a 70 píxeles: {len([region for region in regiones if region.area < 70])}")

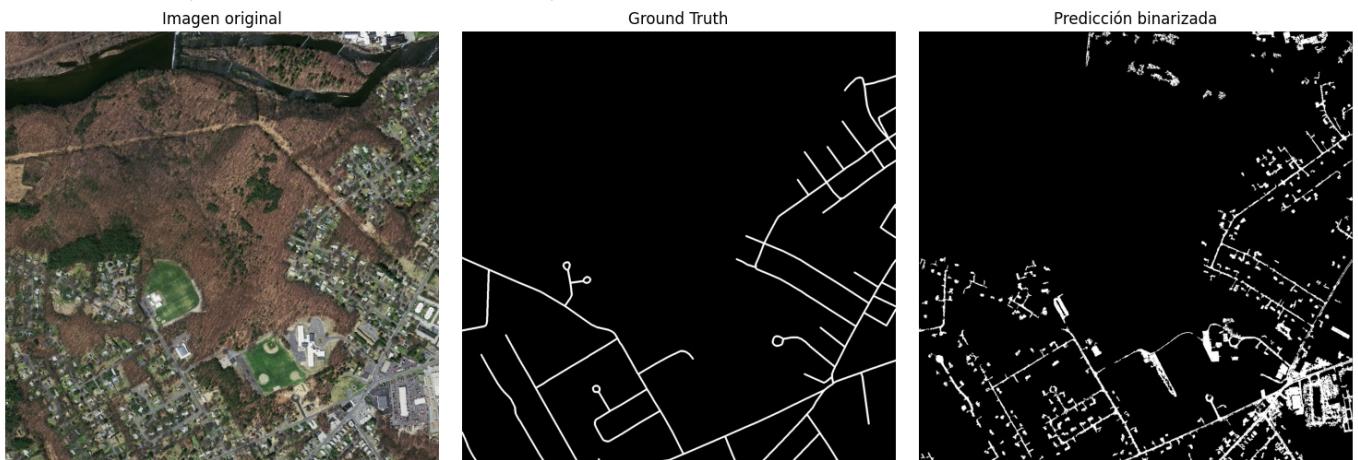
## Eliminamos componentes pequeñas
pred_0_filtrada = eliminar_componentes_pequeñas(pred_0, umbral=0.6, area_minima=70)

## Visualizamos la imagen original, la predicción y la gt
_ = visualizar_prediccion_gt_og(img_0, gt_0, pred_0_filtrada)

regiones_filtradas = componentes_conexas(pred_0_filtrada, umbral=0.6)
print(f"Cantidad de componentes conexas después de eliminar pequeñas: {len(regiones_filtradas)}")
print(f"Área mínima de los componentes: {min([region.area for region in regiones_filtradas])}")
print(f"Área máxima de los componentes: {max([region.area for region in regiones_filtradas])}")

```

Cantidad de componentes conexas: 2763
Área mínima de los componentes: 1.0
Área máxima de los componentes: 51403.0
Cantidad de componentes conexas menores a 70 píxeles: 2467



Cantidad de componentes conexas después de eliminar pequeñas: 296
Área mínima de los componentes: 71.0
Área máxima de los componentes: 51403.0

- **Conclusión:**

Se logran eliminar pequeños puntos de ruido (2467) que no se corresponden con carreteras. Se ha probado a eliminar componentes de tamaño menor a 50, 100 y 200 píxeles, finalmente el tamaño 50 parece reducir bien el ruido sin eliminar componentes de carretera.

Utilizamos **operaciones morfológicas** para intentar eliminar el posible ruido residual e intentar unir componentes que se encuentren separados.

In []:

```

## Probamos a aplicar la operación de cierre
from skimage.morphology import closing, footprint_rectangle, disk

for size in [3, 5, 7]:
    ## Aplicamos la operación de cierre

```

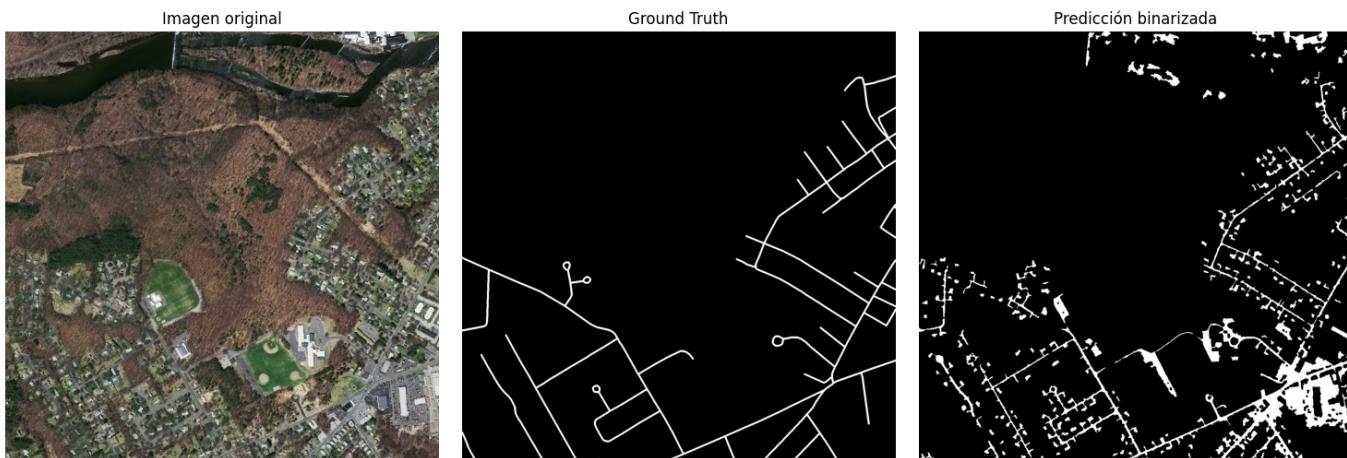
```

pred_0_disk = closing(pred_0_filtrada, disk(size))
pred_0_rectangle = closing(pred_0_filtrada, footprint_rectangle((size, size)))

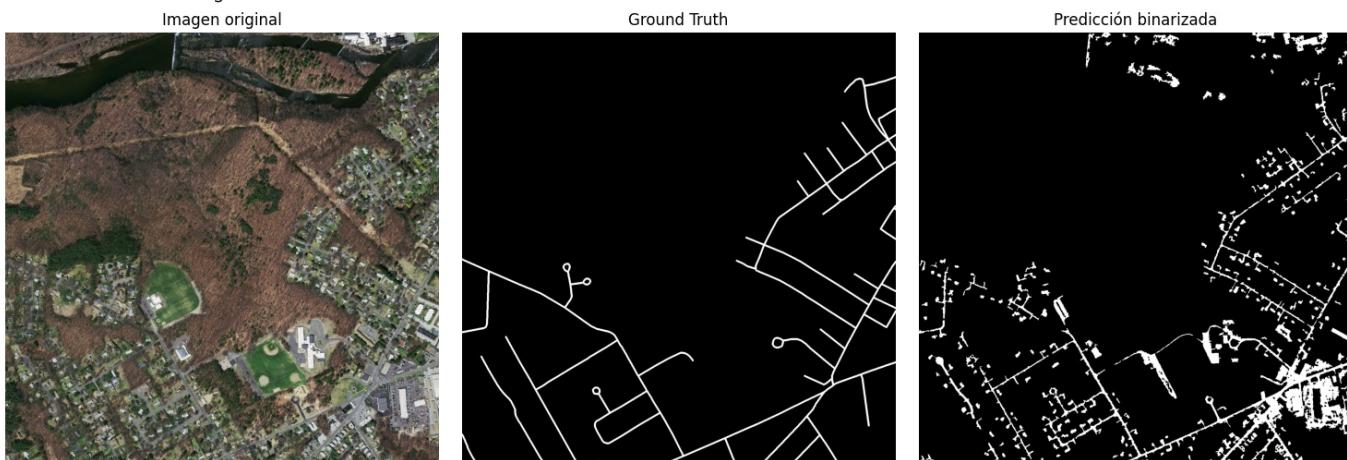
## Visualizamos la imagen original, la predicción y la gt
print(f"Cierre con disco de tamaño {size}")
_ = visualizar_prediccion_gt_og(img_0, gt_0, pred_0_disk, umbral=0.6)
print(f"Cierre con rectángulo de tamaño {size}")
_ = visualizar_prediccion_gt_og(img_0, gt_0, pred_0_rectangle, umbral=0.6)

```

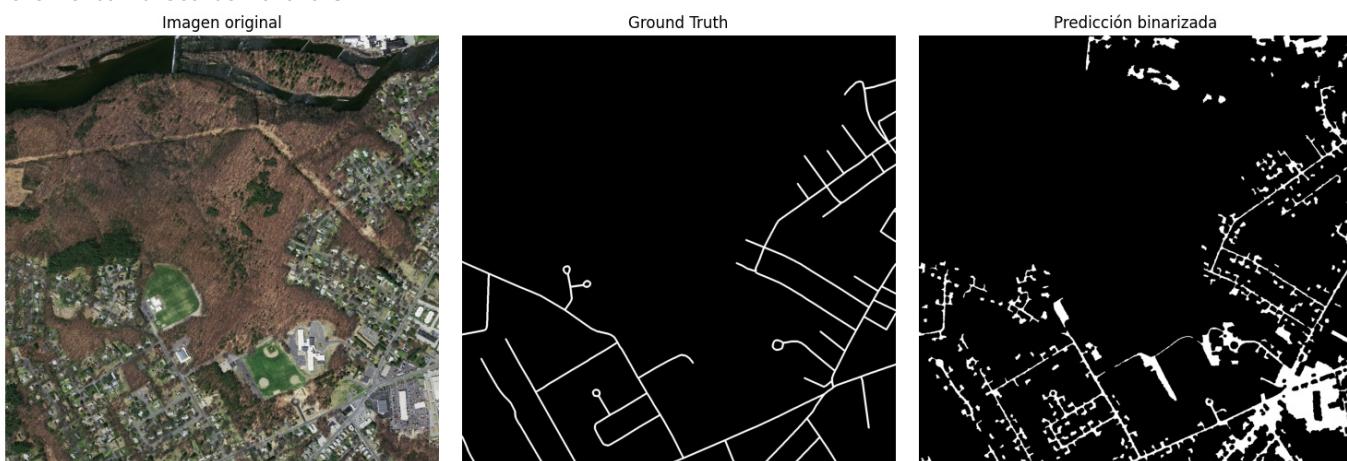
Cierre con disco de tamaño 3



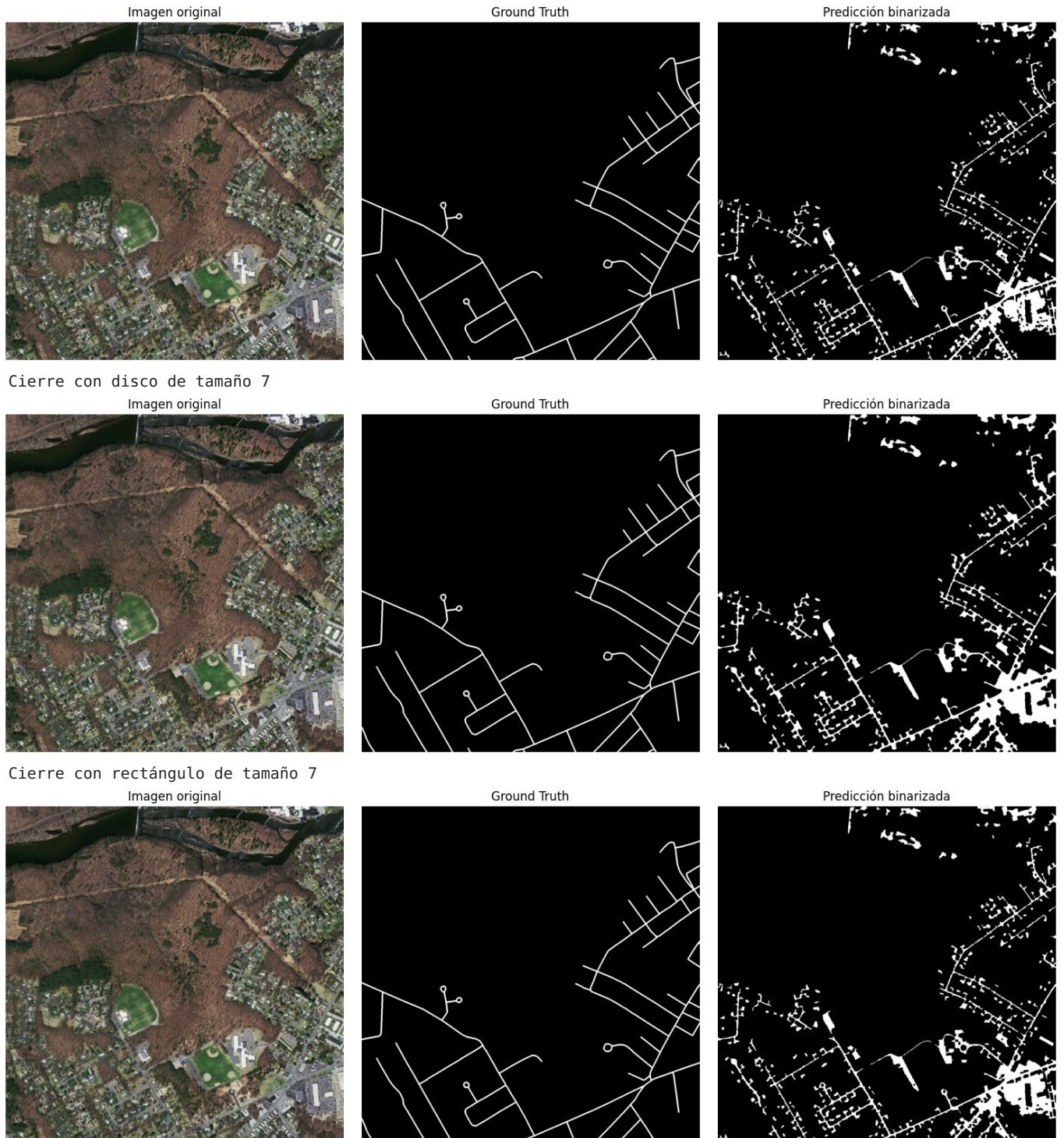
Cierre con rectángulo de tamaño 3



Cierre con disco de tamaño 5



Cierre con rectángulo de tamaño 5



- **Conclusión:**

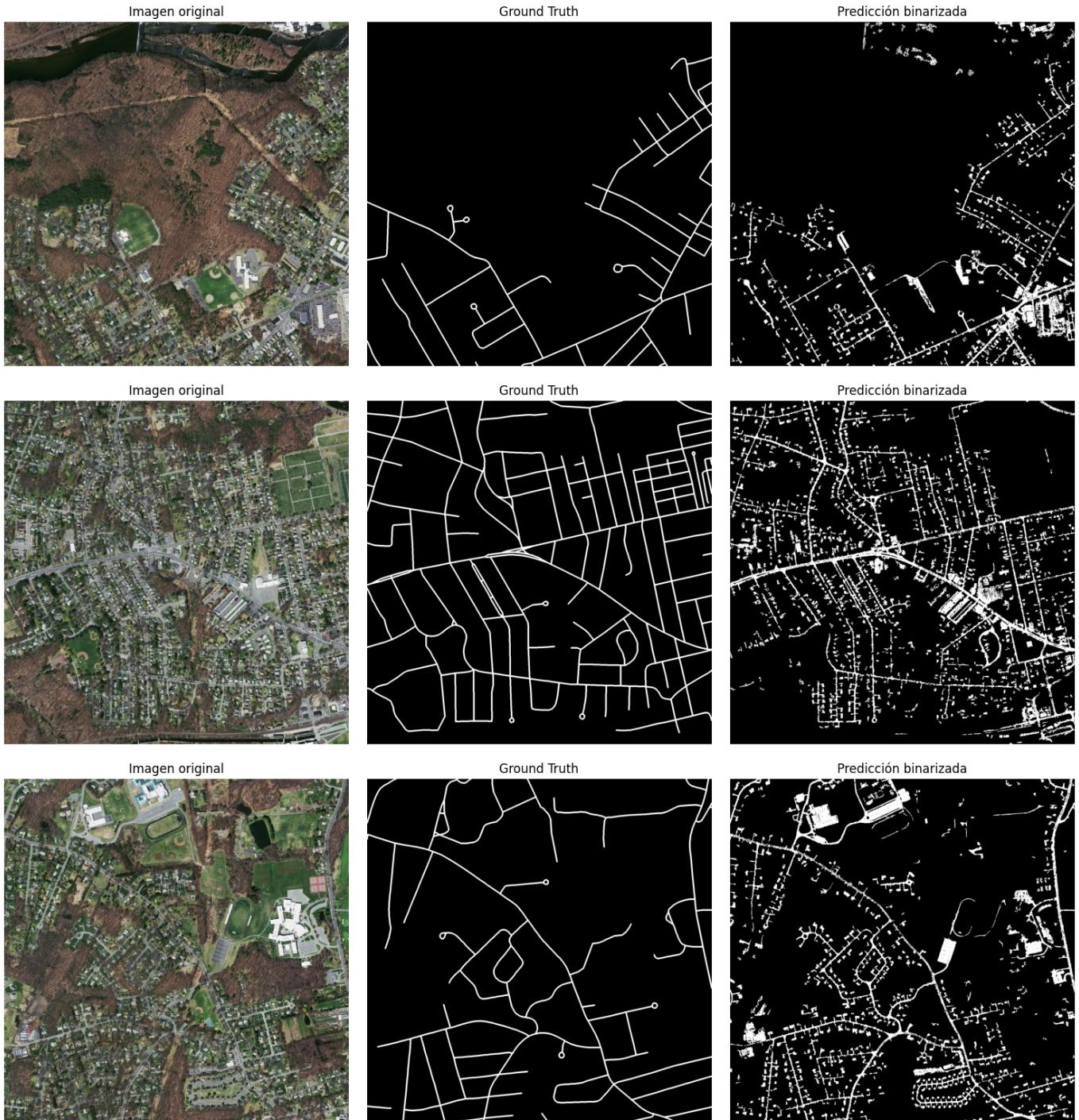
Los resultados no son muy satisfactorios. La operación de cierre si consigue unir componentes que se encuentran separados, pero a cambio, aquellas zonas mal predichas (que son más grandes que el tamaño del EE) tambien se unen.

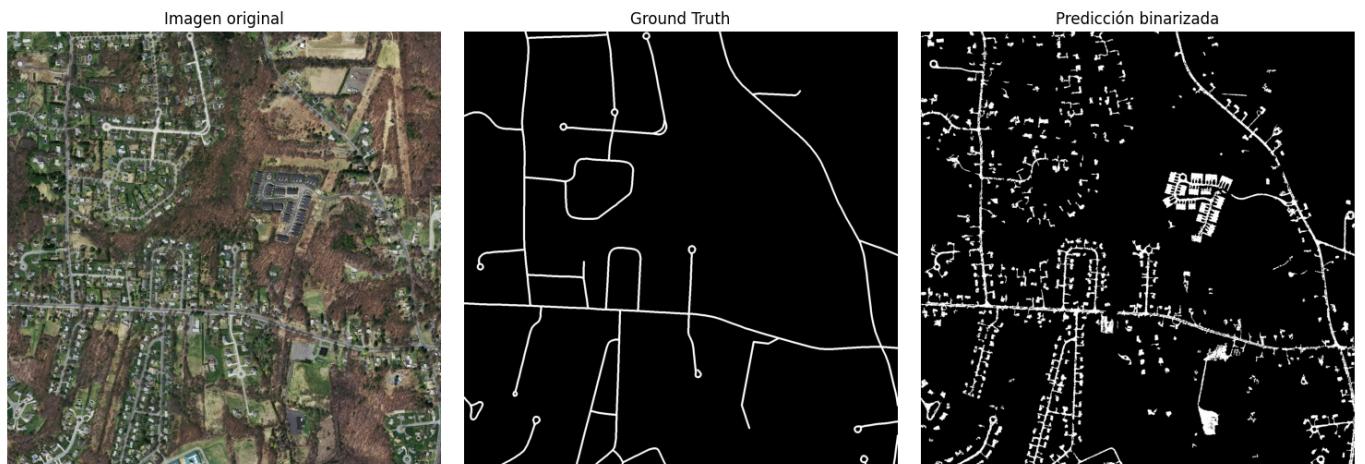
La ventaja de la unión de carreteras no consideramos que compense la desventaja de unir componentes que no son carreteras y que estas destaquean más.

Finalmente nuestro post-procesado se compone de:

- Umbral de 0.6 para la binarización de la imagen obtenida por el clasificador.
- Eliminación de componentes conexos de tamaño menor a 50 píxeles.

```
In [30]: ## Conjunto final de imágenes
imagenes_finales = []
for idx in range(len(x_test)):
    ## Aplicamos el posprocesado de componentes conexas
    prediccion_filtrada = eliminar_componentes_pequeñas(prediccion[idx], umbral=0.6, area_minima=50)
    ## Visualizamos la imagen original, la predicción y la gt
    imagenes_finales.append(visualizar_prediccion_gt_og(x_test[idx], y_test[idx], prediccion_filtrada))
```





Conclusiones finales

- **Valoración general:**

- Cuantitativamente, la segmentación obtenida por el clasificador es bastante buena, con un resultado de AUC de 0.89.
- La segmentación obtenida por el clasificador no es perfecta pero bastante. Se observan errores de segmentación en la imagen binaria obtenida. Estos errores son principalmente píxeles de ruido que se han clasificado como carretera y pequeñas discontinuidades pero tambien hay estructuras más grandes que detecta como carreteras.

- **Puntos débiles del modelo. Errores:**

- Nuestro modelo identifica gran cantidad de tejados como carreteras, no creamos un modelo capaz de diferenciar entre ambos de forma efectiva. Pasa lo mismo con naves industriales o grandes superficies.
- Existen discontinuidades en las carreteras predichas que no se consiguen eliminar en el post-procesado.
- Nuestra imagen binaria tiene puntos mínimos de ruido que no se logran eliminar con el etiquetado de componentes conexos.
- Hay varias carreteras que no se detectan, y que sí son carreteras en el *groud truth* y más importante, pueden ser identificadas como tal en la imagen de satélite.

- **Puntos fuertes del modelo:**

- Nuestras segmentaciones diferencian de forma muy efectiva carreteras de caminos forestales o cortafuegos.
- Se obtienen carreteras no determinadas por el *ground truth* pero que son efectivamente carreteras a la vista de la imagen de satélite.
- Todas las carreteras estrechas entre casas o por el contrario zonas extensas asfaltadas (como parkings) son identificadas como carretera por nuestro modelo.
- Nuestro modelo generaliza bien, la segmentación es buena en las imágenes que ha aprendido pero no sobreentrena a estas y es capaz de predecir las del conjunto de test.

Extra: Prueba de cambio en la función de pérdida

Tal y como comentamos anteriormente, los datos están desbalanceados, cada imagen tiene muchos más píxeles de fondo que de carretera. Esto puede llevar a que el modelo aprenda a clasificar todos los píxeles como fondo y no aprenda nada.

Nuestro modelo anterior no ha tenido este problema, pero para evitarlo, se ha probado a modificar la función de pérdida para que penalice más los errores en la clase minoritaria (carretera).

La función de pérdida utilizada es la siguiente:

```
In [ ]: def weighted_binary_crossentropy(y_true, y_pred):
    """
    BCE ponderado para manejar desbalanceo cuando la clase positiva es minoritaria.
    """
    epsilon = tf.keras.backend.epsilon()
    y_pred = tf.clip_by_value(y_pred, epsilon, 1. - epsilon)

    ## Calculamos pesos
    pos_weight = tf.reduce_sum(1. - y_true) / tf.reduce_sum(y_true)

    ## Aplicamos BCE manual con pesos
    loss = - (pos_weight * y_true * tf.math.log(y_pred) + (1 - y_true) * tf.math.log(1 - y_pred))
    return tf.reduce_mean(loss)
```

Creamos un modelo convolucional, igual que el anterior, pero que hace uso de esta función de pérdida.

```
In [ ]: def modelo1(input_shape):
    inputs = layers.Input(shape=input_shape)

    x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(inputs)
    x = layers.Conv2D(16, (3, 3), activation='relu', padding='same')(x)
    x = layers.Conv2D(1, (1, 1), activation='sigmoid', padding='same')(x)

    outputs = layers.Reshape((input_shape[0], input_shape[1]))(x)
    model = models.Model(inputs=inputs, outputs=outputs)

    model.compile(optimizer=optimizers.Adam(learning_rate=0.001),
                  loss=weighted_binary_crossentropy,
                  metrics=[
                      metrics.BinaryAccuracy(name='accuracy'),
                      metrics.Precision(name='precision'),
                      metrics.Recall(name='recall'),
                      metrics.AUC(name='auc')])

    return model
```

Entrenamos el modelo neuronal.

```
In [ ]: ## Creamos el modelo
INPUT_SHAPE = (1500, 1500, 10)
model = modelo1(input_shape=INPUT_SHAPE)

## Resumen del modelo
model.summary()

## Entrenamos el modelo
history = model.fit(x_train, y_train, epochs=5, batch_size=1)

## Guardamos el modelo
model.save("modelo2.keras")
```

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 1500, 1500, 10)	0
conv2d (Conv2D)	(None, 1500, 1500, 32)	2,912
conv2d_1 (Conv2D)	(None, 1500, 1500, 16)	4,624
conv2d_2 (Conv2D)	(None, 1500, 1500, 1)	17
reshape (Reshape)	(None, 1500, 1500)	0

Total params: 7,553 (29.50 KB)

Trainable params: 7,553 (29.50 KB)

Non-trainable params: 0 (0.00 B)

```

Epoch 1/5
16/16 26s 1s/step - accuracy: 0.6024 - auc: 0.7022 - loss: 1.2109 - precision: 0.0529 - rec
all: 0.6846
Epoch 2/5
16/16 20s 1s/step - accuracy: 0.8504 - auc: 0.8970 - loss: 0.7660 - precision: 0.1828 - rec
all: 0.8367
Epoch 3/5
16/16 26s 2s/step - accuracy: 0.8456 - auc: 0.8990 - loss: 0.7742 - precision: 0.1574 - rec
all: 0.8499
Epoch 4/5
16/16 25s 2s/step - accuracy: 0.8688 - auc: 0.9213 - loss: 0.6916 - precision: 0.1676 - rec
all: 0.8685
Epoch 5/5
16/16 22s 1s/step - accuracy: 0.8643 - auc: 0.9156 - loss: 0.6990 - precision: 0.1795 - rec
all: 0.8653

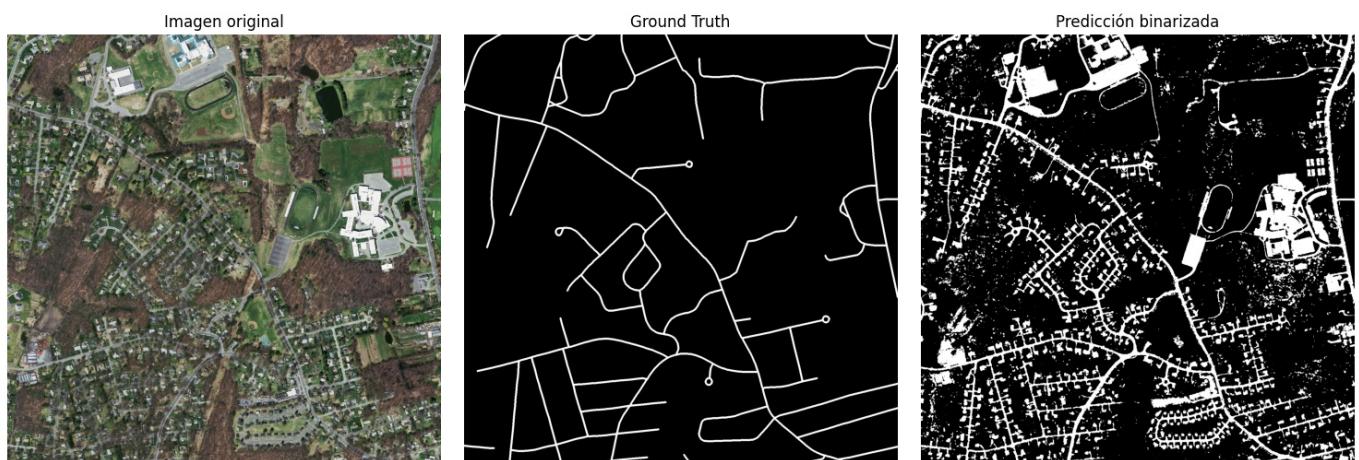
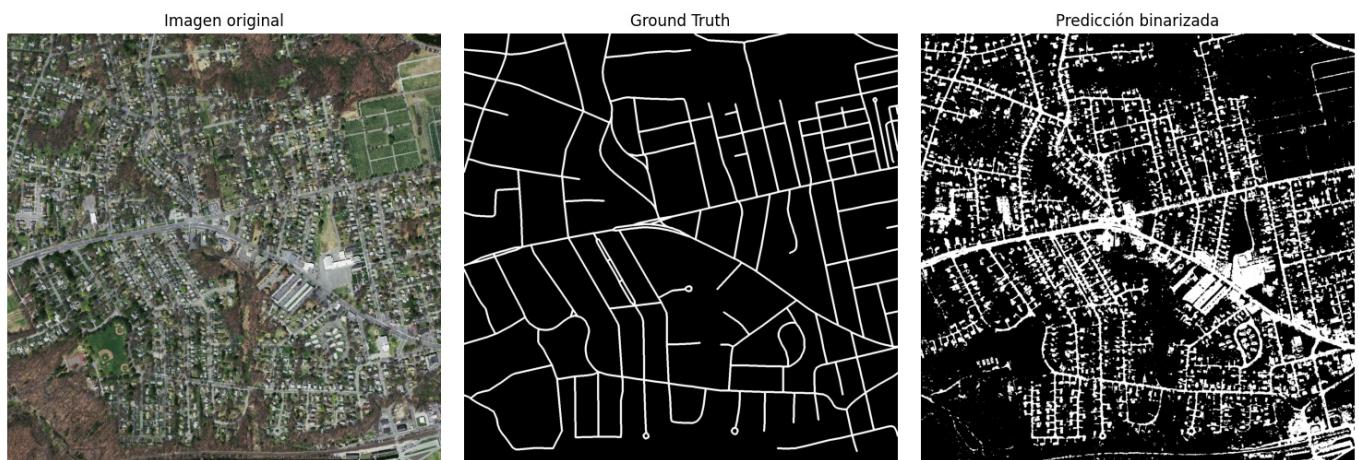
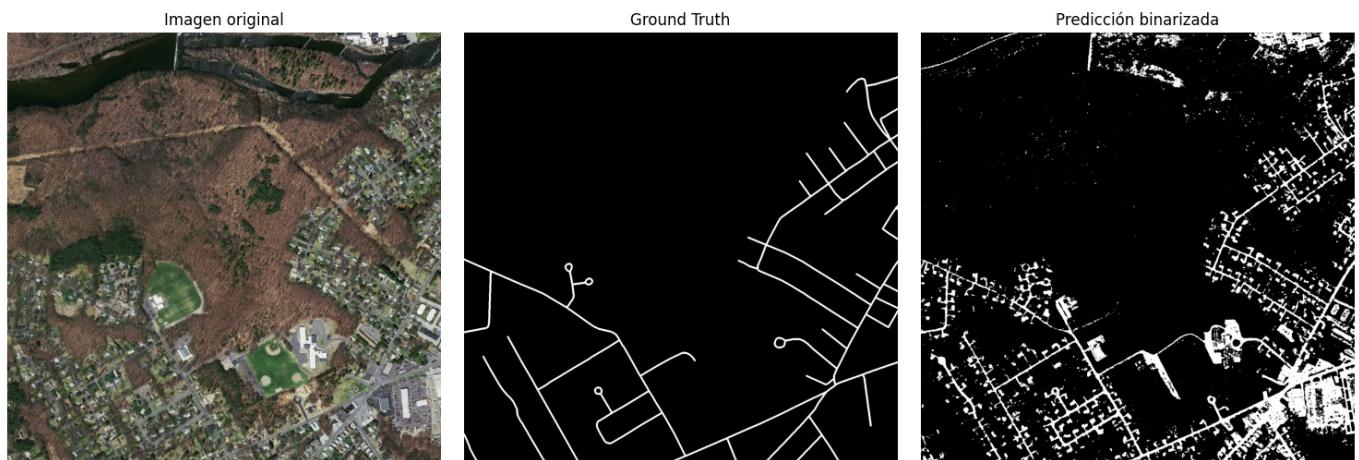
```

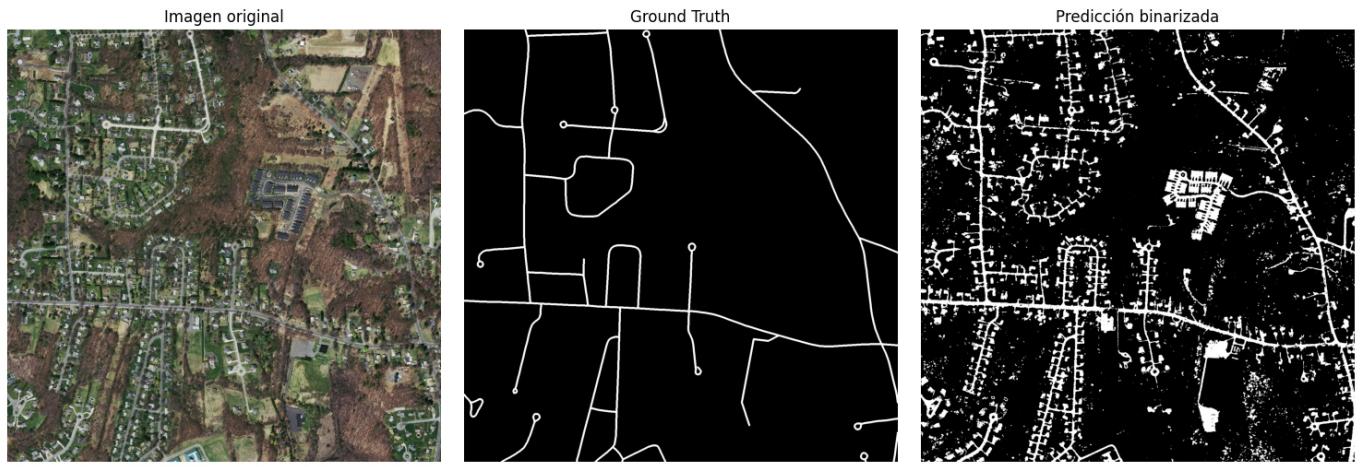
Las métricas parecen satisfactorias, pero probamos directamente a visualizar predicciones en las imágenes de test.

```
In [ ]: ## Predicción sobre el conjunto de test
prediccion = model.predict(x_test)
```

```
1/1 12s 12s/step
```

```
In [ ]: for idx in range(len(x_test)):
    ## Visualizamos la imagen original, la predicción y la gt
    visualizar_prediccion_gt_og(x_test[idx], y_test[idx], prediccion[idx])
```





- **Conclusión:**

La segmentación obtenida (visualizada con umbral 0.5) es similar a la obtenida con el modelo anterior. Aunque se ha dejado como definitivo el modelo sin emplear esta función por su más sencilla implementación e interpretación, la creación de esta función de pérdida es interesante y puede ser **realmente útil en estos problemas de desbalanceo**.

Experimentación extra

Con el modelo de segmentación completado surge la duda de si los resultados son gracias a la capacidad de la red convolucional o si las características extraídas de la imagen son las que permiten la segmentación de carreteras. Para ello, obtenemos como nuevo vector de características solamente los canales R, G y B.

```
In [33]: ## Entrenamos de nuevo un modelo que solo usará los canales R, G y B

x_train_rgb = x_train[:, :, :, :3]
x_test_rgb = x_test[:, :, :, :3]
y_train_rgb = y_train[:, :, :, 0]
y_test_rgb = y_test[:, :, :, 0]

## Creamos el modelo
model_rgb = modelo1(input_shape=(1500, 1500, 3))
model_rgb.summary()

## Entrenamos el modelo
history_rgb = model_rgb.fit(x_train_rgb, y_train_rgb, epochs=5, batch_size=1)

## Evaluación en el conjunto de test
score_rgb = model_rgb.evaluate(x_test_rgb, y_test_rgb)
print(f"Test loss: {score_rgb[0]}")
print(f"Test accuracy: {score_rgb[1]}")
print(f"Test AUC: {score_rgb[4]})
```

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 1500, 1500, 3)	0
conv2d (Conv2D)	(None, 1500, 1500, 32)	896
conv2d_1 (Conv2D)	(None, 1500, 1500, 16)	4,624
conv2d_2 (Conv2D)	(None, 1500, 1500, 1)	17
reshape (Reshape)	(None, 1500, 1500)	0

Total params: 5,537 (21.63 KB)

Trainable params: 5,537 (21.63 KB)

Non-trainable params: 0 (0.00 B)

```

Epoch 1/5
16/16 19s 708ms/step - accuracy: 0.9724 - auc: 0.3595 - loss: 0.4846 - precision: 0.0000e+0
0 - recall: 0.0000e+00
Epoch 2/5
16/16 10s 588ms/step - accuracy: 0.9642 - auc: 0.2781 - loss: 0.2130 - precision: 0.0000e+0
0 - recall: 0.0000e+00
Epoch 3/5

16/16 10s 592ms/step - accuracy: 0.9610 - auc: 0.2590 - loss: 0.2321 - precision: 0.0000e+0
0 - recall: 0.0000e+00
Epoch 4/5
16/16 11s 694ms/step - accuracy: 0.9666 - auc: 0.3027 - loss: 0.1859 - precision: 0.0000e+0
0 - recall: 0.0000e+00
Epoch 5/5
16/16 12s 722ms/step - accuracy: 0.9652 - auc: 0.3257 - loss: 0.1824 - precision: 0.0000e+0
0 - recall: 0.0000e+00
1/1 4s 4s/step - accuracy: 0.9502 - auc: 0.3660 - loss: 0.2249 - precision: 0.0000e+00 - recall: 0.0000e+00
Test loss: 0.22491109371185303
Test accuracy: 0.9502122402191162
Test AUC: 0.3659599721431732

```

- **Nota:**

Es posible que en este modelo con menos características tenga más influencia el desbalanceo de clases y sea necesario aplicar técnicas para corregir esto (cambio en la función de pérdida).

```
In [ ]: ## Creamos un modelo identico pero con funcion de perdida adaptada al desbalanceo de clases
def weighted_binary_crossentropy(y_true, y_pred):
    """
    BCE ponderado para manejar desbalanceo cuando la clase positiva es minoritaria.
    """
    epsilon = tf.keras.backend.epsilon()
    y_pred = tf.clip_by_value(y_pred, epsilon, 1. - epsilon)

    ## Calculamos pesos
    pos_weight = tf.reduce_sum(1. - y_true) / tf.reduce_sum(y_true)

    ## Aplicamos BCE manual con pesos
    loss = - (pos_weight * y_true * tf.math.log(y_pred) + (1 - y_true) * tf.math.log(1 - y_pred))
    return tf.reduce_mean(loss)

def modelo2(input_shape):
    inputs = layers.Input(shape=input_shape)

    x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(inputs)
    x = layers.Conv2D(16, (3, 3), activation='relu', padding='same')(x)
    x = layers.Conv2D(1, (1, 1), activation='sigmoid', padding='same')(x)

    outputs = layers.Reshape((input_shape[0], input_shape[1]))(x)
    model = models.Model(inputs, outputs)

    model.compile(optimizer=optimizers.Adam(learning_rate=0.001),
                  loss=weighted_binary_crossentropy,
                  metrics=[metrics.BinaryAccuracy(name='accuracy'),
                           metrics.Precision(name='precision'),
                           metrics.Recall(name='recall'),
                           metrics.AUC(name='auc')])

    return model
```

Entrenamos el modelo, con una función de pérdida que tiene en cuenta el desbalanceo de clases.

```
In [40]: ## Creamos el modelo
model_rgb2 = modelo2(input_shape=(1500, 1500, 3))
model_rgb2.summary()

## Entrenamos el modelo
history_rgb2 = model_rgb2.fit(x_train_rgb, y_train_rgb, epochs=5, batch_size=1)

## Evaluación en el conjunto de test
score_rgb2 = model_rgb2.evaluate(x_test_rgb, y_test_rgb)
print(f"Test loss: {score_rgb2[0]}")
print(f"Test accuracy: {score_rgb2[1]}")
print(f"Test AUC: {score_rgb2[4]}")
```

Model: "functional_2"

Layer (type)	Output Shape	Param #
input_layer_2 (InputLayer)	(None, 1500, 1500, 3)	0
conv2d_6 (Conv2D)	(None, 1500, 1500, 32)	896
conv2d_7 (Conv2D)	(None, 1500, 1500, 16)	4,624
conv2d_8 (Conv2D)	(None, 1500, 1500, 1)	17
reshape_2 (Reshape)	(None, 1500, 1500)	0

Total params: 5,537 (21.63 KB)

Trainable params: 5,537 (21.63 KB)

Non-trainable params: 0 (0.00 B)

Epoch 1/5

16/16 14s 622ms/step - accuracy: 0.0411 - auc: 0.7343 - loss: 1.3147 - precision: 0.0350 - recall: 0.9983

Epoch 2/5

16/16 9s 579ms/step - accuracy: 0.2448 - auc: 0.8312 - loss: 1.2452 - precision: 0.0442 - recall: 0.9668

Epoch 3/5

16/16 10s 626ms/step - accuracy: 0.7192 - auc: 0.8456 - loss: 1.1370 - precision: 0.0887 - recall: 0.8540

Epoch 4/5

16/16 9s 583ms/step - accuracy: 0.8301 - auc: 0.8725 - loss: 0.9779 - precision: 0.1487 - recall: 0.8207

Epoch 5/5

16/16 10s 605ms/step - accuracy: 0.8635 - auc: 0.8846 - loss: 0.9067 - precision: 0.1698 - recall: 0.7821

1/1 3s 3s/step - accuracy: 0.8450 - auc: 0.8865 - loss: 0.8556 - precision: 0.2141 - recall: 0.7912

Test loss: 0.8555772304534912

Test accuracy: 0.8450157642364502

Test AUC: 0.8864730596542358

```
In [41]: ## Predicción sobre el conjunto de test
prediccion_rgb = model_rgb2.predict(x_test_rgb)
for idx in range(len(x_test_rgb)):
    ## Visualizamos la imagen original, la predicción y la gt
    visualizar_prediccion_gt_og(x_test_rgb[idx], y_test_rgb[idx], prediccion_rgb[idx])
```

1/1 3s 3s/step

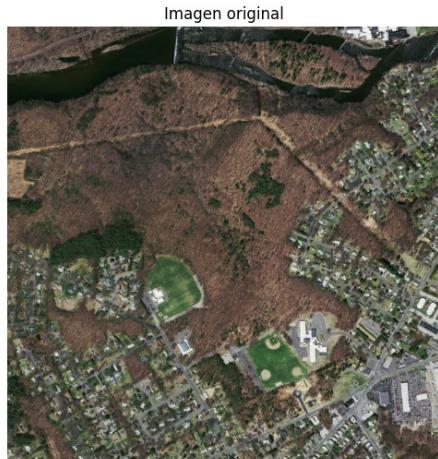
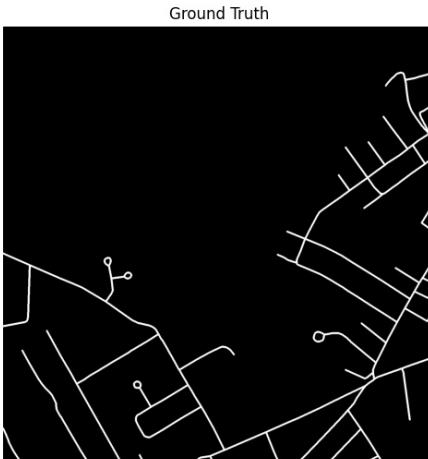


Imagen original

Ground Truth



Predicción binarizada

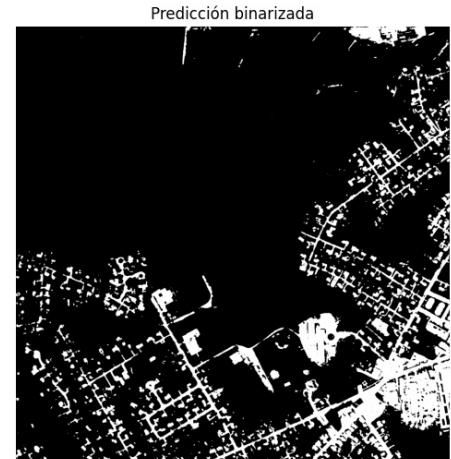


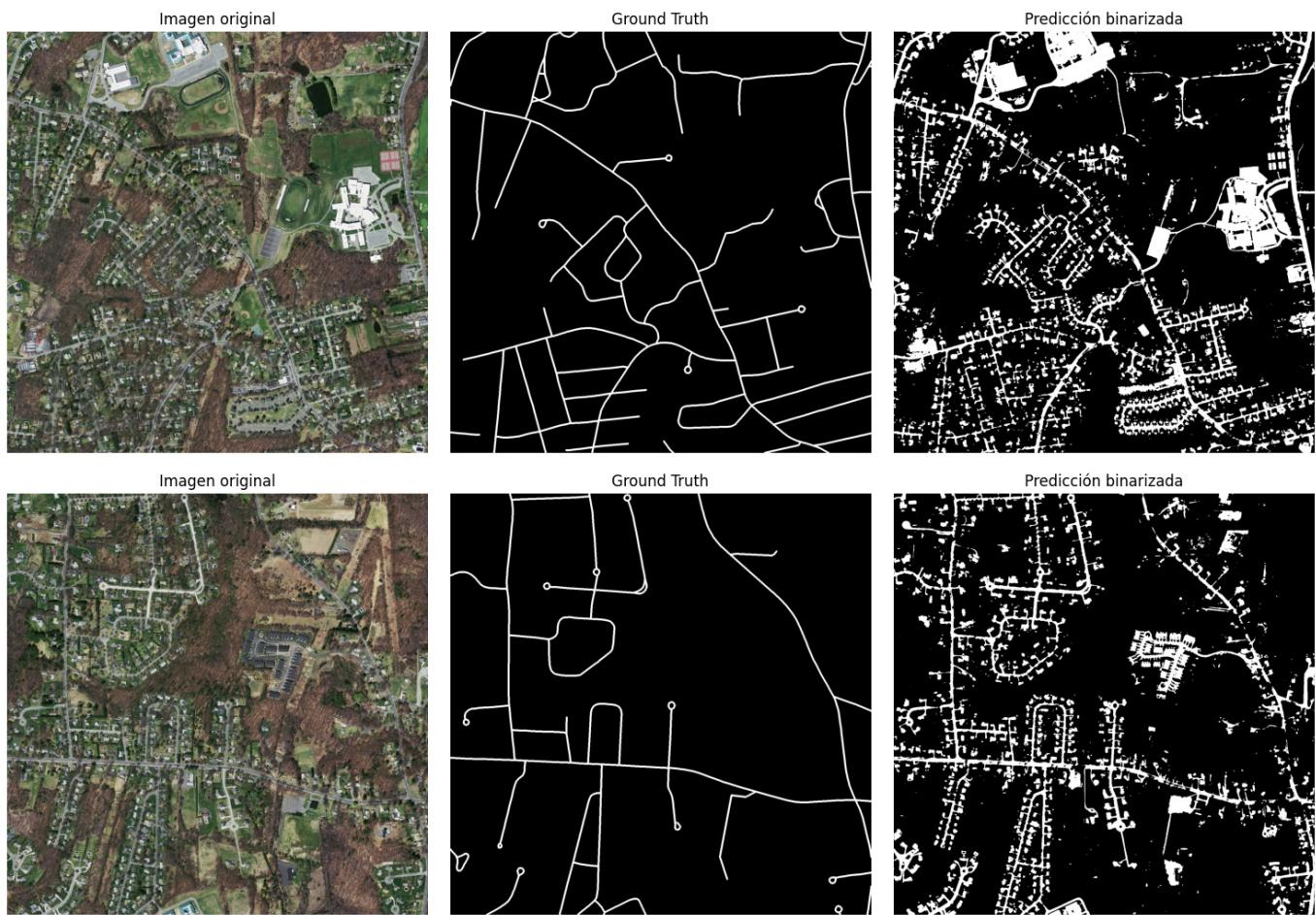
Imagen original

Ground Truth



Predicción binarizada





- **Conclusión:**

En cuanto a las métricas obtenidas, si se logra un AUC de 0.89, lo que indica que el modelo tiene un buen rendimiento. Sin embargo, la segmentación obtenida no es tan buena como la del modelo anterior en la evaluación visual.

Al disponer solo de información de color, el modelo tiene mayor dificultad para diferenciar estructuras similares a carreteras (tejados, naves industriales) y no logra segmentar correctamente las carreteras. La cantidad de falsos positivos es mucho mayor.

De todas formas, cabe resaltar la capacidad de la red convolucional, que pese a disponer solo de información de color y tener solo 2 capas, es capaz de identificar carreteras.

Ejercicio 2: Reconocimiento de objetos

Desarrollar un método computacional que permita:

- **A partir de una imagen (y/o, opcionalmente, su máscara de recorte), identificar si pertenece a la categoría “elephant”, o a la categoría “rhino”**
- **Proporcionar una evaluación cuantitativa de los objetivos del sistema, usando un conjunto de datos de prueba y una metodología de validación adecuada.**

Esquema del trabajo

El objetivo es construir un modelo de aprendizaje automático capaz de clasificar imágenes de elefantes y rinocerontes. En este caso, el proceso ya no es píxel a píxel, sino que se trata de un problema de clasificación de imágenes completas.

1. **Cargar las imágenes** de los animales y sus máscaras.
2. Creación de un **vector de etiquetas** con la clase de cada imagen (elefante rinoceronte o otro).
 - Se asigna la etiqueta 0 a las imágenes de elefantes y la etiqueta 1 a las imágenes de rinocerontes.
3. Entrenamiento de diferentes **modelos de clasificación**.

Utilizaremos **diferentes planteamientos**:

- Recortar las imágenes con la máscara y entrenar con las imágenes recortadas aplastadas como entrada.
- Extraer características generales de las imágenes (área animal, perímetro, etc.) y entrenar con estas características.

- Trabajar con características HOG de las imágenes en nuestros clasificadores.

Para ambos casos se probarán los siguientes **clasificadores**:

- Modelo de redes de neuronas
- Modelo de SVM
- Modelo de Random Forest

Se dividirá en **entrenamiento** y **test** cada conjunto de características de imágenes.

4. Modelo final.

- Buscamos crear el clasificador definitivo, aquel que ofrezca mejores resultados en el conjunto de test. Para ello, se probarán diferentes arquitecturas, parámetros o técnicas.

1.1. Cargar imágenes

Importamos las librerías necesarias para este ejercicio.

```
In [2]: import matplotlib.pyplot as plt
import pandas as pd
from tensorflow.keras import layers, models, optimizers, losses, metrics
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
import seaborn as sns
from skimage.feature import hog
```

Cargamos las imágenes de entrada y las convertimos a tipo de dato float.

```
In [ ]: ## Leemos las imágenes de los animales
elephant = skimage.io.imread_collection("./Materiales/objects/images/elephant/*.jpg")
emu = skimage.io.imread_collection("./Materiales/objects/images/emu/*.jpg")
flamingo = skimage.io.imread_collection("./Materiales/objects/images/flamingo/*.jpg")
rhino = skimage.io.imread_collection("./Materiales/objects/images/rhino/*.jpg")

## Imprimimos el tamaño de las imágenes
print(f"Número de imágenes de elephant: {len(elephant)}")
print(f"Número de imágenes de emu: {len(emu)}")
print(f"Número de imágenes de flamingo: {len(flamingo)}")
print(f"Número de imágenes de rhino: {len(rhino)}")

## Convertimos cada imagen a float
elephant = [skimage.img_as_float(image) for image in elephant]
emu = [skimage.img_as_float(image) for image in emu]
flamingo = [skimage.img_as_float(image) for image in flamingo]
rhino = [skimage.img_as_float(image) for image in rhino]

## Comprobamos si todas las imágenes tienen el mismo tamaño
print("Comprobando tamaños de las imágenes:")
for i, image in enumerate(elephant):
    print(f"Elephant {i}: {image.shape}")
for i, image in enumerate(emu):
    print(f"Emu {i}: {image.shape}")
for i, image in enumerate(flamingo):
    print(f"Flamingo {i}: {image.shape}")
for i, image in enumerate(rhino):
    print(f"Rhino {i}: {image.shape}")
```

Número de imágenes de elephant: 64

Número de imágenes de emu: 53

Número de imágenes de flamingo: 67

Número de imágenes de rhino: 59

Comprobando tamaños de las imágenes:

Elephant 0: (225, 300, 3)

Elephant 1: (225, 300, 3)

Elephant 2: (264, 300, 3)

Elephant 3: (243, 300, 3)

Elephant 4: (188, 300, 3)

Elephant 5: (212, 300, 3)

Elephant 6: (300, 239, 3)

Elephant 7: (225, 300, 3)

Elephant 8: (291, 300, 3)

Elephant 9: (195, 300, 3)

Elephant 10: (152, 300, 3)

Elephant 11: (300, 253, 3)

Elephant 12: (234, 300, 3)

Elephant 13: (263, 300, 3)

Elephant 14: (254, 300, 3)

Elephant 15: (225, 300, 3)

Elephant 16: (252, 300, 3)
Elephant 17: (300, 207, 3)
Elephant 18: (300, 298, 3)
Elephant 19: (241, 300, 3)
Elephant 20: (202, 300, 3)
Elephant 21: (209, 300, 3)
Elephant 22: (248, 300, 3)
Elephant 23: (216, 300, 3)
Elephant 24: (300, 287, 3)
Elephant 25: (300, 198, 3)
Elephant 26: (256, 300, 3)
Elephant 27: (199, 300, 3)
Elephant 28: (192, 300, 3)
Elephant 29: (256, 300, 3)
Elephant 30: (225, 300, 3)
Elephant 31: (260, 300, 3)
Elephant 32: (195, 300, 3)
Elephant 33: (225, 300, 3)
Elephant 34: (300, 261, 3)
Elephant 35: (300, 266, 3)
Elephant 36: (300, 196, 3)
Elephant 37: (221, 300, 3)
Elephant 38: (213, 300, 3)
Elephant 39: (193, 300, 3)
Elephant 40: (247, 300, 3)
Elephant 41: (267, 300, 3)
Elephant 42: (226, 300, 3)
Elephant 43: (246, 300, 3)
Elephant 44: (203, 300, 3)
Elephant 45: (273, 300, 3)
Elephant 46: (258, 300, 3)
Elephant 47: (263, 300, 3)
Elephant 48: (216, 300, 3)
Elephant 49: (212, 300, 3)
Elephant 50: (212, 300, 3)
Elephant 51: (227, 300, 3)
Elephant 52: (282, 300, 3)
Elephant 53: (287, 300, 3)
Elephant 54: (254, 300, 3)
Elephant 55: (234, 300, 3)
Elephant 56: (300, 257, 3)
Elephant 57: (200, 300, 3)
Elephant 58: (225, 300, 3)
Elephant 59: (234, 300)
Elephant 60: (247, 300, 3)
Elephant 61: (226, 300, 3)
Elephant 62: (240, 300, 3)
Elephant 63: (300, 240)
Emu 0: (248, 300, 3)
Emu 1: (220, 300, 3)
Emu 2: (300, 272, 3)
Emu 3: (288, 300, 3)
Emu 4: (300, 300, 3)
Emu 5: (300, 189, 3)
Emu 6: (240, 300, 3)
Emu 7: (225, 300, 3)
Emu 8: (253, 300, 3)
Emu 9: (260, 300, 3)
Emu 10: (300, 257, 3)
Emu 11: (300, 264, 3)
Emu 12: (300, 291, 3)
Emu 13: (225, 300, 3)
Emu 14: (239, 300, 3)
Emu 15: (216, 300, 3)
Emu 16: (252, 300, 3)
Emu 17: (300, 192, 3)
Emu 18: (300, 238, 3)
Emu 19: (253, 300, 3)
Emu 20: (217, 300, 3)
Emu 21: (283, 300, 3)
Emu 22: (300, 221, 3)
Emu 23: (257, 300, 3)
Emu 24: (282, 300, 3)
Emu 25: (300, 286, 3)
Emu 26: (225, 300, 3)
Emu 27: (195, 300, 3)
Emu 28: (274, 300, 3)
Emu 29: (300, 195, 3)
Emu 30: (254, 300, 3)
Emu 31: (220, 300, 3)
Emu 32: (300, 166, 3)
Emu 33: (273, 300)
Emu 34: (224, 300, 3)

Emu 35: (200, 300, 3)
Emu 36: (300, 285, 3)
Emu 37: (300, 292, 3)
Emu 38: (182, 300, 3)
Emu 39: (234, 300, 3)
Emu 40: (271, 300, 3)
Emu 41: (216, 300, 3)
Emu 42: (300, 289, 3)
Emu 43: (300, 290, 3)
Emu 44: (279, 300, 3)
Emu 45: (300, 264, 3)
Emu 46: (300, 241, 3)
Emu 47: (300, 221, 3)
Emu 48: (264, 300, 3)
Emu 49: (260, 300, 3)
Emu 50: (237, 300, 3)
Emu 51: (254, 300, 3)
Emu 52: (223, 300, 3)
Flamingo 0: (300, 250, 3)
Flamingo 1: (300, 80, 3)
Flamingo 2: (300, 225, 3)
Flamingo 3: (225, 300, 3)
Flamingo 4: (300, 300, 3)
Flamingo 5: (300, 277, 3)
Flamingo 6: (300, 212, 3)
Flamingo 7: (300, 181, 3)
Flamingo 8: (300, 182, 3)
Flamingo 9: (219, 300, 3)
Flamingo 10: (300, 295, 3)
Flamingo 11: (300, 272, 3)
Flamingo 12: (300, 170, 3)
Flamingo 13: (300, 256, 3)
Flamingo 14: (300, 211, 3)
Flamingo 15: (300, 194, 3)
Flamingo 16: (300, 231, 3)
Flamingo 17: (300, 178, 3)
Flamingo 18: (300, 186, 3)
Flamingo 19: (300, 246, 3)
Flamingo 20: (300, 217, 3)
Flamingo 21: (194, 300, 3)
Flamingo 22: (300, 179, 3)
Flamingo 23: (300, 257, 3)
Flamingo 24: (300, 182, 3)
Flamingo 25: (300, 272, 3)
Flamingo 26: (300, 298, 3)
Flamingo 27: (251, 300, 3)
Flamingo 28: (300, 198, 3)
Flamingo 29: (300, 277, 3)
Flamingo 30: (300, 300, 3)
Flamingo 31: (300, 256, 3)
Flamingo 32: (300, 225, 3)
Flamingo 33: (300, 218, 3)
Flamingo 34: (300, 243, 3)
Flamingo 35: (300, 201, 3)
Flamingo 36: (300, 200, 3)
Flamingo 37: (300, 236, 3)
Flamingo 38: (300, 298, 3)
Flamingo 39: (230, 300, 3)
Flamingo 40: (300, 195, 3)
Flamingo 41: (300, 208, 3)
Flamingo 42: (250, 300, 3)
Flamingo 43: (300, 219, 3)
Flamingo 44: (300, 297, 3)
Flamingo 45: (300, 243, 3)
Flamingo 46: (297, 300, 3)
Flamingo 47: (218, 300, 3)
Flamingo 48: (300, 239, 3)
Flamingo 49: (300, 284, 3)
Flamingo 50: (300, 207, 3)
Flamingo 51: (300, 202, 3)
Flamingo 52: (300, 200, 3)
Flamingo 53: (201, 300, 3)
Flamingo 54: (300, 229, 3)
Flamingo 55: (300, 219, 3)
Flamingo 56: (300, 281, 3)
Flamingo 57: (300, 188, 3)
Flamingo 58: (248, 300, 3)
Flamingo 59: (300, 170, 3)
Flamingo 60: (300, 148, 3)
Flamingo 61: (289, 300, 3)
Flamingo 62: (300, 155, 3)
Flamingo 63: (284, 300, 3)
Flamingo 64: (300, 234, 3)

Flamingo 65: (300, 198, 3)
Flamingo 66: (300, 261, 3)
Rhino 0: (255, 300, 3)
Rhino 1: (259, 300, 3)
Rhino 2: (200, 300, 3)
Rhino 3: (197, 300, 3)
Rhino 4: (187, 300, 3)
Rhino 5: (204, 300, 3)
Rhino 6: (205, 300, 3)
Rhino 7: (167, 300, 3)
Rhino 8: (155, 300, 3)
Rhino 9: (208, 300, 3)
Rhino 10: (210, 300, 3)
Rhino 11: (203, 300, 3)
Rhino 12: (159, 300, 3)
Rhino 13: (205, 300, 3)
Rhino 14: (176, 300, 3)
Rhino 15: (225, 300, 3)
Rhino 16: (225, 300, 3)
Rhino 17: (201, 300, 3)
Rhino 18: (200, 300, 3)
Rhino 19: (247, 300, 3)
Rhino 20: (193, 300, 3)
Rhino 21: (201, 300, 3)
Rhino 22: (215, 300, 3)
Rhino 23: (200, 300, 3)
Rhino 24: (219, 300, 3)
Rhino 25: (209, 300, 3)
Rhino 26: (182, 300)
Rhino 27: (261, 300, 3)
Rhino 28: (187, 300, 3)
Rhino 29: (210, 300, 3)
Rhino 30: (198, 300, 3)
Rhino 31: (226, 300, 3)
Rhino 32: (209, 300, 3)
Rhino 33: (246, 300, 3)
Rhino 34: (276, 300, 3)
Rhino 35: (224, 300, 3)
Rhino 36: (200, 300, 3)
Rhino 37: (259, 300, 3)
Rhino 38: (200, 300, 3)
Rhino 39: (219, 300, 3)
Rhino 40: (192, 300, 3)
Rhino 41: (203, 300, 3)
Rhino 42: (288, 300, 3)
Rhino 43: (220, 300, 3)
Rhino 44: (213, 300, 3)
Rhino 45: (188, 300, 3)
Rhino 46: (186, 300, 3)
Rhino 47: (175, 300, 3)
Rhino 48: (225, 300, 3)
Rhino 49: (225, 300, 3)
Rhino 50: (196, 300, 3)
Rhino 51: (180, 300, 3)
Rhino 52: (184, 300, 3)
Rhino 53: (256, 300, 3)
Rhino 54: (197, 300, 3)
Rhino 55: (204, 300, 3)
Rhino 56: (194, 300, 3)
Rhino 57: (225, 300, 3)
Rhino 58: (202, 300, 3)

- **Aclaración:**

Mantenemos las imágenes de cada animal como conjuntos separados, de forma que posteriormente nos sea sencillo crear las etiquetas de cada imagen a partir de su nombre.

- **Conclusión:**

- Las imágenes de animales tienen diferentes dimensiones incluso dentro del mismo conjunto.
- La mayoría son imágenes a color, pero hay algunas en escala de gris.

Eso debe tenerse en cuenta para evitar problemas posteriores.

Vamos a crear una función que haga todo el proceso de **carga de imágenes y máscaras**. Esta función recibe como entrada la ruta de las imágenes y devuelve un array con las imágenes y otro con las máscaras. Esta función se encargará de:

1. Cargar las imágenes

2. Convertirlas a tipo float
3. Redimensionarlas a un tamaño fijo (250x250 píxeles)
4. Convertirlas a 3 canales en caso de que sea en escala de gris.

```
In [4]: def cargar_imagenes_y_mascaras(ruta_imagenes, ruta_mascaras):
    """
    Carga imágenes y máscaras desde las rutas especificadas.

    Args:
        ruta_imagenes (str): Ruta a la carpeta que contiene las imágenes.
        ruta_mascaras (str): Ruta a la carpeta que contiene las máscaras.

    Returns:
        list: Lista de imágenes cargadas.
        list: Lista de máscaras cargadas.
    """
    ## Cargamos las imágenes y las máscaras
    imagenes = skimage.io.imread_collection(ruta_imagenes)
    mascaras = skimage.io.imread_collection(ruta_mascaras)

    ## Convertimos cada imagen a float
    imagenes = [skimage.img_as_float(image) for image in imagenes]
    mascaras = [skimage.img_as_float(image) for image in mascaras]

    ## Convertimos a color si es necesario
    for i in range(len(imagenes)):
        if len(imagenes[i].shape) == 2:
            imagenes[i] = skimage.color.gray2rgb(imagenes[i])

    ## Cambiamos el tamaño a 250x250
    for i in range(len(imagenes)):
        imagenes[i] = skimage.transform.resize(imagenes[i], (250, 250), anti_aliasing=True)
        mascaras[i] = skimage.transform.resize(mascaras[i], (250, 250), anti_aliasing=True)

    ## Convertimos las máscaras a 0 y 1
    mascaras = [np.where(mask > 0.5, 1, 0) for mask in mascaras]

    ## Convertimos a numpy array
    imagenes = np.array(imagenes)
    mascaras = np.array(mascaras)

    return imagenes, mascaras
```

Utilizamos la función definida anteriormente sobre las rutas de los animales (imagen y máscara).

```
In [5]: elephant, elephant_masks = cargar_imagenes_y_mascaras("./Materiales/objects/images/elephant/*.jpg",
                                                               "./Materiales/objects/masks/elephant/*.png")
emu, emu_masks = cargar_imagenes_y_mascaras("./Materiales/objects/images/emu/*.jpg",
                                              "./Materiales/objects/masks/emu/*.png")
flamingo, flamingo_masks = cargar_imagenes_y_mascaras("./Materiales/objects/images/flamingo/*.jpg",
                                                       "./Materiales/objects/masks/flamingo/*.png")
rhino, rhino_masks = cargar_imagenes_y_mascaras("./Materiales/objects/images/rhino/*.jpg",
                                                 "./Materiales/objects/masks/rhino/*.png")
```

Imprimimos información de utilidad sobre las imágenes cargadas.

```
In [6]: print("\nInformación de los conjuntos:")
print("-----")
print(f"Número de imágenes de elephant: {len(elephant)}")
print(f"Número de imágenes de emu: {len(emu)}")
print(f"Número de imágenes de flamingo: {len(flamingo)}")
print(f"Número de imágenes de rhino: {len(rhino)}")
print(f"Tamaño de las imágenes: {elephant[0].shape}")
print(f"Tamaño de las máscaras: {elephant_masks[0].shape}")
print(f"\nInformación de una imagen concreta:")
print("-----")
print(f"Tipo de dato imagen: {elephant[0].dtype} con valores entre {elephant[0].min()} y {elephant[0].max()}")
print(f"Tipo de dato máscara: {elephant_masks[0].dtype} con valores entre {elephant_masks[0].min()} y {elephant_masks[0].max()}")
print(f"Valores únicos de la máscara: {np.unique(elephant_masks[0])}")
```

Información de los conjuntos:

Número de imágenes de elephant: 64
Número de imágenes de emu: 53
Número de imágenes de flamingo: 67
Número de imágenes de rhino: 59
Tamaño de las imágenes: (250, 250, 3)
Tamaño de las máscaras: (250, 250)

Información de una imagen concreta:

Tipo de dato imagen: float64 con valores entre 0.030686274509804053 y 1.0
Tipo de dato máscara: int64 con valores entre 0 y 1
Valores únicos de la máscara: [0 1]

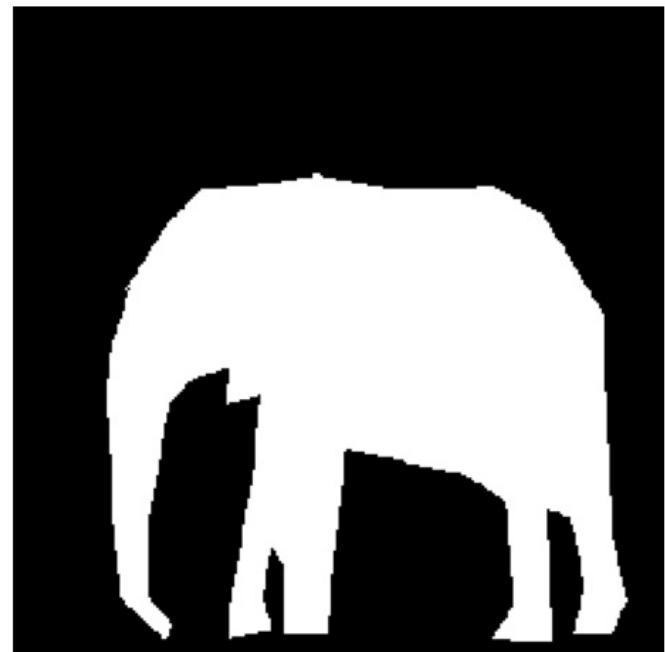
Visualizamos la primera imagen de cada animal y su correspondiente máscara.

```
In [8]: fig, ax = plt.subplots(4, 2, figsize=(10, 20))
for i, (animal, mask) in enumerate(zip([elephant, emu, flamingo, rhino],
                                         [elephant_masks, emu_masks, flamingo_masks, rhino_masks])):
    ax[i, 0].imshow(animal[0])
    ax[i, 0].set_title(f"{{['Elefante', 'Emu', 'Flamingo', 'Rinoceronte']{[i]}}}")
    ax[i, 0].axis("off")
    ax[i, 1].imshow(mask[0], cmap="gray")
    ax[i, 1].set_title(f"{'Máscara {{['Elefante', 'Emu', 'Flamingo', 'Rinoceronte']{[i]}}}'")
    ax[i, 1].axis("off")
plt.tight_layout()
plt.show()
```

Elefante



Máscara Elefante



Emu



Máscara Emu



Flamingo

Máscara Flamingo



Rinoceronte



Máscara Rinoceronte



1.2. Creación del vector de etiquetas

Creamos el **vector de etiquetas** para nuestro modelo de clasificación supervisado. Las imágenes de elefantes tendrán etiqueta 0 y las de rinocerontes etiqueta 1. Los otros animales tendrán etiqueta 2.

En caso de que durante entrenamiento, algún modelo utilizado requiera un vector de etiquetas en formato *one-hot*, se aplicará la transformación posteriormente.

Obtenemos cada etiqueta a partir del conjunto en el que se encuentran las imágenes.

```
In [7]: ## La etiqueta de cada imagen se corresponde con el array en el que se encuentra
etiquetas = []
etiquetas.extend([0] * len(elephant_masks))
etiquetas.extend([2] * len(emu_masks))
etiquetas.extend([2] * len(flamingo_masks))
etiquetas.extend([1] * len(rhino_masks))

## Convertimos a numpy array
etiquetas = np.array(etiquetas)
print(f"Dimensiones de las etiquetas: {etiquetas.shape}")
print(f"Valores únicos de las etiquetas: {np.unique(etiquetas)}")
print(f"Cantidad de etiquetas 0: {len(np.where(etiquetas == 0)[0])}")
print(f"Cantidad de etiquetas 1: {len(np.where(etiquetas == 1)[0])}")
print(f"Cantidad de etiquetas 2: {len(np.where(etiquetas == 2)[0])}")
```

```
Dimensiones de las etiquetas: (243,)  
Valores únicos de las etiquetas: [0 1 2]  
Cantidad de etiquetas 0: 64  
Cantidad de etiquetas 1: 59  
Cantidad de etiquetas 2: 120
```

- **Conclusión:**

Se construye un solo *numpy array* con las etiquetas de cada imagen. Tendrá dimensión (número de imágenes) y los valores serán 0, 1 o 2 dependiendo del animal que se trate.

- 0 para elefantes
- 1 para rinocerontes
- 2 para otros animales

- **Aclaración:**

Será necesario mantener la posición de las etiquetas con respecto a las imágenes. A la hora de dividir el conjunto de entrenamiento y test, se mantendrá la correspondencia entre las imágenes y sus etiquetas

Unificamos en un solo array las imágenes de los cuatro conjuntos (elefantes, emus, rinocerontes y flamencos) y unificamos también las etiquetas. De esta forma, tenemos un solo conjunto de imágenes y un solo vector de etiquetas.

```
In [8]: imagenes = np.concatenate((elephant, emu, flamingo, rhino), axis=0)  
mascaras = np.concatenate((elephant_masks, emu_masks, flamingo_masks, rhino_masks), axis=0)  
print(f"Dimensiones de las imágenes: {imagenes.shape}")  
print(f"Dimensiones de las máscaras: {mascaras.shape}")
```

```
Dimensiones de las imágenes: (243, 250, 250, 3)  
Dimensiones de las máscaras: (243, 250, 250)
```

- **Conclusiones:**

Ahora tenemos un solo conjunto de imágenes y un solo conjunto de máscaras. Como podemos comprobar en las dimensiones, ahora todas las imágenes y máscaras tienen el mismo tamaño.

Debemos tener en cuenta a la hora de dividir en entrenamiento y test que esto se debe hacer **barajando de forma aleatoria** ya que en este array están ordenadas las imágenes de cada conjunto.

1.3. Experimentación de clasificadores

En los siguientes bloques de código se experimentará con **diferentes modelos de clasificación**.

Concretamente se probarán **3 modelos clasificadores** (modelo de redes de neuronas, SVM y Random Forest) y se probarán **diferentes características de entrada** para cada uno de ellos:

1. Imagen aplanada (3 canales) recortada con la máscara como vector de entrada.
2. Características generales de la figura animal (área, perímetro, etc.) como vector de entrada.
3. Información de bordes (magnitud y orientación de gradiente) de la imagen como vector de entrada.

1. Imagen aplanada como entrada

El primer modelo que vamos a probar es un **modelo simple** que utiliza solamente como entrada la imagen aplanada, convertida a un vector.

En este entrenamiento, vamos a tratar de emplear todos los beneficios de contar con las máscaras de recorte, cuyo uso era opcional como entrada adicional o alternativa.

Definimos una función para **recortar la figura del animal** a partir de la máscara. Esta función recibe como entrada la imagen y la máscara y devuelve la imagen recortada. De esta forma ignoraremos todos los píxeles del fondo y nos quedaremos con la información de forma y color del animal.

```
In [9]: def recortar_imagen(imagen, mascara):  
    """  
    Recorta la imagen utilizando la máscara.  
  
    Args:  
        imagen (ndarray): Imagen de entrada.  
        mascara (ndarray): Máscara de entrada (solo valores 0 y 1).  
  
    Returns:  
        ndarray: Imagen recortada.  
    """  
    ## Convertimos la máscara a tipo booleano
```

```

mascara = mascara.astype(bool)

## Recortamos la imagen utilizando la máscara
imagen_recortada = np.zeros_like(imagen)
imagen_recortada[mascara] = imagen[mascara]

return imagen_recortada

```

- **Aclaración:**

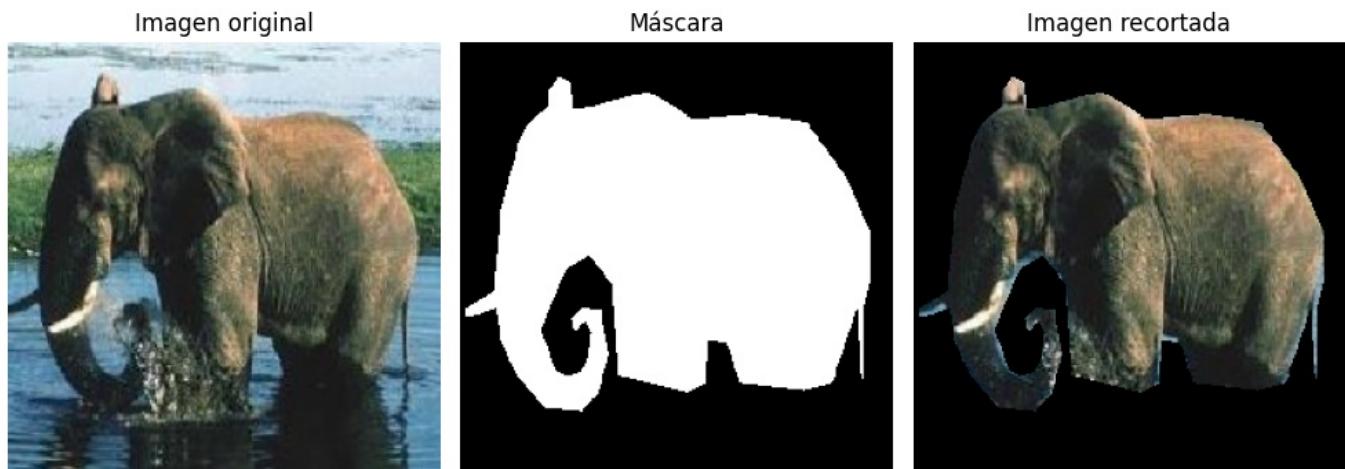
Esta función definida en esta primera experimentación será reutilizada siempre que se considere en otros planteamientos.

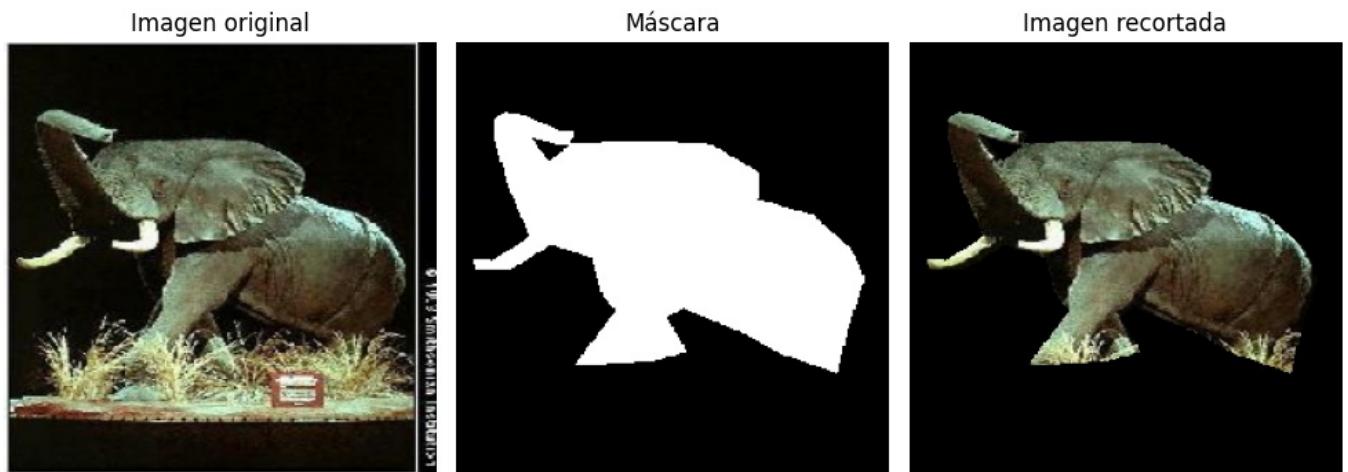
Aplicamos la función definida anteriormente a todas las imágenes de entrada y obtenemos las imágenes recortadas.

```
In [12]: imagenes_recortadas = []
for i in range(len(imagenes)):
    imagen_recortada = recortar_imagen(imagenes[i], mascaras[i])
    imagenes_recortadas.append(imagen_recortada)
```

Para tres imágenes aleatorias visualizamos la imagen original, la máscara y la imagen recortada.

```
In [13]: for i in range(3):
    idx = np.random.randint(0, len(imagenes))
    fig, ax = plt.subplots(1, 3, figsize=(10, 5))
    ax[0].imshow(imagenes[idx])
    ax[0].set_title("Imagen original")
    ax[0].axis("off")
    ax[1].imshow(mascaras[idx], cmap="gray")
    ax[1].set_title("Máscara")
    ax[1].axis("off")
    ax[2].imshow(imagenes_recortadas[idx])
    ax[2].set_title("Imagen recortada")
    ax[2].axis("off")
    plt.tight_layout()
    plt.show()
```





- **Conclusión:**

La función de recorte funciona correctamente, obteniendo la imagen del animal y eliminando el fondo. La imagen resultante tiene un tamaño de 250x250.

Para este primer modelo, la entrada será un vector con la información de RGB de la imagen recortada. Para que no sea computacionalmente muy costoso se **redimensionarán las imágenes a 64x64** píxeles. A continuación se hace un aplanado para obtener el vector de entrada.

```
In [ ]: ## Redimensionamos a 64x64
imagenes_aplanadas = [skimage.transform.resize(imagen, (64, 64), anti_aliasing=True) for imagen in imagenes_recortadas]
imagenes_aplanadas = np.array(imagenes_aplanadas)

## Aplanamos a un vector para cada imagen
imagenes_aplanadas = imagenes_aplanadas.reshape(imagenes_aplanadas.shape[0], -1)
print(f"Dimensiones de las imágenes recortadas: {imagenes_aplanadas.shape}")

Dimensiones de las imágenes recortadas: (243, 12288)
```

- **Conclusión:** Pasamos de las dimensiones de 250x250x3 a 64x64x3 con la función `transform.resize`, bajando la resolución de la imagen. Esto nos permitirá reducir el coste computacional del entrenamiento y la predicción. A continuación el aplanado nos dará un vector de 12288 elementos (64x64x3) para cada imagen.

Dividimos nuestro conjunto de imágenes (recortadas y aplanadas) en un conjunto de entrenamiento y otro de test. En este caso, se ha decidido dividir el 90% de las imágenes para el entrenamiento y el 10% restante para la evaluación del modelo.

```
In [ ]: ## Dividimos en entrenamiento y test de forma aleatoria
x_train, x_test, y_train, y_test = train_test_split(imagenes_aplanadas, etiquetas, test_size=0.1, random_state=42)
print(f"Dimensiones de x_train: {x_train.shape}")
print(f"Dimensiones de x_test: {x_test.shape}")
print(f"Dimensiones de y_train: {y_train.shape}")
print(f"Dimensiones de y_test: {y_test.shape}")

Dimensiones de x_train: (218, 12288)
Dimensiones de x_test: (25, 12288)
Dimensiones de y_train: (218,)
Dimensiones de y_test: (25,)
```

1. MLP

Creamos un modelo simple de red neuronal. La función de activación es softmax con 3 neuronas (una por cada clase) y la función de pérdida es `categorical_crossentropy`. No es necesario aplicar `one-hot` a las etiquetas.

```
In [10]: def modelo_simple(input_shape):
    inputs = layers.Input(shape=input_shape)

    x = layers.Dense(64, activation='relu')(inputs)
    x = layers.Dense(32, activation='relu')(x)
    x = layers.Dense(3, activation='softmax')(x)

    model = models.Model(inputs=inputs, outputs=x)
```

```

        model.compile(optimizer=optimizers.Adam(learning_rate=0.001),
                      loss=losses.SparseCategoricalCrossentropy(),
                      metrics=[
                        metrics.SparseCategoricalAccuracy(name='accuracy')
                      ])
    return model

```

- **Aclaración:**

Recurriremos a esta función posteriormente para la creación de modelos con otras características como entrada.

Entrenamos nuestro modelo con el conjunto anteriormente definido.

```
In [72]: ## Creamos el modelo
model_simple = modelo_simple(input_shape=(64*64*3,))
model_simple.summary()

## Entrenamos el modelo
history_simple = model_simple.fit(x_train, y_train, epochs=100, batch_size=32, validation_split=0.2)
```

Model: "functional_5"

Layer (type)	Output Shape	Param #
input_layer_5 (InputLayer)	(None, 12288)	0
dense_15 (Dense)	(None, 64)	786,496
dense_16 (Dense)	(None, 32)	2,080
dense_17 (Dense)	(None, 3)	99

Total params: 788,675 (3.01 MB)

Trainable params: 788,675 (3.01 MB)

Non-trainable params: 0 (0.00 B)

Epoch 1/100

6/6 2s 39ms/step - accuracy: 0.3687 - loss: 1.1333 - val_accuracy: 0.4773 - val_loss: 1.376

3

Epoch 2/100

6/6 0s 16ms/step - accuracy: 0.6561 - loss: 0.7796 - val_accuracy: 0.5455 - val_loss: 0.995

9

Epoch 3/100

6/6 0s 18ms/step - accuracy: 0.7601 - loss: 0.5333 - val_accuracy: 0.5227 - val_loss: 1.124

7

Epoch 4/100

6/6 0s 17ms/step - accuracy: 0.8860 - loss: 0.3885 - val_accuracy: 0.5682 - val_loss: 1.138

7

Epoch 5/100

6/6 0s 16ms/step - accuracy: 0.9223 - loss: 0.2745 - val_accuracy: 0.5682 - val_loss: 1.307

2

Epoch 6/100

6/6 0s 16ms/step - accuracy: 0.8844 - loss: 0.2762 - val_accuracy: 0.6136 - val_loss: 1.200

1

Epoch 7/100

6/6 0s 15ms/step - accuracy: 0.9353 - loss: 0.2216 - val_accuracy: 0.6136 - val_loss: 1.407

8

Epoch 8/100

6/6 0s 15ms/step - accuracy: 0.9414 - loss: 0.1787 - val_accuracy: 0.5909 - val_loss: 1.221

8

Epoch 9/100

6/6 0s 15ms/step - accuracy: 0.9684 - loss: 0.1275 - val_accuracy: 0.5909 - val_loss: 1.521

5

Epoch 10/100

6/6 0s 15ms/step - accuracy: 0.9850 - loss: 0.0876 - val_accuracy: 0.6136 - val_loss: 1.358

0

Epoch 11/100

6/6 0s 17ms/step - accuracy: 0.9735 - loss: 0.0917 - val_accuracy: 0.6364 - val_loss: 1.458

1

Epoch 12/100

6/6 0s 15ms/step - accuracy: 0.9814 - loss: 0.0792 - val_accuracy: 0.6364 - val_loss: 1.645

9

Epoch 13/100

6/6 0s 17ms/step - accuracy: 0.9850 - loss: 0.0564 - val_accuracy: 0.6364 - val_loss: 1.531

9

Epoch 14/100

6/6 0s 16ms/step - accuracy: 0.9897 - loss: 0.0508 - val_accuracy: 0.6364 - val_loss: 1.550

1

Epoch 15/100
6/6 0s 15ms/step - accuracy: 1.0000 - loss: 0.0358 - val_accuracy: 0.6364 - val_loss: 1.597
9
Epoch 16/100
6/6 0s 15ms/step - accuracy: 1.0000 - loss: 0.0366 - val_accuracy: 0.6364 - val_loss: 1.691
7
Epoch 17/100
6/6 0s 14ms/step - accuracy: 1.0000 - loss: 0.0359 - val_accuracy: 0.6364 - val_loss: 1.711
4
Epoch 18/100
6/6 0s 14ms/step - accuracy: 1.0000 - loss: 0.0275 - val_accuracy: 0.6136 - val_loss: 1.761
5
Epoch 19/100
6/6 0s 14ms/step - accuracy: 1.0000 - loss: 0.0257 - val_accuracy: 0.6591 - val_loss: 1.726
7
Epoch 20/100
6/6 0s 15ms/step - accuracy: 1.0000 - loss: 0.0240 - val_accuracy: 0.6591 - val_loss: 1.702
1
Epoch 21/100
6/6 0s 17ms/step - accuracy: 1.0000 - loss: 0.0172 - val_accuracy: 0.6591 - val_loss: 1.761
5
Epoch 22/100
6/6 0s 19ms/step - accuracy: 1.0000 - loss: 0.0150 - val_accuracy: 0.6591 - val_loss: 1.840
2
Epoch 23/100
6/6 0s 16ms/step - accuracy: 1.0000 - loss: 0.0153 - val_accuracy: 0.6591 - val_loss: 1.895
9
Epoch 24/100
6/6 0s 15ms/step - accuracy: 1.0000 - loss: 0.0122 - val_accuracy: 0.6591 - val_loss: 1.887
7
Epoch 25/100
6/6 0s 16ms/step - accuracy: 1.0000 - loss: 0.0111 - val_accuracy: 0.6591 - val_loss: 1.880
1
Epoch 26/100
6/6 0s 16ms/step - accuracy: 1.0000 - loss: 0.0079 - val_accuracy: 0.6591 - val_loss: 1.845
1
Epoch 27/100
6/6 0s 14ms/step - accuracy: 1.0000 - loss: 0.0100 - val_accuracy: 0.6591 - val_loss: 1.929
4
Epoch 28/100
6/6 0s 16ms/step - accuracy: 1.0000 - loss: 0.0082 - val_accuracy: 0.6591 - val_loss: 1.976
0
Epoch 29/100
6/6 0s 16ms/step - accuracy: 1.0000 - loss: 0.0072 - val_accuracy: 0.6591 - val_loss: 1.961
8
Epoch 30/100
6/6 0s 18ms/step - accuracy: 1.0000 - loss: 0.0100 - val_accuracy: 0.6591 - val_loss: 1.951
0
Epoch 31/100
6/6 0s 20ms/step - accuracy: 1.0000 - loss: 0.0082 - val_accuracy: 0.6591 - val_loss: 1.991
2
Epoch 32/100
6/6 0s 16ms/step - accuracy: 1.0000 - loss: 0.0063 - val_accuracy: 0.6591 - val_loss: 2.008
9
Epoch 33/100
6/6 0s 15ms/step - accuracy: 1.0000 - loss: 0.0051 - val_accuracy: 0.6591 - val_loss: 2.034
5
Epoch 34/100
6/6 0s 16ms/step - accuracy: 1.0000 - loss: 0.0051 - val_accuracy: 0.6591 - val_loss: 2.065
6
Epoch 35/100
6/6 0s 17ms/step - accuracy: 1.0000 - loss: 0.0079 - val_accuracy: 0.6591 - val_loss: 2.047
8
Epoch 36/100
6/6 0s 17ms/step - accuracy: 1.0000 - loss: 0.0040 - val_accuracy: 0.6591 - val_loss: 2.050
8
Epoch 37/100
6/6 0s 17ms/step - accuracy: 1.0000 - loss: 0.0043 - val_accuracy: 0.6591 - val_loss: 2.050
0
Epoch 38/100
6/6 0s 15ms/step - accuracy: 1.0000 - loss: 0.0049 - val_accuracy: 0.6591 - val_loss: 2.074
5
Epoch 39/100
6/6 0s 18ms/step - accuracy: 1.0000 - loss: 0.0038 - val_accuracy: 0.6591 - val_loss: 2.075
4
Epoch 40/100
6/6 0s 16ms/step - accuracy: 1.0000 - loss: 0.0037 - val_accuracy: 0.6591 - val_loss: 2.086
7
Epoch 41/100
6/6 0s 14ms/step - accuracy: 1.0000 - loss: 0.0036 - val_accuracy: 0.6591 - val_loss: 2.100
8
Epoch 42/100
6/6 0s 14ms/step - accuracy: 1.0000 - loss: 0.0049 - val_accuracy: 0.6591 - val_loss: 2.106

8
Epoch 43/100
6/6 0s 17ms/step - accuracy: 1.0000 - loss: 0.0036 - val_accuracy: 0.6591 - val_loss: 2.109
2
Epoch 44/100
6/6 0s 16ms/step - accuracy: 1.0000 - loss: 0.0036 - val_accuracy: 0.6591 - val_loss: 2.125
4
Epoch 45/100
6/6 0s 16ms/step - accuracy: 1.0000 - loss: 0.0028 - val_accuracy: 0.6591 - val_loss: 2.149
4
Epoch 46/100
6/6 0s 18ms/step - accuracy: 1.0000 - loss: 0.0027 - val_accuracy: 0.6591 - val_loss: 2.164
3
Epoch 47/100
6/6 0s 15ms/step - accuracy: 1.0000 - loss: 0.0028 - val_accuracy: 0.6591 - val_loss: 2.176
0
Epoch 48/100
6/6 0s 17ms/step - accuracy: 1.0000 - loss: 0.0022 - val_accuracy: 0.6591 - val_loss: 2.161
2
Epoch 49/100
6/6 0s 17ms/step - accuracy: 1.0000 - loss: 0.0024 - val_accuracy: 0.6591 - val_loss: 2.177
7
Epoch 50/100
6/6 0s 16ms/step - accuracy: 1.0000 - loss: 0.0019 - val_accuracy: 0.6591 - val_loss: 2.186
6
Epoch 51/100
6/6 0s 16ms/step - accuracy: 1.0000 - loss: 0.0023 - val_accuracy: 0.6591 - val_loss: 2.188
6
Epoch 52/100
6/6 0s 16ms/step - accuracy: 1.0000 - loss: 0.0029 - val_accuracy: 0.6591 - val_loss: 2.204
0
Epoch 53/100
6/6 0s 16ms/step - accuracy: 1.0000 - loss: 0.0023 - val_accuracy: 0.6591 - val_loss: 2.214
8
Epoch 54/100
6/6 0s 14ms/step - accuracy: 1.0000 - loss: 0.0020 - val_accuracy: 0.6591 - val_loss: 2.222
5
Epoch 55/100
6/6 0s 14ms/step - accuracy: 1.0000 - loss: 0.0026 - val_accuracy: 0.6591 - val_loss: 2.240
4
Epoch 56/100
6/6 0s 24ms/step - accuracy: 1.0000 - loss: 0.0019 - val_accuracy: 0.6591 - val_loss: 2.244
7
Epoch 57/100
6/6 0s 16ms/step - accuracy: 1.0000 - loss: 0.0017 - val_accuracy: 0.6591 - val_loss: 2.256
1
Epoch 58/100
6/6 0s 16ms/step - accuracy: 1.0000 - loss: 0.0024 - val_accuracy: 0.6591 - val_loss: 2.258
3
Epoch 59/100
6/6 0s 16ms/step - accuracy: 1.0000 - loss: 0.0017 - val_accuracy: 0.6591 - val_loss: 2.275
2
Epoch 60/100
6/6 0s 18ms/step - accuracy: 1.0000 - loss: 0.0018 - val_accuracy: 0.6591 - val_loss: 2.292
9
Epoch 61/100
6/6 0s 17ms/step - accuracy: 1.0000 - loss: 0.0022 - val_accuracy: 0.6591 - val_loss: 2.276
3
Epoch 62/100
6/6 0s 15ms/step - accuracy: 1.0000 - loss: 0.0013 - val_accuracy: 0.6591 - val_loss: 2.278
9
Epoch 63/100
6/6 0s 14ms/step - accuracy: 1.0000 - loss: 0.0013 - val_accuracy: 0.6591 - val_loss: 2.277
2
Epoch 64/100
6/6 0s 16ms/step - accuracy: 1.0000 - loss: 0.0017 - val_accuracy: 0.6591 - val_loss: 2.294
3
Epoch 65/100
6/6 0s 15ms/step - accuracy: 1.0000 - loss: 0.0015 - val_accuracy: 0.6591 - val_loss: 2.308
8
Epoch 66/100
6/6 0s 14ms/step - accuracy: 1.0000 - loss: 0.0019 - val_accuracy: 0.6591 - val_loss: 2.329
4
Epoch 67/100
6/6 0s 15ms/step - accuracy: 1.0000 - loss: 0.0013 - val_accuracy: 0.6591 - val_loss: 2.329
6
Epoch 68/100
6/6 0s 14ms/step - accuracy: 1.0000 - loss: 0.0013 - val_accuracy: 0.6591 - val_loss: 2.341
7
Epoch 69/100
6/6 0s 14ms/step - accuracy: 1.0000 - loss: 0.0010 - val_accuracy: 0.6591 - val_loss: 2.345
8
Epoch 70/100

6/6 ━━━━━━ 0s 14ms/step - accuracy: 1.0000 - loss: 0.0013 - val_accuracy: 0.6591 - val_loss: 2.343
5
Epoch 71/100
6/6 ━━━━━━ 0s 19ms/step - accuracy: 1.0000 - loss: 0.0012 - val_accuracy: 0.6591 - val_loss: 2.342
6
Epoch 72/100
6/6 ━━━━━━ 0s 17ms/step - accuracy: 1.0000 - loss: 9.5112e-04 - val_accuracy: 0.6591 - val_loss: 2
.3442
Epoch 73/100
6/6 ━━━━━━ 0s 16ms/step - accuracy: 1.0000 - loss: 9.8466e-04 - val_accuracy: 0.6591 - val_loss: 2
.3568
Epoch 74/100
6/6 ━━━━━━ 0s 16ms/step - accuracy: 1.0000 - loss: 0.0015 - val_accuracy: 0.6591 - val_loss: 2.362
1
Epoch 75/100
6/6 ━━━━━━ 0s 16ms/step - accuracy: 1.0000 - loss: 0.0014 - val_accuracy: 0.6591 - val_loss: 2.382
6
Epoch 76/100
6/6 ━━━━━━ 0s 16ms/step - accuracy: 1.0000 - loss: 0.0015 - val_accuracy: 0.6591 - val_loss: 2.379
8
Epoch 77/100
6/6 ━━━━━━ 0s 17ms/step - accuracy: 1.0000 - loss: 0.0012 - val_accuracy: 0.6591 - val_loss: 2.387
0
Epoch 78/100
6/6 ━━━━━━ 0s 17ms/step - accuracy: 1.0000 - loss: 0.0015 - val_accuracy: 0.6591 - val_loss: 2.399
3
Epoch 79/100
6/6 ━━━━━━ 0s 14ms/step - accuracy: 1.0000 - loss: 8.7181e-04 - val_accuracy: 0.6591 - val_loss: 2
.3911
Epoch 80/100
6/6 ━━━━━━ 0s 14ms/step - accuracy: 1.0000 - loss: 8.1217e-04 - val_accuracy: 0.6591 - val_loss: 2
.4016
Epoch 81/100
6/6 ━━━━━━ 0s 16ms/step - accuracy: 1.0000 - loss: 7.2600e-04 - val_accuracy: 0.6591 - val_loss: 2
.4098
Epoch 82/100
6/6 ━━━━━━ 0s 16ms/step - accuracy: 1.0000 - loss: 7.8275e-04 - val_accuracy: 0.6591 - val_loss: 2
.4101
Epoch 83/100
6/6 ━━━━━━ 0s 16ms/step - accuracy: 1.0000 - loss: 8.5038e-04 - val_accuracy: 0.6591 - val_loss: 2
.4201
Epoch 84/100
6/6 ━━━━━━ 0s 16ms/step - accuracy: 1.0000 - loss: 7.4225e-04 - val_accuracy: 0.6591 - val_loss: 2
.4313
Epoch 85/100
6/6 ━━━━━━ 0s 15ms/step - accuracy: 1.0000 - loss: 0.0011 - val_accuracy: 0.6591 - val_loss: 2.432
5
Epoch 86/100
6/6 ━━━━━━ 0s 17ms/step - accuracy: 1.0000 - loss: 0.0010 - val_accuracy: 0.6591 - val_loss: 2.433
1
Epoch 87/100
6/6 ━━━━━━ 0s 20ms/step - accuracy: 1.0000 - loss: 6.5057e-04 - val_accuracy: 0.6591 - val_loss: 2
.4475
Epoch 88/100
6/6 ━━━━━━ 0s 17ms/step - accuracy: 1.0000 - loss: 6.6255e-04 - val_accuracy: 0.6591 - val_loss: 2
.4519
Epoch 89/100
6/6 ━━━━━━ 0s 17ms/step - accuracy: 1.0000 - loss: 5.8588e-04 - val_accuracy: 0.6591 - val_loss: 2
.4555
Epoch 90/100
6/6 ━━━━━━ 0s 16ms/step - accuracy: 1.0000 - loss: 6.5398e-04 - val_accuracy: 0.6591 - val_loss: 2
.4585
Epoch 91/100
6/6 ━━━━━━ 0s 15ms/step - accuracy: 1.0000 - loss: 6.9980e-04 - val_accuracy: 0.6591 - val_loss: 2
.4712
Epoch 92/100
6/6 ━━━━━━ 0s 14ms/step - accuracy: 1.0000 - loss: 7.0680e-04 - val_accuracy: 0.6591 - val_loss: 2
.4864
Epoch 93/100
6/6 ━━━━━━ 0s 14ms/step - accuracy: 1.0000 - loss: 7.4559e-04 - val_accuracy: 0.6591 - val_loss: 2
.4860
Epoch 94/100
6/6 ━━━━━━ 0s 15ms/step - accuracy: 1.0000 - loss: 6.9376e-04 - val_accuracy: 0.6591 - val_loss: 2
.4917
Epoch 95/100
6/6 ━━━━━━ 0s 17ms/step - accuracy: 1.0000 - loss: 8.7814e-04 - val_accuracy: 0.6591 - val_loss: 2
.4950
Epoch 96/100
6/6 ━━━━━━ 0s 15ms/step - accuracy: 1.0000 - loss: 6.0234e-04 - val_accuracy: 0.6591 - val_loss: 2
.4965
Epoch 97/100
6/6 ━━━━━━ 0s 15ms/step - accuracy: 1.0000 - loss: 5.7620e-04 - val_accuracy: 0.6591 - val_loss: 2
.5035

```

Epoch 98/100
6/6 0s 16ms/step - accuracy: 1.0000 - loss: 5.6862e-04 - val_accuracy: 0.6591 - val_loss: 2
.5113
Epoch 99/100
6/6 0s 15ms/step - accuracy: 1.0000 - loss: 6.9141e-04 - val_accuracy: 0.6591 - val_loss: 2
.5193
Epoch 100/100
6/6 0s 15ms/step - accuracy: 1.0000 - loss: 7.4664e-04 - val_accuracy: 0.6591 - val_loss: 2
.5302

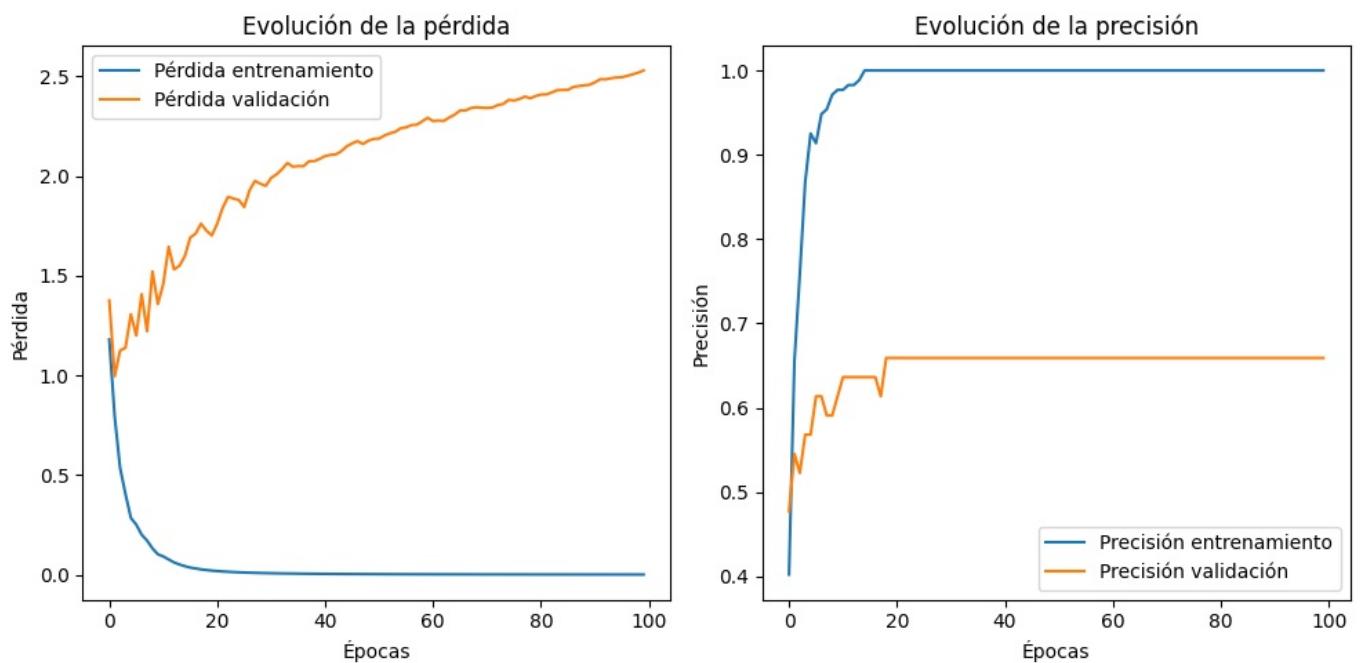
```

In [73]: *## Visualizamos la evolución de la función de pérdida y la precisión*

```

fig, ax = plt.subplots(1, 2, figsize=(10, 5))
ax[0].plot(history_simple.history['loss'], label='Pérdida entrenamiento')
ax[0].plot(history_simple.history['val_loss'], label='Pérdida validación')
ax[0].set_title('Evolución de la pérdida')
ax[0].set_xlabel('Épocas')
ax[0].set_ylabel('Pérdida')
ax[0].legend()
ax[1].plot(history_simple.history['accuracy'], label='Precisión entrenamiento')
ax[1].plot(history_simple.history['val_accuracy'], label='Precisión validación')
ax[1].set_title('Evolución de la precisión')
ax[1].set_xlabel('Épocas')
ax[1].set_ylabel('Precisión')
ax[1].legend()
plt.tight_layout()
plt.show()

```



- **Conclusión:**

Nuestro modelo parece tener un problema de sobreentrenamiento.

El conjunto de datos es excesivamente pequeño, no se logran aprender patrones generales y el modelo se adapta a los datos de entrenamiento. Esto es un problema, ya que el modelo no generaliza bien y no es capaz de clasificar correctamente las imágenes de test.

- **Nota:**

Esto no es más que una exploración de posibles modelos, por eso no se han realizado experimentaciones con el tamaño del conjunto de entrenamiento, ni con la arquitectura del modelo. Se ha optado por un modelo simple para ver si es capaz de aprender algo. Más adelante, trataremos de optimizar alguno de los modelos.

Predecimos las etiquetas de las imágenes de test y presentamos los resultados de forma visual. Imagen original, etiqueta real y etiqueta predicha por el modelo.

Evaluamos en el conjunto de test y obtenemos las métricas de evaluación.

In [74]: `prediccion_simple = model_simple.predict(x_test)`

```

## Comparamos la predicción con las etiquetas y visualizamos todas las imágenes
fig, ax = plt.subplots(5, 5, figsize=(15, 15))
for i in range(5):
    for j in range(5):
        idx = i * 5 + j

```

```

ax[i, j].imshow(x_test[idx].reshape(64, 64, 3))
ax[i, j].set_title(f"Predicción: {np.argmax(prediccion_simple[idx])}, Etiqueta: {y_test[idx]}")
ax[i, j].axis("off")

```

1/1 ————— 0s 47ms/step

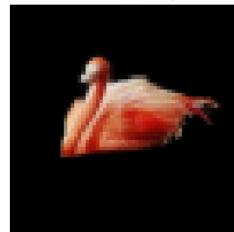
Predicción: 2, Etiqueta: 0



Predicción: 0, Etiqueta: 0



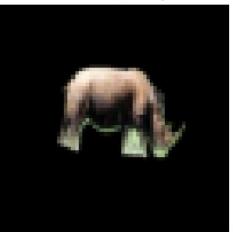
Predicción: 2, Etiqueta: 2



Predicción: 1, Etiqueta: 1



Predicción: 1, Etiqueta: 1



Predicción: 2, Etiqueta: 2



Predicción: 1, Etiqueta: 1



Predicción: 1, Etiqueta: 2



Predicción: 2, Etiqueta: 0



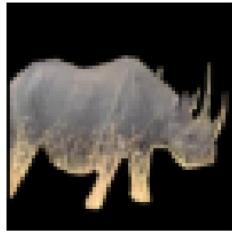
Predicción: 2, Etiqueta: 2



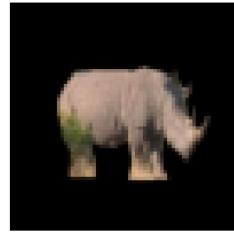
Predicción: 0, Etiqueta: 0



Predicción: 1, Etiqueta: 1



Predicción: 2, Etiqueta: 1



Predicción: 2, Etiqueta: 2



Predicción: 2, Etiqueta: 2



Predicción: 2, Etiqueta: 2



Predicción: 2, Etiqueta: 1



Predicción: 0, Etiqueta: 0



Predicción: 1, Etiqueta: 2



Predicción: 2, Etiqueta: 2



Predicción: 0, Etiqueta: 0



Predicción: 1, Etiqueta: 0



Predicción: 2, Etiqueta: 2



Predicción: 0, Etiqueta: 0



Predicción: 1, Etiqueta: 1



```

In [75]: ## Evaluación en el conjunto de test
score_simple = model_simple.evaluate(x_test, y_test)
print(f"Test accuracy: {score_simple[1]}")

```

1/1 ————— 0s 33ms/step - accuracy: 0.7200 - loss: 2.3968

Test accuracy: 0.7200000286102295

Definimos una función auxiliar para presentar de forma muy visual la matriz de confusión (como heatmap)

```

In [11]: def confusion_matrix_plot(cm):
    """
    Función para visualizar la matriz de confusión de forma bonita con seaborn.
    """
    plt.figure(figsize=(4, 3))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
                xticklabels=['Elefante', 'Rinoceronte', 'Otro'],
                yticklabels=['Elefante', 'Rinoceronte', 'Otro'])
    plt.xlabel('Predicción')
    plt.ylabel('Real')
    plt.title('Matriz de confusión')
    plt.show()

```

```

In [77]: ## Matriz de confusión
y_pred_simple = model_simple.predict(x_test)
y_pred_simple = np.argmax(y_pred_simple, axis=1)

```

```

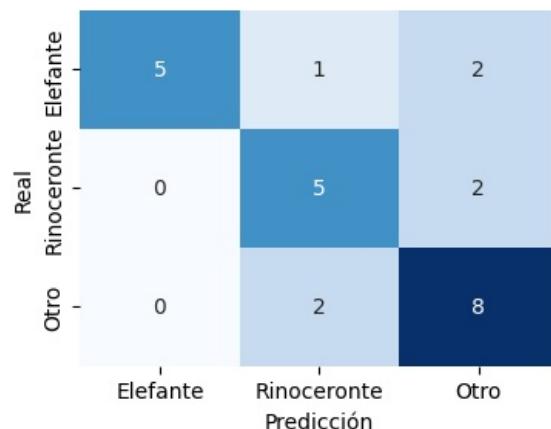
cm_simple = confusion_matrix(y_test, y_pred_simple)
confusion_matrix_plot(cm_simple)

## Reporte de clasificación
print(classification_report(y_test, y_pred_simple, target_names=['Elefante', 'Rinoceronte', 'Otro']))

```

1/1 ————— 0s 25ms/step

Matriz de confusión



	precision	recall	f1-score	support
Elefante	1.00	0.62	0.77	8
Rinoceronte	0.62	0.71	0.67	7
Otro	0.67	0.80	0.73	10
accuracy			0.72	25
macro avg	0.76	0.71	0.72	25
weighted avg	0.76	0.72	0.72	25

2. SVM

```

In [79]: ## Creamos el modelo SVM
svm_model = SVC(kernel='linear', C=1.0, random_state=42)

## Entrenamos el modelo SVM
svm_model.fit(x_train, y_train)

## Predicción sobre el conjunto de test
y_pred_svm = svm_model.predict(x_test)
print(f"Test accuracy SVM: {np.mean(y_pred_svm == y_test)}")

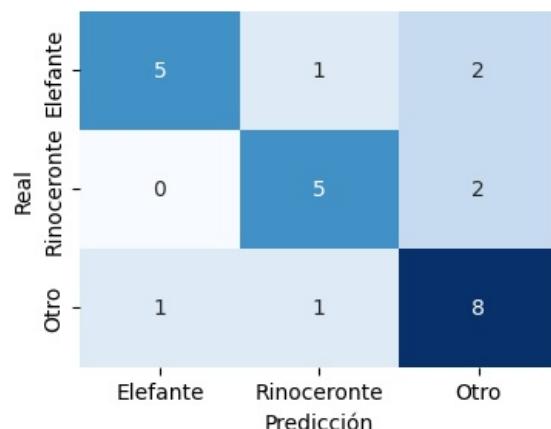
## Matriz de confusión
cm_svm = confusion_matrix(y_test, y_pred_svm)
confusion_matrix_plot(cm_svm)

## Reporte de clasificación
print(classification_report(y_test, y_pred_svm, target_names=['Elefante', 'Rinoceronte', 'Otro']))

```

Test accuracy SVM: 0.72

Matriz de confusión



	precision	recall	f1-score	support
Elefante	0.83	0.62	0.71	8
Rinoceronte	0.71	0.71	0.71	7
Otro	0.67	0.80	0.73	10
accuracy			0.72	25
macro avg	0.74	0.71	0.72	25
weighted avg	0.73	0.72	0.72	25

3. Random Forest

```
In [80]: ## Creamos el modelo Random Forest
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)

## Entrenamos el modelo Random Forest
rf_model.fit(x_train, y_train)

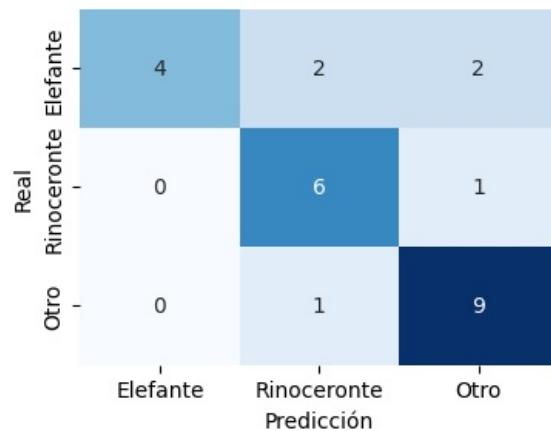
## Predicción sobre el conjunto de test
y_pred_rf = rf_model.predict(x_test)
print(f"Test accuracy RF: {np.mean(y_pred_rf == y_test)}")

## Matriz de confusión
cm_rf = confusion_matrix(y_test, y_pred_rf)
confusion_matrix_plot(cm_rf)

## Reporte de clasificación
print(classification_report(y_test, y_pred_rf, target_names=['Elefante', 'Rinoceronte', 'Otro']))
```

Test accuracy RF: 0.76

Matriz de confusión



	precision	recall	f1-score	support
Elefante	1.00	0.50	0.67	8
Rinoceronte	0.67	0.86	0.75	7
Otro	0.75	0.90	0.82	10
accuracy			0.76	25
macro avg	0.81	0.75	0.74	25
weighted avg	0.81	0.76	0.75	25

- **Conclusión:**

En general, los tres modelos parecen ofrecer bastantes buenos resultados, entorno a 0.75 de precisión en el conjunto de test.

Los tres modelos consiguen diferenciar bastante bien elefantes de rinocerontes, dándose principalmente los errores en clasificar estos dos animales como otros. Concretamente en ningún modelo un rinoceronte es predicho como elefante y solamente un elefante es predicho como rinoceronte (en Random Forest son 2).

Los resultados son buenos para no haber experimentado con la arquitectura de los modelos ni con la resolución de las imágenes aplanadas.

Cabe destacar que la limitación de tamaño del conjunto de evaluación puede afectar a las conclusiones.

2. Modelo que utiliza características generales de la imagen

En este modelo vamos a utilizar **características generales** de color y forma.

Definimos una función para extraer características de la imagen. Esta función recibe como entrada la imagen y la máscara y devuelve un vector de características.

- Área del animal (número de píxeles de la máscara)
- Perímetro del animal (número de píxeles de la máscara)
- Media de los canales R, G y B de la figura (imagen recortada para no utilizar el fondo)
- Media de los canales H, S y V de la figura (imagen recortada para no utilizar el fondo)
- Centroide de la figura (coordenadas x e y del centroide de la máscara)
- Ejes mayor y menor de la figura (distancia mayor y menor entre 2 puntos de la figura)

```
In [12]: def extraer_caracteristicas_estadisticas(imagenes, mascaras):
    """
    Extrae características estadísticas de las imágenes y sus máscaras.

    Args:
        imagenes (ndarray): Imágenes de entrada.
        mascaras (ndarray): Máscaras de entrada.

    Returns:
        list: Lista de características extraídas.
    """
    caracteristicas = []
    for i in range(len(imagenes)):
        ## Área de la figura
        area = np.sum(mascaras[i])

        ## Perímetro de la figura
        perimetro = np.sum(np.abs(np.diff(mascaras[i], axis=0))) + np.sum(np.abs(np.diff(mascaras[i], axis=1)))

        ## Media de los canales R, G y B (imagen recortada)
        media_r = np.mean(imagenes[i][mascaras[i] == 1][:, 0])
        media_g = np.mean(imagenes[i][mascaras[i] == 1][:, 1])
        media_b = np.mean(imagenes[i][mascaras[i] == 1][:, 2])

        ## Media canales H, S y V (imagen recortada)
        imagen_hsv = skimage.color.rgb2hsv(imagenes[i])
        media_h = np.mean(imagen_hsv[mascaras[i] == 1][:, 0])
        media_s = np.mean(imagen_hsv[mascaras[i] == 1][:, 1])
        media_v = np.mean(imagen_hsv[mascaras[i] == 1][:, 2])

        ## Centroide de la figura
        centroide = np.mean(np.argwhere(mascaras[i] == 1), axis=0)
        centroide_x = int(centroide[0])
        centroide_y = int(centroide[1])

        ## Eje mayor y menor de la figura
        ejes = np.max(np.argwhere(mascaras[i] == 1), axis=0) - np.min(np.argwhere(mascaras[i] == 1), axis=0)
        eje_mayor = np.max(ejes)
        eje_menor = np.min(ejes)

        caracteristicas.append([area, perimetro, media_r, media_g, media_b, media_h, media_s, media_v, centroide_x, centroide_y, eje_mayor, eje_menor])

    return np.array(caracteristicas)
```

- **Aclaración:**

La información de centro de masas y de ejes mayor y menor fueron añadidas posteriormente a la función de características. En un principio no se consideraron como características, pero se ha visto que pueden ser interesantes para la clasificación.

- **Conclusión:**

Realmente el área y el perímetro de la figura no son muy buenas características en un problema de clasificación de reconocimiento de objetos ya que dependen de manera directa de la escala de la imagen.

En nuestro caso lo utilizamos porque las imágenes tienden todas a tener un plano similar, y el animal está en primer plano. Pero hay que destacar que en caso de que las imágenes tuvieran diferentes escalas, estas características no serían efectivas.

Extraemos el vector de características de todas las imágenes haciendo uso de la función definida anteriormente.

```
In [13]: caracteristicas = extraer_caracteristicas_estadisticas(imagenes, mascaras)
```

Vemos las características extraídas de la una imagen aleatoria junto con la imagen elegida

```
In [16]: ## Índice aleatorio
idx = np.random.randint(0, len(imagenes))

plt.imshow(imagenes[idx])
plt.title("Imagen original")
plt.axis("off")
plt.show()
```

```

print(f"Características extraídas de la primera imagen:")
print(f"Área: {caracteristicas[idx][0]}")
print(f"Perímetro: {caracteristicas[idx][1]}")
print(f"Media R: {caracteristicas[idx][2]}")
print(f"Media G: {caracteristicas[idx][3]}")
print(f"Media B: {caracteristicas[idx][4]}")
print(f"Media H: {caracteristicas[idx][5]}")
print(f"Media S: {caracteristicas[idx][6]}")
print(f"Media V: {caracteristicas[idx][7]}")
print(f"Centroide X: {caracteristicas[idx][8]}")
print(f"Centroide Y: {caracteristicas[idx][9]}")
print(f"Eje mayor: {caracteristicas[idx][10]}")
print(f"Eje menor: {caracteristicas[idx][11]}")

```

Imagen original



Características extraídas de la primera imagen:

```

Área: 22453.0
Perímetro: 1260.0
Media R: 0.2999903564308189
Media G: 0.2145362244619043
Media B: 0.08997024260699696
Media H: 0.12226420063351315
Media S: 0.7411080423544347
Media V: 0.30141481491184635
Centroide X: 106.0
Centroide Y: 128.0
Eje mayor: 205.0
Eje menor: 168.0

```

Dividimos, igual que antes, en conjunto de entrenamiento y test.

```
In [17]: x_train, x_test, y_train, y_test = train_test_split(caracteristicas, etiquetas, test_size=0.1, random_state=42)
print(f"Dimensiones de x_train: {x_train.shape}")
print(f"Dimensiones de x_test: {x_test.shape}")
print(f"Dimensiones de y_train: {y_train.shape}")
print(f"Dimensiones de y_test: {y_test.shape}")

Dimensiones de x_train: (218, 12)
Dimensiones de x_test: (25, 12)
Dimensiones de y_train: (218,)
Dimensiones de y_test: (25,)
```

1. MLP

Modelo de red neuronal simple, construido igual que el anterior.

```
In [18]: ## Creamos el modelo
model_simple = modelo_simple(input_shape=(12,))
model_simple.summary()

## Entrenamos el modelo
history_simple = model_simple.fit(x_train, y_train, epochs=100, batch_size=32, validation_split=0.2)

Model: "functional"
```

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 12)	0
dense (Dense)	(None, 64)	832
dense_1 (Dense)	(None, 32)	2,080
dense_2 (Dense)	(None, 3)	99

Total params: 3,011 (11.76 KB)

Trainable params: 3,011 (11.76 KB)

Non-trainable params: 0 (0.00 B)

Epoch 1/100

6/6  2s 72ms/step - accuracy: 0.2804 - loss: 599.6578 - val_accuracy: 0.4545 - val_loss: 305.0894

Epoch 2/100

6/6  0s 21ms/step - accuracy: 0.5207 - loss: 235.2468 - val_accuracy: 0.2045 - val_loss: 162.6183

Epoch 3/100

6/6  0s 20ms/step - accuracy: 0.2716 - loss: 191.4586 - val_accuracy: 0.3409 - val_loss: 127.3662

Epoch 4/100

6/6  0s 20ms/step - accuracy: 0.2415 - loss: 117.4675 - val_accuracy: 0.2727 - val_loss: 209.2548

Epoch 5/100

6/6  0s 20ms/step - accuracy: 0.3844 - loss: 120.7200 - val_accuracy: 0.3409 - val_loss: 38.7652

Epoch 6/100

6/6  0s 22ms/step - accuracy: 0.3775 - loss: 38.1393 - val_accuracy: 0.4091 - val_loss: 56.7669

Epoch 7/100

6/6  0s 22ms/step - accuracy: 0.5543 - loss: 24.8630 - val_accuracy: 0.3636 - val_loss: 58.8331

Epoch 8/100

6/6  0s 24ms/step - accuracy: 0.5215 - loss: 50.1531 - val_accuracy: 0.3864 - val_loss: 71.2099

Epoch 9/100

6/6  0s 24ms/step - accuracy: 0.4189 - loss: 54.4247 - val_accuracy: 0.4545 - val_loss: 69.3181

Epoch 10/100

6/6  0s 22ms/step - accuracy: 0.4761 - loss: 45.7808 - val_accuracy: 0.6136 - val_loss: 17.0782

Epoch 11/100

6/6  0s 23ms/step - accuracy: 0.5065 - loss: 26.3705 - val_accuracy: 0.6364 - val_loss: 16.2517

Epoch 12/100

6/6  0s 24ms/step - accuracy: 0.6234 - loss: 19.0342 - val_accuracy: 0.4091 - val_loss: 40.9492

Epoch 13/100

6/6  0s 20ms/step - accuracy: 0.5584 - loss: 18.3605 - val_accuracy: 0.4318 - val_loss: 20.4823

Epoch 14/100

6/6  0s 26ms/step - accuracy: 0.5304 - loss: 15.4999 - val_accuracy: 0.6364 - val_loss: 17.1539

Epoch 15/100

6/6  0s 23ms/step - accuracy: 0.5799 - loss: 12.3527 - val_accuracy: 0.5455 - val_loss: 23.6826

Epoch 16/100

6/6  0s 20ms/step - accuracy: 0.5037 - loss: 18.9714 - val_accuracy: 0.4318 - val_loss: 17.0028

Epoch 17/100

6/6  0s 24ms/step - accuracy: 0.5411 - loss: 15.0700 - val_accuracy: 0.3409 - val_loss: 51.4649

Epoch 18/100

6/6  0s 21ms/step - accuracy: 0.3909 - loss: 34.6368 - val_accuracy: 0.4318 - val_loss: 23.4815

Epoch 19/100

6/6  0s 24ms/step - accuracy: 0.4022 - loss: 33.9360 - val_accuracy: 0.4091 - val_loss: 46.6087

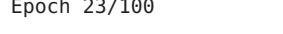
Epoch 20/100

6/6  0s 21ms/step - accuracy: 0.5334 - loss: 29.3848 - val_accuracy: 0.5000 - val_loss: 16.1397

Epoch 21/100

6/6  0s 23ms/step - accuracy: 0.5560 - loss: 17.9560 - val_accuracy: 0.4773 - val_loss: 24.4375

Epoch 22/100

6/6  0s 23ms/step - accuracy: 0.4439 - loss: 22.1918 - val_accuracy: 0.5682 - val_loss: 42.3628

Epoch 23/100

6/6 ━━━━━━ 0s 21ms/step - accuracy: 0.5146 - loss: 46.4414 - val_accuracy: 0.4091 - val_loss: 49.9
080
Epoch 24/100
6/6 ━━━━━━ 0s 26ms/step - accuracy: 0.5265 - loss: 28.7309 - val_accuracy: 0.6364 - val_loss: 26.3
739
Epoch 25/100
6/6 ━━━━━━ 0s 22ms/step - accuracy: 0.5977 - loss: 25.6378 - val_accuracy: 0.4773 - val_loss: 29.1
719
Epoch 26/100
6/6 ━━━━━━ 0s 22ms/step - accuracy: 0.5075 - loss: 31.2069 - val_accuracy: 0.4773 - val_loss: 19.8
168
Epoch 27/100
6/6 ━━━━━━ 0s 24ms/step - accuracy: 0.5897 - loss: 12.8630 - val_accuracy: 0.4318 - val_loss: 26.1
770
Epoch 28/100
6/6 ━━━━━━ 0s 21ms/step - accuracy: 0.6091 - loss: 18.9627 - val_accuracy: 0.4318 - val_loss: 42.0
468
Epoch 29/100
6/6 ━━━━━━ 0s 24ms/step - accuracy: 0.5850 - loss: 23.2326 - val_accuracy: 0.5909 - val_loss: 17.1
021
Epoch 30/100
6/6 ━━━━━━ 0s 25ms/step - accuracy: 0.5395 - loss: 20.9806 - val_accuracy: 0.4545 - val_loss: 34.0
501
Epoch 31/100
6/6 ━━━━━━ 0s 32ms/step - accuracy: 0.5769 - loss: 20.5369 - val_accuracy: 0.5455 - val_loss: 22.8
591
Epoch 32/100
6/6 ━━━━━━ 0s 26ms/step - accuracy: 0.5885 - loss: 16.1489 - val_accuracy: 0.4773 - val_loss: 34.5
148
Epoch 33/100
6/6 ━━━━━━ 0s 25ms/step - accuracy: 0.6409 - loss: 25.2415 - val_accuracy: 0.5455 - val_loss: 30.8
353
Epoch 34/100
6/6 ━━━━━━ 0s 24ms/step - accuracy: 0.4944 - loss: 31.9580 - val_accuracy: 0.4545 - val_loss: 44.0
027
Epoch 35/100
6/6 ━━━━━━ 0s 22ms/step - accuracy: 0.5367 - loss: 25.4126 - val_accuracy: 0.7045 - val_loss: 13.9
471
Epoch 36/100
6/6 ━━━━━━ 0s 27ms/step - accuracy: 0.5921 - loss: 15.6614 - val_accuracy: 0.4318 - val_loss: 26.1
396
Epoch 37/100
6/6 ━━━━━━ 0s 24ms/step - accuracy: 0.5998 - loss: 15.4433 - val_accuracy: 0.4318 - val_loss: 16.6
700
Epoch 38/100
6/6 ━━━━━━ 0s 24ms/step - accuracy: 0.5797 - loss: 16.3838 - val_accuracy: 0.6364 - val_loss: 20.7
067
Epoch 39/100
6/6 ━━━━━━ 0s 23ms/step - accuracy: 0.5737 - loss: 27.5301 - val_accuracy: 0.6364 - val_loss: 12.7
571
Epoch 40/100
6/6 ━━━━━━ 0s 22ms/step - accuracy: 0.5222 - loss: 18.7495 - val_accuracy: 0.4545 - val_loss: 43.4
068
Epoch 41/100
6/6 ━━━━━━ 0s 22ms/step - accuracy: 0.5473 - loss: 31.9200 - val_accuracy: 0.6818 - val_loss: 23.8
491
Epoch 42/100
6/6 ━━━━━━ 0s 23ms/step - accuracy: 0.5797 - loss: 26.7363 - val_accuracy: 0.6136 - val_loss: 26.5
705
Epoch 43/100
6/6 ━━━━━━ 0s 22ms/step - accuracy: 0.6101 - loss: 22.2011 - val_accuracy: 0.3636 - val_loss: 36.4
369
Epoch 44/100
6/6 ━━━━━━ 0s 28ms/step - accuracy: 0.4277 - loss: 40.9918 - val_accuracy: 0.3864 - val_loss: 103.
3833
Epoch 45/100
6/6 ━━━━━━ 0s 23ms/step - accuracy: 0.5435 - loss: 53.1850 - val_accuracy: 0.6364 - val_loss: 49.4
720
Epoch 46/100
6/6 ━━━━━━ 0s 22ms/step - accuracy: 0.5846 - loss: 48.3037 - val_accuracy: 0.5227 - val_loss: 23.6
831
Epoch 47/100
6/6 ━━━━━━ 0s 21ms/step - accuracy: 0.6666 - loss: 21.7051 - val_accuracy: 0.4545 - val_loss: 37.5
004
Epoch 48/100
6/6 ━━━━━━ 0s 20ms/step - accuracy: 0.5068 - loss: 26.7605 - val_accuracy: 0.4545 - val_loss: 45.0
386
Epoch 49/100
6/6 ━━━━━━ 0s 21ms/step - accuracy: 0.5763 - loss: 29.8735 - val_accuracy: 0.5000 - val_loss: 40.1
385
Epoch 50/100
6/6 ━━━━━━ 0s 21ms/step - accuracy: 0.5114 - loss: 38.0082 - val_accuracy: 0.6136 - val_loss: 29.1
126

Epoch 51/100
6/6 0s 22ms/step - accuracy: 0.5436 - loss: 37.6165 - val_accuracy: 0.4545 - val_loss: 77.7
041
Epoch 52/100
6/6 0s 22ms/step - accuracy: 0.5613 - loss: 45.4198 - val_accuracy: 0.4773 - val_loss: 33.4
272
Epoch 53/100
6/6 0s 19ms/step - accuracy: 0.4944 - loss: 30.0985 - val_accuracy: 0.3182 - val_loss: 48.5
535
Epoch 54/100
6/6 0s 21ms/step - accuracy: 0.4099 - loss: 41.3830 - val_accuracy: 0.4091 - val_loss: 31.3
755
Epoch 55/100
6/6 0s 21ms/step - accuracy: 0.5905 - loss: 20.4040 - val_accuracy: 0.3864 - val_loss: 39.2
456
Epoch 56/100
6/6 0s 21ms/step - accuracy: 0.6261 - loss: 30.6117 - val_accuracy: 0.6136 - val_loss: 39.3
911
Epoch 57/100
6/6 0s 25ms/step - accuracy: 0.5534 - loss: 35.9975 - val_accuracy: 0.3864 - val_loss: 55.2
943
Epoch 58/100
6/6 0s 21ms/step - accuracy: 0.5775 - loss: 28.5571 - val_accuracy: 0.5909 - val_loss: 20.9
765
Epoch 59/100
6/6 0s 20ms/step - accuracy: 0.5739 - loss: 24.5694 - val_accuracy: 0.5000 - val_loss: 36.9
872
Epoch 60/100
6/6 0s 20ms/step - accuracy: 0.4573 - loss: 34.3559 - val_accuracy: 0.5227 - val_loss: 20.3
501
Epoch 61/100
6/6 0s 20ms/step - accuracy: 0.5324 - loss: 23.7307 - val_accuracy: 0.4318 - val_loss: 51.5
050
Epoch 62/100
6/6 0s 20ms/step - accuracy: 0.5825 - loss: 46.3025 - val_accuracy: 0.2955 - val_loss: 114.
2878
Epoch 63/100
6/6 0s 21ms/step - accuracy: 0.4485 - loss: 60.3532 - val_accuracy: 0.6364 - val_loss: 48.4
339
Epoch 64/100
6/6 0s 21ms/step - accuracy: 0.5197 - loss: 54.1819 - val_accuracy: 0.4091 - val_loss: 46.6
311
Epoch 65/100
6/6 0s 20ms/step - accuracy: 0.5741 - loss: 31.7884 - val_accuracy: 0.4545 - val_loss: 76.1
013
Epoch 66/100
6/6 0s 20ms/step - accuracy: 0.5920 - loss: 38.2430 - val_accuracy: 0.5227 - val_loss: 55.5
974
Epoch 67/100
6/6 0s 21ms/step - accuracy: 0.4619 - loss: 54.7599 - val_accuracy: 0.3864 - val_loss: 100.
0639
Epoch 68/100
6/6 0s 20ms/step - accuracy: 0.5437 - loss: 60.4609 - val_accuracy: 0.6591 - val_loss: 17.7
661
Epoch 69/100
6/6 0s 20ms/step - accuracy: 0.6391 - loss: 22.8028 - val_accuracy: 0.6591 - val_loss: 20.5
353
Epoch 70/100
6/6 0s 24ms/step - accuracy: 0.5586 - loss: 14.9500 - val_accuracy: 0.5000 - val_loss: 38.2
725
Epoch 71/100
6/6 0s 21ms/step - accuracy: 0.6187 - loss: 29.6902 - val_accuracy: 0.5909 - val_loss: 24.6
299
Epoch 72/100
6/6 0s 21ms/step - accuracy: 0.5445 - loss: 20.5616 - val_accuracy: 0.4545 - val_loss: 41.0
769
Epoch 73/100
6/6 0s 21ms/step - accuracy: 0.5412 - loss: 26.0619 - val_accuracy: 0.5000 - val_loss: 28.8
098
Epoch 74/100
6/6 0s 22ms/step - accuracy: 0.6115 - loss: 15.8718 - val_accuracy: 0.4091 - val_loss: 48.1
223
Epoch 75/100
6/6 0s 20ms/step - accuracy: 0.6144 - loss: 26.7944 - val_accuracy: 0.4318 - val_loss: 36.4
500
Epoch 76/100
6/6 0s 21ms/step - accuracy: 0.4480 - loss: 34.1626 - val_accuracy: 0.4545 - val_loss: 47.5
017
Epoch 77/100
6/6 0s 20ms/step - accuracy: 0.5852 - loss: 26.1540 - val_accuracy: 0.5000 - val_loss: 28.3
695
Epoch 78/100
6/6 0s 19ms/step - accuracy: 0.4628 - loss: 31.9560 - val_accuracy: 0.4545 - val_loss: 41.9

759
 Epoch 79/100
 6/6 0s 20ms/step - accuracy: 0.5900 - loss: 30.1892 - val_accuracy: 0.4773 - val_loss: 40.8
 280
 Epoch 80/100
 6/6 0s 22ms/step - accuracy: 0.5942 - loss: 25.6728 - val_accuracy: 0.5000 - val_loss: 25.8
 312
 Epoch 81/100
 6/6 0s 20ms/step - accuracy: 0.5823 - loss: 15.6179 - val_accuracy: 0.4318 - val_loss: 27.1
 779
 Epoch 82/100
 6/6 0s 22ms/step - accuracy: 0.5879 - loss: 18.6342 - val_accuracy: 0.6136 - val_loss: 19.5
 883
 Epoch 83/100
 6/6 0s 22ms/step - accuracy: 0.6443 - loss: 16.0898 - val_accuracy: 0.5000 - val_loss: 28.6
 987
 Epoch 84/100
 6/6 0s 22ms/step - accuracy: 0.5942 - loss: 15.8597 - val_accuracy: 0.6364 - val_loss: 19.3
 469
 Epoch 85/100
 6/6 0s 21ms/step - accuracy: 0.5857 - loss: 17.1892 - val_accuracy: 0.5227 - val_loss: 18.4
 726
 Epoch 86/100
 6/6 0s 19ms/step - accuracy: 0.6247 - loss: 8.9784 - val_accuracy: 0.5909 - val_loss: 14.23
 74
 Epoch 87/100
 6/6 0s 20ms/step - accuracy: 0.6153 - loss: 12.5988 - val_accuracy: 0.4091 - val_loss: 22.8
 763
 Epoch 88/100
 6/6 0s 20ms/step - accuracy: 0.5306 - loss: 16.4878 - val_accuracy: 0.4545 - val_loss: 44.7
 020
 Epoch 89/100
 6/6 0s 22ms/step - accuracy: 0.5408 - loss: 34.7952 - val_accuracy: 0.5000 - val_loss: 34.8
 337
 Epoch 90/100
 6/6 0s 20ms/step - accuracy: 0.6203 - loss: 22.8200 - val_accuracy: 0.4091 - val_loss: 83.3
 434
 Epoch 91/100
 6/6 0s 21ms/step - accuracy: 0.4591 - loss: 54.7309 - val_accuracy: 0.4773 - val_loss: 45.4
 504
 Epoch 92/100
 6/6 0s 19ms/step - accuracy: 0.4921 - loss: 37.1980 - val_accuracy: 0.5455 - val_loss: 43.2
 155
 Epoch 93/100
 6/6 0s 19ms/step - accuracy: 0.6298 - loss: 23.8573 - val_accuracy: 0.3864 - val_loss: 35.7
 564
 Epoch 94/100
 6/6 0s 31ms/step - accuracy: 0.6435 - loss: 14.0664 - val_accuracy: 0.4773 - val_loss: 19.1
 125
 Epoch 95/100
 6/6 0s 21ms/step - accuracy: 0.5857 - loss: 19.8874 - val_accuracy: 0.2955 - val_loss: 77.7
 735
 Epoch 96/100
 6/6 0s 22ms/step - accuracy: 0.4593 - loss: 44.5587 - val_accuracy: 0.4773 - val_loss: 27.0
 636
 Epoch 97/100
 6/6 0s 22ms/step - accuracy: 0.6160 - loss: 22.3721 - val_accuracy: 0.4091 - val_loss: 23.7
 011
 Epoch 98/100
 6/6 0s 22ms/step - accuracy: 0.5549 - loss: 21.9918 - val_accuracy: 0.6818 - val_loss: 20.9
 454
 Epoch 99/100
 6/6 0s 22ms/step - accuracy: 0.5576 - loss: 24.5191 - val_accuracy: 0.6364 - val_loss: 21.6
 807
 Epoch 100/100
 6/6 0s 22ms/step - accuracy: 0.5731 - loss: 22.9770 - val_accuracy: 0.6591 - val_loss: 15.1
 219

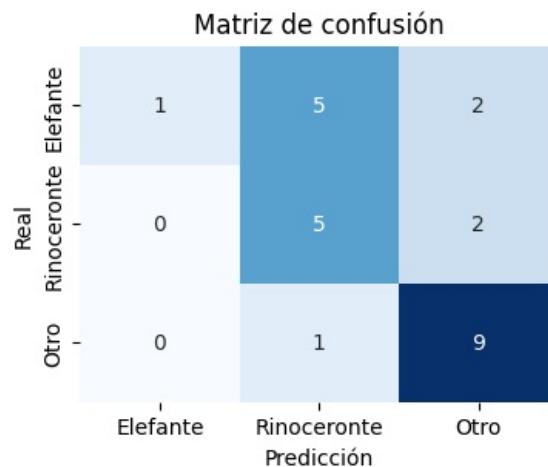
Métricas de evaluación en el conjunto de test.

```
In [19]: ## Evaluación en el conjunto de test
score_simple = model_simple.evaluate(x_test, y_test)
print(f"Test accuracy: {score_simple[1]}")

## Matriz de confusión
y_pred_simple = model_simple.predict(x_test)
y_pred_simple = np.argmax(y_pred_simple, axis=1)
cm_simple = confusion_matrix(y_test, y_pred_simple)
confusion_matrix_plot(cm_simple)

## Reporte de clasificación
print(classification_report(y_test, y_pred_simple, target_names=['Elefante', 'Rinoceronte', 'Otro'], zero_divis:
```

```
1/1 ━━━━━━━━ 0s 38ms/step - accuracy: 0.6000 - loss: 15.2878
Test accuracy: 0.6000000238418579
1/1 ━━━━━━ 0s 70ms/step
```



	precision	recall	f1-score	support
Elefante	1.00	0.12	0.22	8
Rinoceronte	0.45	0.71	0.56	7
Otro	0.69	0.90	0.78	10
accuracy			0.60	25
macro avg	0.72	0.58	0.52	25
weighted avg	0.72	0.60	0.54	25

2. SVM

Modelo SVM con kernel lineal.

```
In [ ]: svm_model = SVC(kernel='linear', C=1.0, random_state=42)
svm_model.fit(x_train, y_train)
```

```
Out[ ]: SVC(kernel='linear', random_state=422)
```

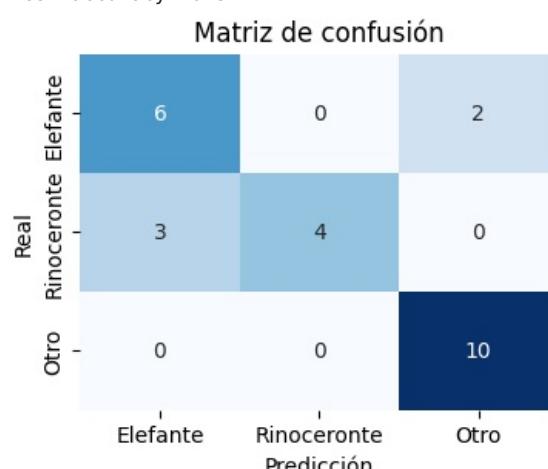
Métricas de evaluación en el conjunto de test.

```
In [27]: ## Evaluación en el conjunto de test
score_svm = svm_model.score(x_test, y_test)
print(f"Test accuracy: {score_svm}")

## Matriz de confusión
cm_svm = confusion_matrix(y_test, svm_model.predict(x_test))
confusion_matrix_plot(cm_svm)

## Informe de clasificación
print("\nInforme de clasificación:")
print("---*30")
print(classification_report(y_test, svm_model.predict(x_test), target_names=["Elefante", "Rinoceronte", "Otros"]))

Test accuracy: 0.8
```



Informe de clasificación:

	precision	recall	f1-score	support
Elefante	0.67	0.75	0.71	8
Rinoceronte	1.00	0.57	0.73	7
Otros	0.83	1.00	0.91	10
accuracy			0.80	25
macro avg	0.83	0.77	0.78	25
weighted avg	0.83	0.80	0.79	25

3. Random Forest

Clasificador Random Forest con 100 árboles.

```
In [24]: rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(x_train, y_train)
```

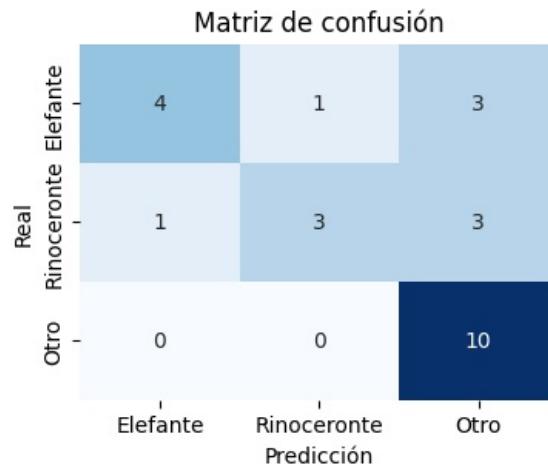
```
Out[24]: RandomForestClassifier(random_state=42)
```

```
In [25]: score_rf = rf_model.score(x_test, y_test)
print(f"Test accuracy: {score_rf}")
```

```
## Matriz de confusión
cm_rf = confusion_matrix(y_test, rf_model.predict(x_test))
confusion_matrix_plot(cm_rf)
```

```
## Informe de clasificación
print("\nInforme de clasificación:")
print("-"*30)
print(classification_report(y_test, rf_model.predict(x_test), target_names=["Elefante", "Rinoceronte", "Otros"])
```

Test accuracy: 0.68



Informe de clasificación:

	precision	recall	f1-score	support
Elefante	0.80	0.50	0.62	8
Rinoceronte	0.75	0.43	0.55	7
Otros	0.62	1.00	0.77	10
accuracy			0.68	25
macro avg	0.72	0.64	0.64	25
weighted avg	0.72	0.68	0.66	25

• Conclusiones:

Los resultados son peores que los obtenidos con el modelo anterior. La precisión en el conjunto de test es de entorno a 0.6 en el caso de la red neuronal y random forest, lo que indica que el modelo no es capaz de clasificar de forma correcta. El caso de SVC, ofrece un mejor resultado, con una precisión del 80%.

Las características generales empleadas parecen no ser suficientes para una clasificación efectiva. En un principio no se habían utilizado las coordenadas del centroide y los ejes mayor y menor y el rendimiento era aun peor.

Nuestros modelos tienen problemas para aprender patrones y la generalización no es buena, es posible que aumentando el número

de características generales se aumente el rendimiento del modelo. Si se decide optimizar este modelo, se podría añadir algún otro descriptor más adelante.

3. Modelo que utiliza información de bordes como entrada

La idea inicial era extraer información sobre los **bordes y la forma** de los objetos usando **gradientes y direcciones**, porque intuía que podían ser útiles para diferenciar imágenes de las clases elefante y rinoceronte. Tras un proceso previo de búsqueda de información y planteamiento del procesamiento, vi que HOG (Histogram of Oriented Gradients) se adaptaba bien a lo que buscaba.

HOG (Histogram of Oriented Gradients) es una técnica que nos permite extraer información sobre la forma de los objetos en una imagen. Lo que hace básicamente es dividir la imagen en partes pequeñas y calcular en cada una la **dirección de los bordes** (es decir, las orientaciones de los gradientes). Luego junta toda esa información en un único vector que podemos usar para entrenar modelos de clasificación.

Definimos la función siguiente, que haciendo uso de la función `hog` de `skimage`, nos permite extraer el vector HOG de todas las imágenes.

```
In [12]: def extraer_hog(imagenes):
    """
    Extrae características HOG de las imágenes.

    Args:
        imagenes (ndarray): Imágenes de entrada.

    Returns:
        list: Lista de características HOG extraídas.
    """
    caracteristicas_hog = []
    for i in range(len(imagenes)):
        hog_features = hog(imagenes[i], orientations=8, pixels_per_cell=(16, 16),
                           cells_per_block=(1, 1), visualize=False, channel_axis=-1)
        caracteristicas_hog.append(hog_features)

    return np.array(caracteristicas_hog)
```

- **Aclaración:**

Funcionamiento de la función `hog` según la documentación de `skimage`:

Algorithm overview:

Compute a Histogram of Oriented Gradients (HOG) by

- (optional) global image normalisation
- computing the gradient image in x and y
- computing gradient histograms
- normalising across blocks
- flattening into a feature vector

Explicación de los parámetros de la función `hog`:

- `image` : imagen de entrada
- `orientations` : número de orientaciones a considerar
- `pixels_per_cell` : tamaño de la celda en píxeles
- `cells_per_block` : número de celdas por bloque
- `visualize` : si se quiere visualizar la imagen HOG

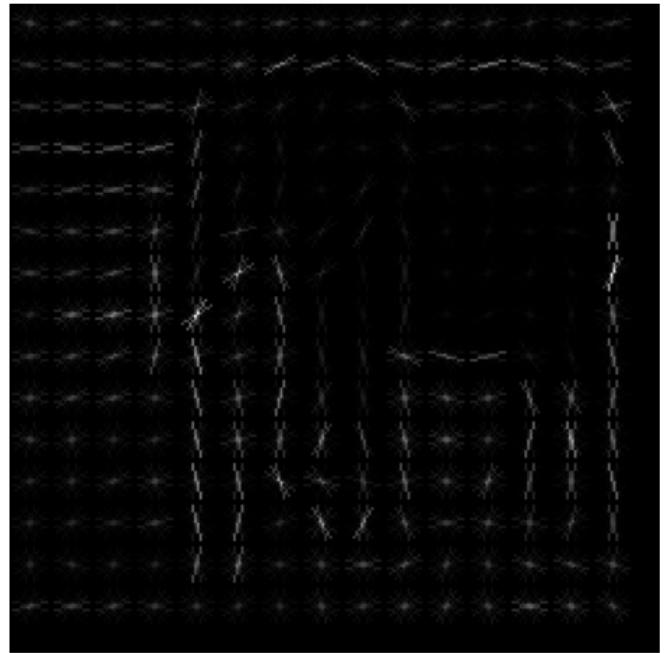
```
In [41]: # Probamos la función hog solo con una imagen
idx = np.random.randint(0, len(imagenes))
hog_features = hog(imagenes[idx], orientations=8, pixels_per_cell=(16, 16),
                   cells_per_block=(1, 1), visualize=True, channel_axis=-1)

## Visualizamos la imagen original y la imagen HOG
fig, ax = plt.subplots(1, 2, figsize=(10, 5))
ax[0].imshow(imagenes[idx])
ax[0].set_title("Imagen original")
ax[0].axis("off")
ax[1].imshow(hog_features[1], cmap="gray")
ax[1].set_title("Imagen HOG")
ax[1].axis("off")
plt.tight_layout()
plt.show()
```

Imagen original



Imagen HOG



```
In [42]: características_hog = extraer_hog(imágenes)
print(f"Dimensiones de las características HOG: {características_hog.shape}")
```

Dimensiones de las características HOG: (243, 1800)

- **Conclusión:**

El tamaño del vector de características dependerá de los parámetros `orientations`, `pixels_per_cell` y `cells_per_block`. En este caso, se ha optado por un número de orientaciones de 8, un tamaño de celda de 16x16 píxeles y un número de celdas por bloque de 1x1. Esto nos dará un vector de características de tamaño 1800, ya que la imagen de entrada tiene un tamaño de 250x250 píxeles (225 celdas x 8 orientaciones).

Dividimos en entrenamiento y test, igual que en los modelos anteriores.

```
In [43]: x_train_hog, x_test_hog, y_train_hog, y_test_hog = train_test_split(características_hog, etiquetas, test_size=0
print(f"Dimensiones de x_train_hog: {x_train_hog.shape}")
print(f"Dimensiones de x_test_hog: {x_test_hog.shape}")
print(f"Dimensiones de y_train_hog: {y_train_hog.shape}")
print(f"Dimensiones de y_test_hog: {y_test_hog.shape}")
```

Dimensiones de x_train_hog: (218, 1800)
 Dimensiones de x_test_hog: (25, 1800)
 Dimensiones de y_train_hog: (218,)
 Dimensiones de y_test_hog: (25,)

1. MLP

Modelo de red neuronal simple, construido igual que el anterior.

```
In [44]: ## Creamos el modelo
modelo_mlp_hog = modelo_simple(input_shape=(características_hog.shape[1],))
modelo_mlp_hog.summary()

## Entrenamos el modelo
history_mlp_hog = modelo_mlp_hog.fit(x_train_hog, y_train_hog, epochs=100, batch_size=32, validation_split=0.2)
```

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 1800)	0
dense (Dense)	(None, 64)	115,264
dense_1 (Dense)	(None, 32)	2,080
dense_2 (Dense)	(None, 3)	99

Total params: 117,443 (458.76 KB)

Trainable params: 117,443 (458.76 KB)

Non-trainable params: 0 (0.00 B)

Epoch 1/100
6/6 1s 35ms/step - accuracy: 0.4078 - loss: 1.1278 - val_accuracy: 0.4545 - val_loss: 1.029

Epoch 2/100
6/6 0s 12ms/step - accuracy: 0.5483 - loss: 0.9785 - val_accuracy: 0.4545 - val_loss: 1.054

Epoch 3/100
6/6 0s 11ms/step - accuracy: 0.5497 - loss: 0.9214 - val_accuracy: 0.6136 - val_loss: 0.992

Epoch 4/100
6/6 0s 12ms/step - accuracy: 0.6550 - loss: 0.9136 - val_accuracy: 0.4545 - val_loss: 1.003

Epoch 5/100
6/6 0s 10ms/step - accuracy: 0.5691 - loss: 0.8687 - val_accuracy: 0.5455 - val_loss: 0.889

Epoch 6/100
6/6 0s 11ms/step - accuracy: 0.6127 - loss: 0.8225 - val_accuracy: 0.4318 - val_loss: 0.954

Epoch 7/100
6/6 0s 11ms/step - accuracy: 0.5896 - loss: 0.8439 - val_accuracy: 0.4773 - val_loss: 1.024

Epoch 8/100
6/6 0s 11ms/step - accuracy: 0.6232 - loss: 0.8273 - val_accuracy: 0.4318 - val_loss: 0.916

Epoch 9/100
6/6 0s 11ms/step - accuracy: 0.6858 - loss: 0.7176 - val_accuracy: 0.5682 - val_loss: 0.852

Epoch 10/100
6/6 0s 12ms/step - accuracy: 0.7105 - loss: 0.6780 - val_accuracy: 0.5227 - val_loss: 0.856

Epoch 11/100
6/6 0s 12ms/step - accuracy: 0.7391 - loss: 0.6840 - val_accuracy: 0.6136 - val_loss: 0.847

Epoch 12/100
6/6 0s 13ms/step - accuracy: 0.7569 - loss: 0.6898 - val_accuracy: 0.5000 - val_loss: 0.910

Epoch 13/100
6/6 0s 12ms/step - accuracy: 0.7053 - loss: 0.6545 - val_accuracy: 0.7727 - val_loss: 0.764

Epoch 14/100
6/6 0s 12ms/step - accuracy: 0.7605 - loss: 0.6155 - val_accuracy: 0.5455 - val_loss: 0.916

Epoch 15/100
6/6 0s 11ms/step - accuracy: 0.7796 - loss: 0.6562 - val_accuracy: 0.4773 - val_loss: 1.042

Epoch 16/100
6/6 0s 11ms/step - accuracy: 0.6812 - loss: 0.6963 - val_accuracy: 0.5909 - val_loss: 0.806

Epoch 17/100
6/6 0s 16ms/step - accuracy: 0.8288 - loss: 0.5411 - val_accuracy: 0.5909 - val_loss: 0.868

Epoch 18/100
6/6 0s 12ms/step - accuracy: 0.7104 - loss: 0.5889 - val_accuracy: 0.7500 - val_loss: 0.730

Epoch 19/100
6/6 0s 12ms/step - accuracy: 0.7937 - loss: 0.5431 - val_accuracy: 0.5909 - val_loss: 0.794

Epoch 20/100
6/6 0s 12ms/step - accuracy: 0.8187 - loss: 0.4922 - val_accuracy: 0.7273 - val_loss: 0.704

Epoch 21/100
6/6 0s 13ms/step - accuracy: 0.8233 - loss: 0.4669 - val_accuracy: 0.5909 - val_loss: 0.823

Epoch 22/100
6/6 1s 260ms/step - accuracy: 0.7976 - loss: 0.4855 - val_accuracy: 0.7955 - val_loss: 0.6911

Epoch 23/100
6/6 3s 338ms/step - accuracy: 0.8761 - loss: 0.4207 - val_accuracy: 0.5227 - val_loss: 0.9031

Epoch 24/100
6/6 0s 12ms/step - accuracy: 0.7264 - loss: 0.5447 - val_accuracy: 0.6591 - val_loss: 0.7416

Epoch 25/100
6/6 0s 13ms/step - accuracy: 0.7553 - loss: 0.5314 - val_accuracy: 0.6591 - val_loss: 0.7343

Epoch 26/100
6/6 0s 11ms/step - accuracy: 0.8249 - loss: 0.4080 - val_accuracy: 0.6136 - val_loss: 0.7970

Epoch 27/100
6/6 0s 11ms/step - accuracy: 0.9036 - loss: 0.3594 - val_accuracy: 0.6818 - val_loss: 0.7090

Epoch 28/100
6/6 0s 12ms/step - accuracy: 0.8997 - loss: 0.3990 - val_accuracy: 0.6591 - val_loss: 0.753

5
Epoch 29/100
6/6 0s 12ms/step - accuracy: 0.9430 - loss: 0.3014 - val_accuracy: 0.5455 - val_loss: 0.834
0
Epoch 30/100
6/6 0s 11ms/step - accuracy: 0.8823 - loss: 0.3661 - val_accuracy: 0.7045 - val_loss: 0.722
7
Epoch 31/100
6/6 0s 11ms/step - accuracy: 0.9111 - loss: 0.3465 - val_accuracy: 0.7045 - val_loss: 0.799
1
Epoch 32/100
6/6 0s 14ms/step - accuracy: 0.8998 - loss: 0.3384 - val_accuracy: 0.5909 - val_loss: 0.803
2
Epoch 33/100
6/6 0s 16ms/step - accuracy: 0.8718 - loss: 0.3671 - val_accuracy: 0.6591 - val_loss: 0.763
3
Epoch 34/100
6/6 0s 12ms/step - accuracy: 0.9050 - loss: 0.2885 - val_accuracy: 0.7045 - val_loss: 0.692
1
Epoch 35/100
6/6 1s 246ms/step - accuracy: 0.9235 - loss: 0.2622 - val_accuracy: 0.8182 - val_loss: 0.6718
18
Epoch 36/100
6/6 2s 312ms/step - accuracy: 0.9628 - loss: 0.2618 - val_accuracy: 0.6818 - val_loss: 0.6860
60
Epoch 37/100
6/6 0s 11ms/step - accuracy: 0.9593 - loss: 0.2317 - val_accuracy: 0.7500 - val_loss: 0.6969
9
Epoch 38/100
6/6 0s 14ms/step - accuracy: 0.9680 - loss: 0.2070 - val_accuracy: 0.7727 - val_loss: 0.6659
9
Epoch 39/100
6/6 0s 12ms/step - accuracy: 0.9430 - loss: 0.2527 - val_accuracy: 0.7273 - val_loss: 0.6957
7
Epoch 40/100
6/6 0s 10ms/step - accuracy: 0.9461 - loss: 0.2446 - val_accuracy: 0.5455 - val_loss: 0.9436
6
Epoch 41/100
6/6 0s 12ms/step - accuracy: 0.8911 - loss: 0.3126 - val_accuracy: 0.7045 - val_loss: 0.7252
2
Epoch 42/100
6/6 0s 12ms/step - accuracy: 0.9457 - loss: 0.2248 - val_accuracy: 0.6136 - val_loss: 0.7893
3
Epoch 43/100
6/6 0s 12ms/step - accuracy: 0.9508 - loss: 0.1833 - val_accuracy: 0.8182 - val_loss: 0.6611
1
Epoch 44/100
6/6 0s 11ms/step - accuracy: 0.9752 - loss: 0.1573 - val_accuracy: 0.7500 - val_loss: 0.6760
0
Epoch 45/100
6/6 0s 15ms/step - accuracy: 0.9730 - loss: 0.1641 - val_accuracy: 0.7727 - val_loss: 0.6620
0
Epoch 46/100
6/6 0s 11ms/step - accuracy: 0.9624 - loss: 0.1630 - val_accuracy: 0.7955 - val_loss: 0.6814
4
Epoch 47/100
6/6 0s 12ms/step - accuracy: 0.9661 - loss: 0.1513 - val_accuracy: 0.7273 - val_loss: 0.6752
2
Epoch 48/100
6/6 2s 293ms/step - accuracy: 0.9876 - loss: 0.1344 - val_accuracy: 0.6818 - val_loss: 0.7036
36
Epoch 49/100
6/6 2s 332ms/step - accuracy: 0.9791 - loss: 0.1386 - val_accuracy: 0.7273 - val_loss: 0.7664
64
Epoch 50/100
6/6 0s 11ms/step - accuracy: 0.9277 - loss: 0.2114 - val_accuracy: 0.6591 - val_loss: 0.8415
5
Epoch 51/100
6/6 0s 12ms/step - accuracy: 0.9772 - loss: 0.1829 - val_accuracy: 0.6591 - val_loss: 0.7583
3
Epoch 52/100
6/6 0s 11ms/step - accuracy: 0.9727 - loss: 0.1638 - val_accuracy: 0.7955 - val_loss: 0.6796
6
Epoch 53/100
6/6 0s 12ms/step - accuracy: 0.9864 - loss: 0.1290 - val_accuracy: 0.7500 - val_loss: 0.6820
0
Epoch 54/100
6/6 0s 13ms/step - accuracy: 0.9830 - loss: 0.1133 - val_accuracy: 0.7727 - val_loss: 0.6847
7
Epoch 55/100
6/6 0s 12ms/step - accuracy: 0.9923 - loss: 0.1054 - val_accuracy: 0.6591 - val_loss: 0.7149
9
Epoch 56/100

6/6 ━━━━━━━━ 0s 12ms/step - accuracy: 0.9922 - loss: 0.1058 - val_accuracy: 0.7955 - val_loss: 0.688
5
Epoch 57/100
6/6 ━━━━━━━━ 0s 12ms/step - accuracy: 0.9850 - loss: 0.1117 - val_accuracy: 0.7727 - val_loss: 0.677
9
Epoch 58/100
6/6 ━━━━━━━━ 0s 12ms/step - accuracy: 0.9897 - loss: 0.1126 - val_accuracy: 0.7273 - val_loss: 0.703
9
Epoch 59/100
6/6 ━━━━━━ 0s 11ms/step - accuracy: 0.9809 - loss: 0.0916 - val_accuracy: 0.7727 - val_loss: 0.702
1
Epoch 60/100
6/6 ━━━━━━ 0s 12ms/step - accuracy: 0.9754 - loss: 0.1055 - val_accuracy: 0.6364 - val_loss: 0.799
4
Epoch 61/100
6/6 ━━━━━━ 0s 13ms/step - accuracy: 0.9876 - loss: 0.1098 - val_accuracy: 0.7500 - val_loss: 0.721
1
Epoch 62/100
6/6 ━━━━━━ 0s 12ms/step - accuracy: 0.9609 - loss: 0.1492 - val_accuracy: 0.6136 - val_loss: 1.002
3
Epoch 63/100
6/6 ━━━━━━ 0s 48ms/step - accuracy: 0.9438 - loss: 0.1490 - val_accuracy: 0.6818 - val_loss: 0.772
1
Epoch 64/100
6/6 ━━━━ 2s 310ms/step - accuracy: 0.9674 - loss: 0.1337 - val_accuracy: 0.6591 - val_loss: 0.91
73
Epoch 65/100
6/6 ━━━━ 1s 40ms/step - accuracy: 0.9773 - loss: 0.1201 - val_accuracy: 0.6136 - val_loss: 0.958
7
Epoch 66/100
6/6 ━━━━ 0s 12ms/step - accuracy: 0.9298 - loss: 0.1649 - val_accuracy: 0.6818 - val_loss: 0.837
5
Epoch 67/100
6/6 ━━━━ 0s 11ms/step - accuracy: 0.9809 - loss: 0.0992 - val_accuracy: 0.6591 - val_loss: 0.735
0
Epoch 68/100
6/6 ━━━━ 0s 12ms/step - accuracy: 0.9949 - loss: 0.0664 - val_accuracy: 0.7955 - val_loss: 0.704
0
Epoch 69/100
6/6 ━━━━ 0s 12ms/step - accuracy: 0.9881 - loss: 0.0715 - val_accuracy: 0.7727 - val_loss: 0.711
3
Epoch 70/100
6/6 ━━━━ 0s 12ms/step - accuracy: 0.9932 - loss: 0.0589 - val_accuracy: 0.7500 - val_loss: 0.723
4
Epoch 71/100
6/6 ━━━━ 0s 12ms/step - accuracy: 0.9616 - loss: 0.0986 - val_accuracy: 0.6818 - val_loss: 0.765
3
Epoch 72/100
6/6 ━━━━ 0s 11ms/step - accuracy: 0.9835 - loss: 0.0727 - val_accuracy: 0.7500 - val_loss: 0.766
8
Epoch 73/100
6/6 ━━━━ 0s 10ms/step - accuracy: 0.9858 - loss: 0.0819 - val_accuracy: 0.7045 - val_loss: 0.751
4
Epoch 74/100
6/6 ━━━━ 0s 13ms/step - accuracy: 0.9845 - loss: 0.0744 - val_accuracy: 0.7500 - val_loss: 0.731
9
Epoch 75/100
6/6 ━━━━ 0s 10ms/step - accuracy: 0.9736 - loss: 0.0938 - val_accuracy: 0.6818 - val_loss: 0.893
9
Epoch 76/100
6/6 ━━━━ 0s 12ms/step - accuracy: 0.9885 - loss: 0.0969 - val_accuracy: 0.6364 - val_loss: 0.829
5
Epoch 77/100
6/6 ━━━━ 2s 344ms/step - accuracy: 0.9830 - loss: 0.0760 - val_accuracy: 0.7273 - val_loss: 0.81
39
Epoch 78/100
6/6 ━━━━ 1s 52ms/step - accuracy: 0.9975 - loss: 0.0633 - val_accuracy: 0.7500 - val_loss: 0.741
1
Epoch 79/100
6/6 ━━━━ 0s 12ms/step - accuracy: 0.9808 - loss: 0.0690 - val_accuracy: 0.7045 - val_loss: 0.805
3
Epoch 80/100
6/6 ━━━━ 0s 11ms/step - accuracy: 0.9850 - loss: 0.0659 - val_accuracy: 0.6818 - val_loss: 0.795
1
Epoch 81/100
6/6 ━━━━ 0s 11ms/step - accuracy: 0.9984 - loss: 0.0582 - val_accuracy: 0.7045 - val_loss: 0.774
0
Epoch 82/100
6/6 ━━━━ 0s 11ms/step - accuracy: 0.9882 - loss: 0.0583 - val_accuracy: 0.7955 - val_loss: 0.720
0
Epoch 83/100
6/6 ━━━━ 0s 11ms/step - accuracy: 0.9890 - loss: 0.0420 - val_accuracy: 0.6818 - val_loss: 0.793
6

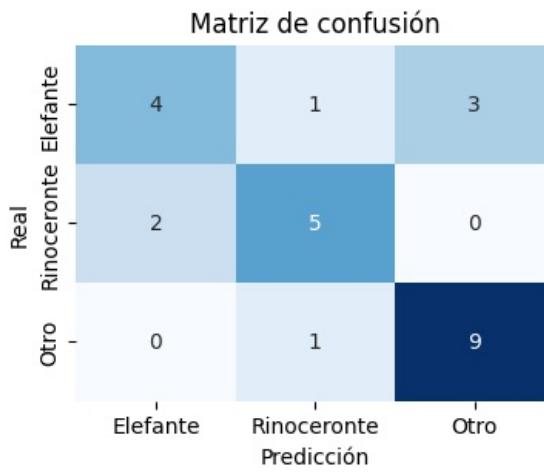
```
Epoch 84/100
6/6 ━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9938 - loss: 0.0557 - val_accuracy: 0.7273 - val_loss: 0.755
2
Epoch 85/100
6/6 ━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9856 - loss: 0.0584 - val_accuracy: 0.7727 - val_loss: 0.749
9
Epoch 86/100
6/6 ━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9923 - loss: 0.0468 - val_accuracy: 0.6364 - val_loss: 0.879
6
Epoch 87/100
6/6 ━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9910 - loss: 0.0581 - val_accuracy: 0.7500 - val_loss: 0.820
8
Epoch 88/100
6/6 ━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9912 - loss: 0.0565 - val_accuracy: 0.7500 - val_loss: 0.763
0
Epoch 89/100
6/6 ━━━━━━━━━━ 0s 15ms/step - accuracy: 0.9856 - loss: 0.0416 - val_accuracy: 0.6364 - val_loss: 0.867
1
Epoch 90/100
6/6 ━━━━━━━━━━ 0s 33ms/step - accuracy: 0.9710 - loss: 0.0810 - val_accuracy: 0.6136 - val_loss: 1.141
7
Epoch 91/100
6/6 ━━━━━━━━━━ 2s 314ms/step - accuracy: 0.9565 - loss: 0.1054 - val_accuracy: 0.7045 - val_loss: 0.86
57
Epoch 92/100
6/6 ━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9882 - loss: 0.0674 - val_accuracy: 0.7045 - val_loss: 0.809
6
Epoch 93/100
6/6 ━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9901 - loss: 0.0478 - val_accuracy: 0.7727 - val_loss: 0.751
0
Epoch 94/100
6/6 ━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9882 - loss: 0.0476 - val_accuracy: 0.7727 - val_loss: 0.750
3
Epoch 95/100
6/6 ━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9949 - loss: 0.0407 - val_accuracy: 0.7273 - val_loss: 0.753
4
Epoch 96/100
6/6 ━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9949 - loss: 0.0308 - val_accuracy: 0.7045 - val_loss: 0.772
4
Epoch 97/100
6/6 ━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9949 - loss: 0.0340 - val_accuracy: 0.7727 - val_loss: 0.759
4
Epoch 98/100
6/6 ━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9927 - loss: 0.0293 - val_accuracy: 0.6591 - val_loss: 0.904
7
Epoch 99/100
6/6 ━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9926 - loss: 0.0494 - val_accuracy: 0.6818 - val_loss: 0.822
6
Epoch 100/100
6/6 ━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9926 - loss: 0.0358 - val_accuracy: 0.7500 - val_loss: 0.788
6
```

Métricas de evaluación en el conjunto de test.

```
In [47]: ## Evaluación en el conjunto de test
score_mlp_hog = model_mlp_hog.evaluate(x_test_hog, y_test_hog)
print(f"Test accuracy: {score_mlp_hog[1]}")

## Matriz de confusión
y_pred_mlp_hog = model_mlp_hog.predict(x_test_hog)
y_pred_mlp_hog = np.argmax(y_pred_mlp_hog, axis=1)
cm_mlp_hog = confusion_matrix(y_test_hog, y_pred_mlp_hog)
confusion_matrix_plot(cm_mlp_hog)

## Informe de clasificación
print("\nInforme de clasificación:")
print("---*30")
print(classification_report(y_test_hog, y_pred_mlp_hog, target_names=["Elefante", "Rinoceronte", "Otros"], zero_1/1 ━━━━━━━━━━ 0s 26ms/step - accuracy: 0.7200 - loss: 0.9836
Test accuracy: 0.7200000286102295
1/1 ━━━━━━━━━━ 0s 24ms/step
```



Informe de clasificación:

	precision	recall	f1-score	support
Elefante	0.67	0.50	0.57	8
Rinoceronte	0.71	0.71	0.71	7
Otros	0.75	0.90	0.82	10
accuracy			0.72	25
macro avg	0.71	0.70	0.70	25
weighted avg	0.71	0.72	0.71	25

2. SVM

SVM con kernel lineal.

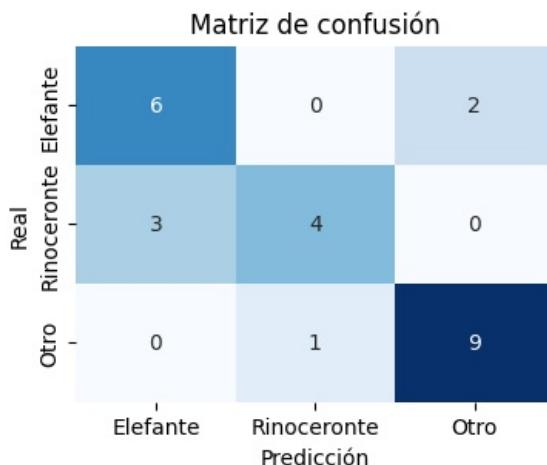
```
In [48]: ## Creamos el modelo SVM
svm_model_hog = SVC(kernel='linear', C=1.0, random_state=42)
svm_model_hog.fit(x_train_hog, y_train_hog)

## Evaluación en el conjunto de test
score_svm_hog = svm_model_hog.score(x_test_hog, y_test_hog)
print(f"Test accuracy: {score_svm_hog}")

# Matriz de confusión
cm_svm_hog = confusion_matrix(y_test_hog, svm_model_hog.predict(x_test_hog))
confusion_matrix_plot(cm_svm_hog)

# Informe de clasificación
print("\nInforme de clasificación:")
print("---*30")
print(classification_report(y_test_hog, svm_model_hog.predict(x_test_hog), target_names=["Elefante", "Rinoceronte"]))

Test accuracy: 0.76
```



Informe de clasificación:

	precision	recall	f1-score	support
Elefante	0.67	0.75	0.71	8
Rinoceronte	0.80	0.57	0.67	7
Otros	0.82	0.90	0.86	10
accuracy			0.76	25
macro avg	0.76	0.74	0.74	25
weighted avg	0.76	0.76	0.76	25

3. Random Forest

Clasificador Random Forest con 100 árboles.

```
In [50]: ## Creamos el modelo Random Forest
rf_model_hog = RandomForestClassifier(n_estimators=100, random_state=42)
```

```
## Entrenamos el modelo
rf_model_hog.fit(x_train_hog, y_train_hog)
```

```
Out[50]: ▾ RandomForestClassifier ⓘ ⓘ
RandomForestClassifier(random_state=42)
```

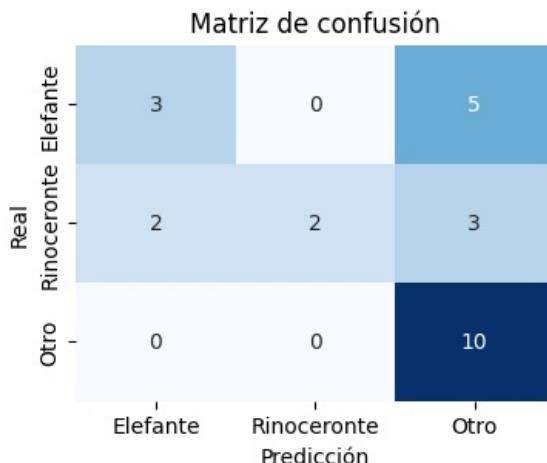
Métricas de evaluación en el conjunto de test.

```
In [51]: ## Evaluación en el conjunto de test
score_rf_hog = rf_model_hog.score(x_test_hog, y_test_hog)
print(f"Test accuracy: {score_rf_hog}")
```

```
# Matriz de confusión
cm_rf_hog = confusion_matrix(y_test_hog, rf_model_hog.predict(x_test_hog))
confusion_matrix_plot(cm_rf_hog)
```

```
## Informe de clasificación
print("\nInforme de clasificación:")
print("-"*30)
print(classification_report(y_test_hog, rf_model_hog.predict(x_test_hog), target_names=["Elefante", "Rinoceronte"]))
```

Test accuracy: 0.6



Informe de clasificación:

	precision	recall	f1-score	support
Elefante	0.60	0.38	0.46	8
Rinoceronte	1.00	0.29	0.44	7
Otros	0.56	1.00	0.71	10
accuracy			0.60	25
macro avg	0.72	0.55	0.54	25
weighted avg	0.69	0.60	0.56	25

- **Conclusión**

El uso de descriptores HOG ofrece un rendimiento similar al de la imagen aplanada en términos de precisión (entorno a 0.75), siendo algo más bajo en el caso de Random Forest.

Sin embargo, a la vista de la matriz de confusión, el modelo parece tener más problemas para diferenciar entre elefantes y rinocerontes, equivocándose en 3 ocasiones entre estas dos clases.

1.4. Modelo final

El primer paso que vamos a implementar para tratar de obtener un mejor clasificador es aumentar el tamaño del conjunto de entrenamiento. Para ello, en un primer lugar se ha accedido al dataset original, pero hemos observado que no presenta más imágenes de elefantes o rinocerontes que las ya descargadas, por lo que, finalmente, se ha optado por aplicar un aumento de datos a las imágenes de entrada.

Este aumento de datos consiste en aplicar **transformaciones a las imágenes** para crear nuevas imágenes a partir de las originales.

Con esto trataremos de evitar el sobreentrenamiento del modelo y mejorar la generalización. Las transformaciones aplicadas son:

- Rotación de la imagen (entre -45 y 45 grados)
- Traslación de la imagen (de 10 píxeles o menos)
- Cambio de brillo aleatorio (ajuste gamma entre 0.5 y 1.5)

Sería posible aplicar más transformaciones o combinar las aplicadas, pero hemos considerado experimentar sin un aumento excesivo del coste computacional y no crear un conjunto excesivamente grande.

```
In [13]: def data_augmentation(imagenes, mascaras, etiquetas):
    """
    Aplica aumento de datos a imágenes, sus máscaras correspondientes y etiquetas.
    Genera versiones modificadas de cada imagen mediante transformaciones aleatorias.

    Args:
        imagenes (ndarray): Conjunto de imágenes de entrada.
        mascaras (ndarray): Máscaras correspondientes.
        etiquetas (ndarray): Etiquetas para cada imagen.

    Returns:
        ndarray: Conjunto aumentado de imágenes.
        ndarray: Conjunto aumentado de máscaras.
        ndarray: Conjunto aumentado de etiquetas.
    """
    imagenes_aumentadas = []
    mascaras_aumentadas = []
    etiquetas_aumentadas = []

    for img, mask, label in zip(imagenes, mascaras, etiquetas):
        # Aplicar rotación, traslación y cambio de brillo
        img_aug, mask_aug, label_aug = transformar_imagen(img, mask, label)

        # Agregar los datos aumentados
        imagenes_aumentadas.append(img_aug)
        mascaras_aumentadas.append(mask_aug)
        etiquetas_aumentadas.append(label_aug)

    return np.array(imagenes_aumentadas), np.array(mascaras_aumentadas), np.array(etiquetas_aumentadas)
```

```

## Semilla para reproducibilidad
np.random.seed(422)

for i in range(len(imagenes)):
    ## 1. Conservamos la imagen original
    imagenes_aumentadas.append(imagenes[i])
    mascaras_aumentadas.append(mascaras[i])
    etiquetas_aumentadas.append(etiquetas[i])

    ## Generamos parámetros aleatorios para este par imagen-máscara
    angulo = np.random.randint(-45, 45)           # Ángulo de rotación
    tx, ty = np.random.randint(-50, 50, size=2) # Desplazamiento en X e Y
    gamma = np.random.uniform(0.5, 1.5)          # Factor de brillo

    ## 2. Rotación (aplicamos el mismo ángulo a imagen y máscara)
    ## - Para imágenes usamos interpolación bilineal (order=1)
    ## - Para máscaras usamos interpolación por vecinos más cercanos (order=0) para mantener valores binarios
    img_rotada = skimage.transform.rotate(imagenes[i], angle=angulo, mode='constant', cval=0)
    masc_rotada = skimage.transform.rotate(mascaras[i], angle=angulo, order=0, mode='constant', cval=0)

    imagenes_aumentadas.append(img_rotada)
    mascaras_aumentadas.append(masc_rotada)
    etiquetas_aumentadas.append(etiquetas[i])

    ## 3. Traslación (mismo desplazamiento para imagen y máscara)
    transformacion = skimage.transform.AffineTransform(translation=(tx, ty))
    img_traslada = skimage.transform.warp(imagenes[i], transformacion, mode='constant', cval=0)
    masc_traslada = skimage.transform.warp(mascaras[i], transformacion, order=0, mode='constant', cval=0)

    imagenes_aumentadas.append(img_traslada)
    mascaras_aumentadas.append(masc_traslada)
    etiquetas_aumentadas.append(etiquetas[i])

    ## 4. Ajuste de brillo (solo aplicamos a la imagen, no a la máscara)
    img_brillo = skimage.exposure.adjust_gamma(imagenes[i], gamma=gamma)
    imagenes_aumentadas.append(img_brillo)
    mascaras_aumentadas.append(mascaras[i]) # Máscara original
    etiquetas_aumentadas.append(etiquetas[i])

return np.array(imagenes_aumentadas), np.array(mascaras_aumentadas), np.array(etiquetas_aumentadas)

```

- **Aclaración:**

Fijamos la semilla para que el aumento sea reproducible. Las transformaciones tienen un gran factor de aleatoriedad, con esto se fija el mismo aumento de datos en caso de que se ejecute varias veces el código para que las conclusiones se mantengan

```

In [14]: ## Aplicamos data augmentation
imagenes_aumentadas, mascaras_aumentadas, etiquetas_aumentadas = data_augmentation(imagenes, mascaras, etiquetas)

## Imprimimos las dimensiones como control
print(f"Dimensiones de las imágenes aumentadas: {imagenes_aumentadas.shape}")
print(f"Dimensiones de las máscaras aumentadas: {mascaras_aumentadas.shape}")
print(f"Dimensiones de las etiquetas aumentadas: {etiquetas_aumentadas.shape}")

```

Dimensiones de las imágenes aumentadas: (972, 250, 250, 3)
Dimensiones de las máscaras aumentadas: (972, 250, 250)
Dimensiones de las etiquetas aumentadas: (972,)

- **Conclusión:**

Multiplicamos por 4 el número de imágenes para entrenar y evaluar el modelo. Esto nos permitirá obtener un modelo más robusto y que generalice mejor.

Visualizamos las 4 imágenes obtenidas a partir de la imagen original junto con la máscara para ver que el funcionamiento es correcto

```

In [15]: fig, ax = plt.subplots(4, 2, figsize=(10, 20))
for i in range(4):
    ax[i, 0].imshow(imagenes_aumentadas[i])
    ax[i, 0].set_title(f"Imagen {i+1}")
    ax[i, 0].axis("off")
    ax[i, 1].imshow(mascaras_aumentadas[i], cmap="gray")
    ax[i, 1].set_title(f"Máscara {i+1}")
    ax[i, 1].axis("off")
plt.tight_layout()
plt.show()

```

Imagen 1



Máscara 1



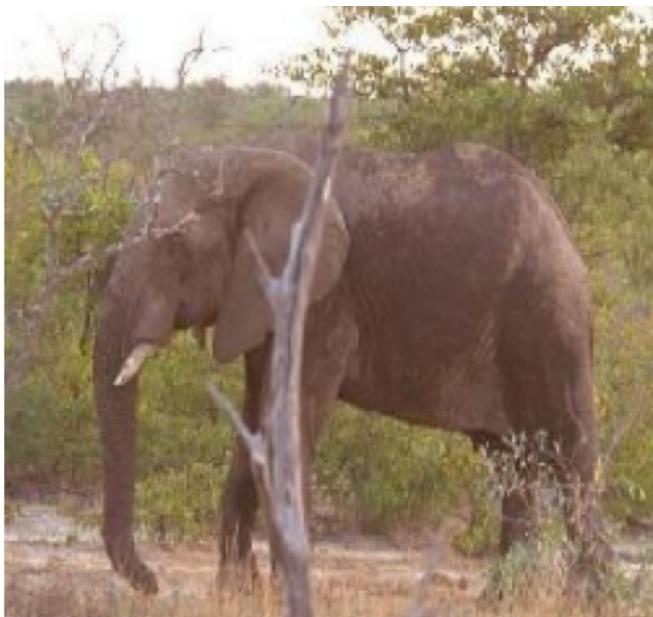
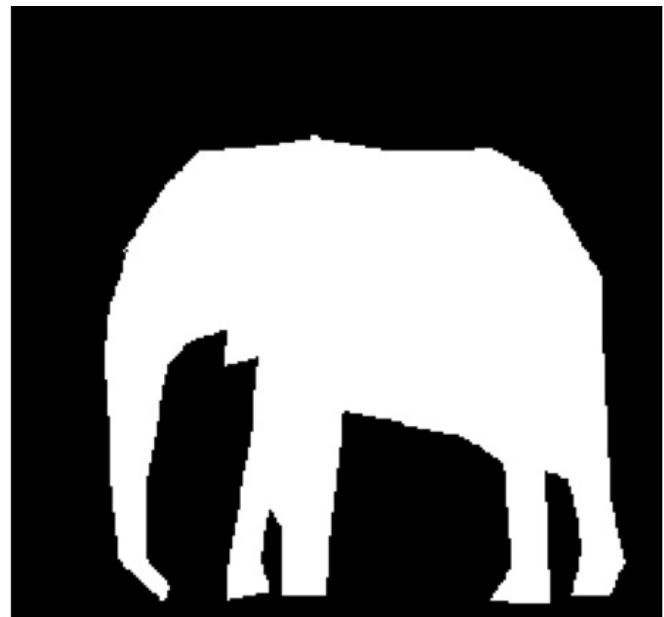


Imagen 2



Máscara 2

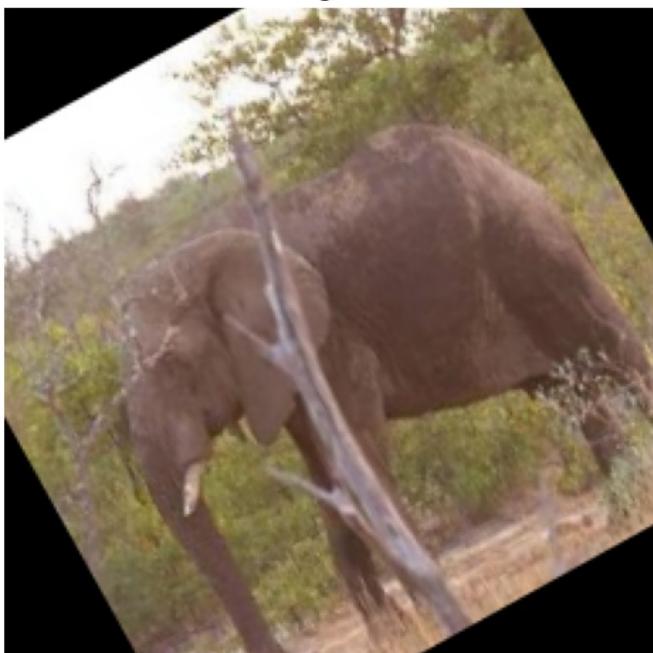
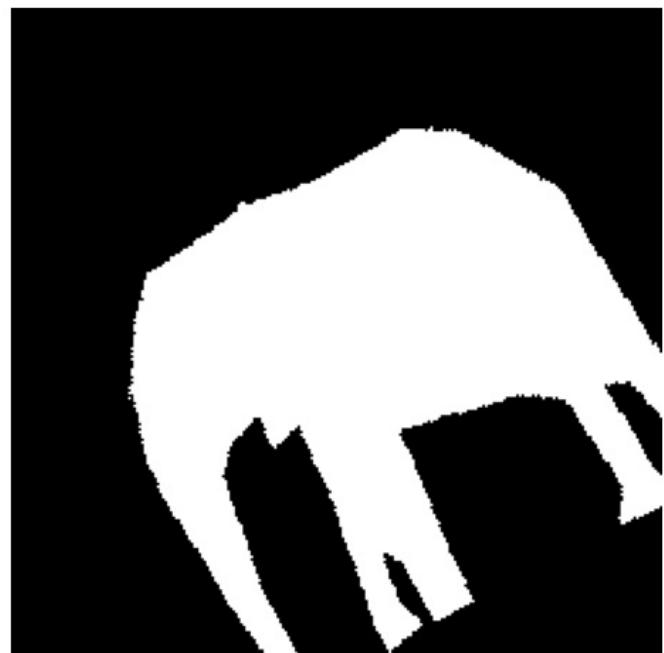


Imagen 3



Máscara 3

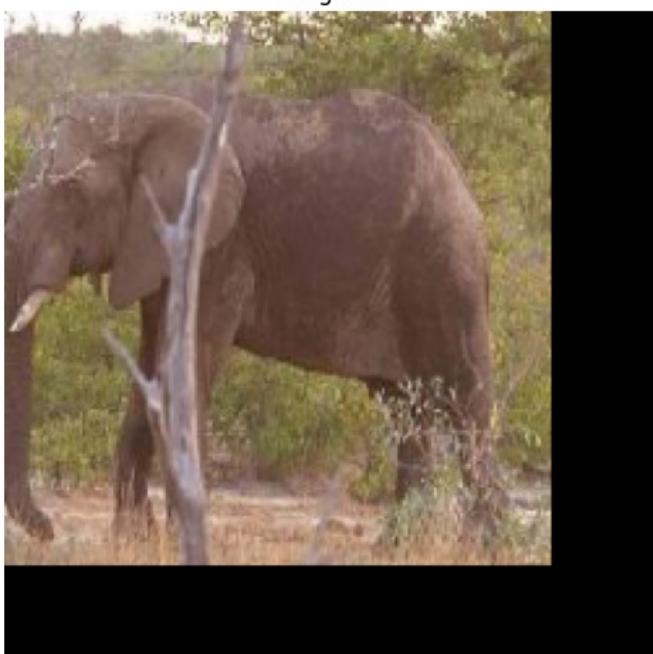
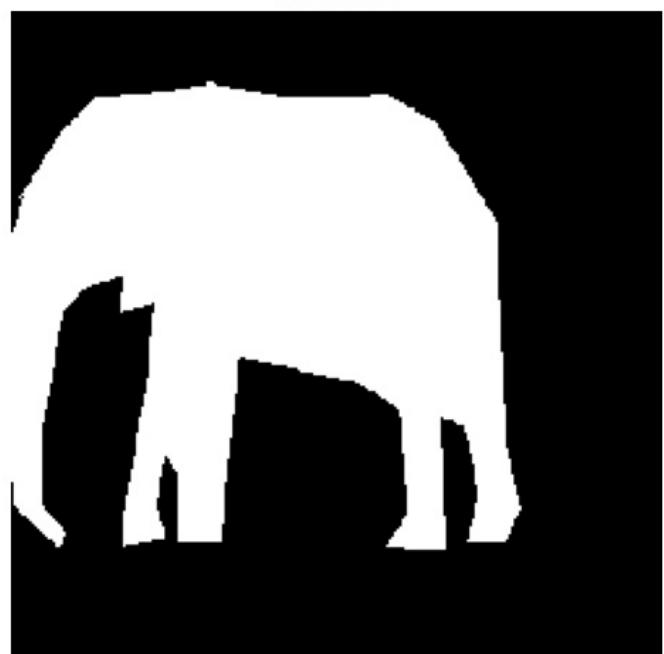
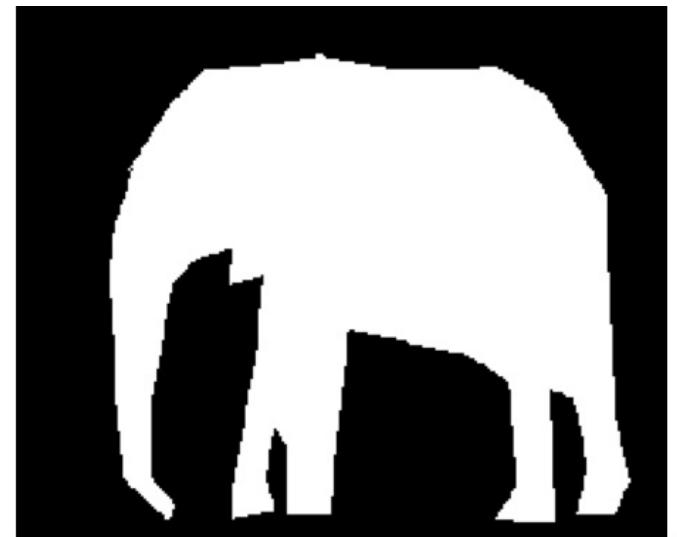


Imagen 4



Máscara 4



- **Aclaración:**

Tanto en el caso de la rotación como en la translación, en caso de que la imagen resultante "se salga de los bordes", se rellenará con color negro.

A continuación, se utilizarán las máscaras para recortar las imágenes (conjunto aplanado) y quedarnos solo con la figura del animal (utilizando la función definida anteriormente)

```
In [16]: ## Hacemos uso de la función recortar_imagen para recortar las imágenes aumentadas
imagenes_aumentadas_recortadas = []
for i in range(len(imagenes_aumentadas)):
    imagen_recortada = recortar_imagen(imagenes_aumentadas[i], mascaras_aumentadas[i])
    imagenes_aumentadas_recortadas.append(imagen_recortada)

imagenes_aumentadas_recortadas = np.array(imagenes_aumentadas_recortadas)
```

Ahora toca elegir qué utilizaremos como características de entrada de nuestro modelo

En la exploración previa vimos que el uso de características HOG era bueno pero no superior a la imagen aplanada. Sin embargo, el uso de HOG ofrece un **método que hace mayor uso de las técnicas de procesamiento de imagen** vistas en clase y es más interesante. El caso de las imágenes aplanadas, no requiere de técnicas avanzadas de procesamiento sino que aprovecha las técnicas de aprendizaje supervisado para aprender "a lo bruto".

Además, con el uso de HOG podremos experimentar más parámetros y parece que se puede obtener un rendimiento óptimo.

Utilizaremos entonces este algoritmo para construir nuestros vectores.

Inicialmente probamos a extraer las características HOG con los mismos parámetros que en la exploración previa. Esto nos dará un vector de características de tamaño 1800.

```
In [165...]: características_hog_aumentadas = extraer_hog(imagenes_aumentadas_recortadas)
print(f"Dimensiones de las características HOG aumentadas: {características_hog_aumentadas.shape}")

Dimensiones de las características HOG aumentadas: (972, 1800)
```

- **Conclusión:**

Finalmente, al utilizar este modelo, la transformación de brillo no tiene un efecto considerable, ya que la función gamma desplaza los pixeles por igual. De todas formas, se han mantenido estas imágenes en el conjunto por si se quiere emplear en otro modelo.

Dividimos nuestros conjuntos de entrenamiento y evaluación

```
In [166...]: x_train_hog_aumentadas, x_test_hog_aumentadas, y_train_hog_aumentadas, y_test_hog_aumentadas = train_test_split
print(f"Dimensiones de x_train_hog_aumentadas: {x_train_hog_aumentadas.shape}")
print(f"Dimensiones de x_test_hog_aumentadas: {x_test_hog_aumentadas.shape}")

Dimensiones de x_train_hog_aumentadas: (874, 1800)
Dimensiones de x_test_hog_aumentadas: (98, 1800)
```

La siguiente decisión es la del modelo clasificador a emplear (SVM, Random Forest o red neuronal). En este caso, nos hemos decantado por el modelo SVM, por su simplicidad y velocidad de entrenamiento, ya que por lo visto anteriormente, el rendimiento es bueno.

Entrenamos un modelo simple SVM con kernel lineal y obtenemos las métricas de evaluación.

```
In [167]: ## Creamos el modelo
svm_model_hog_aumentadas = SVC(kernel='linear', C=1.0, random_state=42)

## Entrenamos el modelo SVM
svm_model_hog_aumentadas.fit(x_train_hog_aumentadas, y_train_hog_aumentadas)
```

```
Out[167]: SVC
SVC(kernel='linear', random_state=42)
```

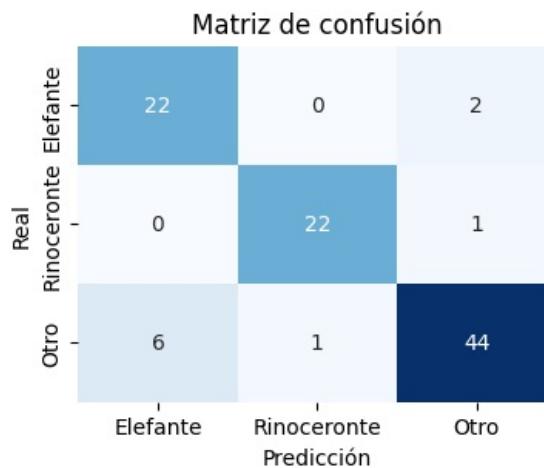
Evaluación en el conjunto de test, precisión, matriz de confusión y informe de clasificación.

```
In [168]: score_svm_hog_aumentadas = svm_model_hog_aumentadas.score(x_test_hog_aumentadas, y_test_hog_aumentadas)
print(f"Test accuracy: {score_svm_hog_aumentadas}")

# Matriz de confusión
cm_svm_hog_aumentadas = confusion_matrix(y_test_hog_aumentadas, svm_model_hog_aumentadas.predict(x_test_hog_aumentadas))
confusion_matrix_plot(cm_svm_hog_aumentadas)

# Informe de clasificación
print("\nInforme de clasificación:")
print("----*30")
print(classification_report(y_test_hog_aumentadas, svm_model_hog_aumentadas.predict(x_test_hog_aumentadas), target_names=['Elefante', 'Rinoceronte', 'Otro']))
```

Test accuracy: 0.8979591836734694



Informe de clasificación:

	precision	recall	f1-score	support
Elefante	0.79	0.92	0.85	24
Rinoceronte	0.96	0.96	0.96	23
Otros	0.94	0.86	0.90	51
accuracy			0.90	98
macro avg	0.89	0.91	0.90	98
weighted avg	0.90	0.90	0.90	98

- **Conclusión:**

Solamente utilizando el conjunto de datos aumentado, aplicando la misma función de extracción de características y un modelo SVM con kernel lineal, se obtienen unos resultados excelentes. La precisión en el conjunto de test (ahora de 90 imágenes) es de 0.9, lo que indica que el modelo es capaz de aprender patrones generales y generaliza bien a las imágenes de test.

Si nos fijamos más detalladamente, el modelo clasifica correctamente el 79% de elefantes y el 96% de los rinocerontes.

Más en detalle, no existe ningún caso en el cual un rinoceronte es clasificado como elefante ni viceversa. Los errores se corresponden con clasificar como la clase "otros", que obviamente, es un problema a tratar de evitar pero que es mejor que una confusión entre las clases de interés.

Como última experimentación, probaremos a modificar parámetros en la función HOG para ver si obtenemos una mejor clasificación.

```
In [17]: def extraer_hog(imagenes, orientaciones=8, pixels_por_celda=(16, 16), celdas_por_bloque=(1, 1)):
    """
    Extrae características HOG de las imágenes.

    Args:
        imagenes (ndarray): Imágenes de entrada.
        orientaciones (int): Número de orientaciones para HOG.
    """

    # Resto del código...
```

```
pixels_por_celda (tuple): Tamaño de la celda en píxeles.  
celdas_por_bloque (tuple): Número de celdas por bloque.
```

```
Returns:  
    list: Lista de características HOG extraídas.  
    ...  
    características_hog = []  
    for i in range(len(imágenes)):  
        hog_features = hog(imágenes[i], orientations=orientaciones, pixels_per_cell=pixels_por_celda,  
                            cells_per_block=celdas_por_bloque, visualize=False, channel_axis=-1)  
        características_hog.append(hog_features)  
  
    return np.array(características_hog)
```

Probamos los siguientes parámetros:

- `orientations`: 16, 32
- `pixels_per_cell`: 8x8, 16x16
- `cells_per_block`: 1x1, 2x2

```
In [170...]  
orientaciones = [16, 32]  
pixels_por_celda = [(8, 8), (16, 16)]  
celdas_por_bloque = [(1, 1), (2, 2)]  
resultados = []  
for orientación in orientaciones:  
    for pixels in pixels_por_celda:  
        for celdas in celdas_por_bloque:  
            print(f"Entrenando SVM con HOG: orientaciones={orientación}, pixels_por_celda={pixels}, celdas_por_bloque={celdas}")  
            características_hog = extraer_hog(imágenes_aumentadas_recortadas, orientaciones=orientación, pixels=pixels, celdas=celdas)  
            x_train_hog_aumentadas, x_test_hog_aumentadas, y_train_hog_aumentadas, y_test_hog_aumentadas = train_test_split(características_hog, y, test_size=0.2, random_state=42)  
            svm_model_hog_aumentadas = SVC(kernel='linear', C=1.0, random_state=42)  
            svm_model_hog_aumentadas.fit(x_train_hog_aumentadas, y_train_hog_aumentadas)  
            score_svm_hog_aumentadas = svm_model_hog_aumentadas.score(x_test_hog_aumentadas, y_test_hog_aumentadas)  
            resultados.append((orientación, pixels, celdas, score_svm_hog_aumentadas))
```

Entrenando SVM con HOG: orientaciones=16, pixels_por_celda=(8, 8), celdas_por_bloque=(1, 1)
Entrenando SVM con HOG: orientaciones=16, pixels_por_celda=(8, 8), celdas_por_bloque=(2, 2)
Entrenando SVM con HOG: orientaciones=16, pixels_por_celda=(16, 16), celdas_por_bloque=(1, 1)
Entrenando SVM con HOG: orientaciones=16, pixels_por_celda=(16, 16), celdas_por_bloque=(2, 2)
Entrenando SVM con HOG: orientaciones=32, pixels_por_celda=(8, 8), celdas_por_bloque=(1, 1)
Entrenando SVM con HOG: orientaciones=32, pixels_por_celda=(8, 8), celdas_por_bloque=(2, 2)
Entrenando SVM con HOG: orientaciones=32, pixels_por_celda=(16, 16), celdas_por_bloque=(1, 1)
Entrenando SVM con HOG: orientaciones=32, pixels_por_celda=(16, 16), celdas_por_bloque=(2, 2)

Presentamos la precisión obtenida en el conjunto de test para cada combinación de parámetros.

```
In [174...]  
resultados_df = pd.DataFrame(resultados, columns=["Orientaciones", "Pixels por celda", "Celdas por bloque", "Precisión"])  
resultados_df = resultados_df.sort_values(by="Precisión", ascending=False)  
print(resultados_df)  
  
Orientaciones Pixels por celda Celdas por bloque Precisión  
2 16 (16, 16) (1, 1) 0.918367  
7 32 (16, 16) (2, 2) 0.918367  
6 32 (16, 16) (1, 1) 0.918367  
5 32 (8, 8) (2, 2) 0.918367  
3 16 (16, 16) (2, 2) 0.908163  
4 32 (8, 8) (1, 1) 0.908163  
0 16 (8, 8) (1, 1) 0.897959  
1 16 (8, 8) (2, 2) 0.897959
```

• Conclusión:

Hay varias combinaciones de parámetros que ofrecen los mismos resultados, una precisión de 0.92.

Con el cambio de los parámetros la precisión se mantiene más o menos constante y no se aprecia ninguna tendencia clara creciente con una de las combinaciones. El modelo es capaz de aprender tanto con 8 orientaciones como con 16 o 32. Lo mismo pasa con el resto de parámetros.

Esta experimentación nos hace pensar que las limitaciones del modelo no están en la extracción de características, o no por lo menos en la elección de los parámetros.

▪ Nota:

Se ha probado a aumentar el número de orientaciones a 64, pero el coste computacional se ve incrementado y no se logran mejoras significativas en la precisión.

De todas formas, entrenamos el modelo con unos de los nuevos parámetros que mejoran levemente la precisión en el conjunto de test.

Modelo final:

- **Modelo SVM con kernel lineal.**
- **HOG** con 32 orientaciones, 16x16 píxeles por celda y 1x1 celdas por bloque.
- **Aumento de datos** para aumentar el tamaño del conjunto de entrenamiento (4 veces más imágenes).
- Utilización de las máscaras para **recortar las imágenes** y quedarnos solo con la figura del animal.

```
In [18]: ## Extraemos características HOG con los mejores parámetros
características_hog = extraer_hog(imágenes_aumentadas_recortadas, orientaciones=32, pixels_por_celda=(16, 16),)

## Dividimos en entrenamiento y test de forma aleatoria
x_train_hog_aumentadas, x_test_hog_aumentadas, y_train_hog_aumentadas, y_test_hog_aumentadas = train_test_split
print(f"Dimensiones de x_train_hog_aumentadas: {x_train_hog_aumentadas.shape}")
print(f"Dimensiones de x_test_hog_aumentadas: {x_test_hog_aumentadas.shape}")

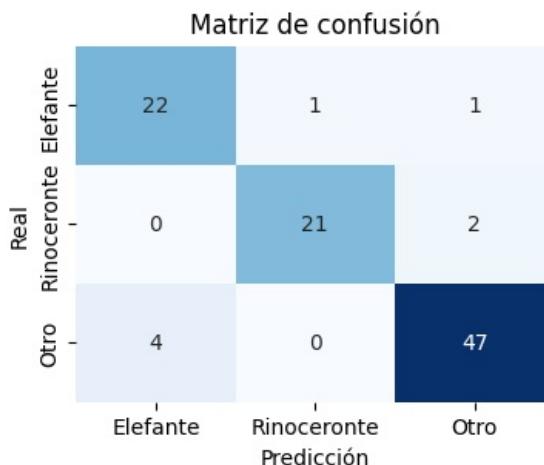
## Creamos el modelo SVM y lo entrenamos
svm_model_hog_aumentadas = SVC(kernel='linear', C=1.0, random_state=42)
svm_model_hog_aumentadas.fit(x_train_hog_aumentadas, y_train_hog_aumentadas)

## Evaluación en el conjunto de test
score_svm_hog_aumentadas = svm_model_hog_aumentadas.score(x_test_hog_aumentadas, y_test_hog_aumentadas)
print(f"Test accuracy: {score_svm_hog_aumentadas}")

## Matriz de confusión
cm_svm_hog_aumentadas = confusion_matrix(y_test_hog_aumentadas, svm_model_hog_aumentadas.predict(x_test_hog_aumentadas))
confusion_matrix_plot(cm_svm_hog_aumentadas)

## Informe de clasificación
print("\nInforme de clasificación:")
print("-"*30)
print(classification_report(y_test_hog_aumentadas, svm_model_hog_aumentadas.predict(x_test_hog_aumentadas), target_names=['Elefante', 'Rinoceronte', 'Otro']))

Dimensiones de x_train_hog_aumentadas: (874, 7200)
Dimensiones de x_test_hog_aumentadas: (98, 7200)
Test accuracy: 0.9183673469387755
```



Informe de clasificación:

	precision	recall	f1-score	support
Elefante	0.85	0.92	0.88	24
Rinoceronte	0.95	0.91	0.93	23
Otros	0.94	0.92	0.93	51
accuracy			0.92	98
macro avg	0.91	0.92	0.91	98
weighted avg	0.92	0.92	0.92	98

- **Conclusión:**

El modelo final es capaz de clasificar correctamente el 92% de las imágenes de test.

- El 85% de los elefantes son clasificados correctamente.
- El 95% de los rinocerontes son clasificados correctamente.
- Solo hay 1 elefante que se clasifique como rinoceronte y ningun rinoceronte como elefante
- Como error destacado, hay 4 emus o flamencos que son clasificados como elefantes.

Visualizamos una gran parte de las imágenes de test junto con su etiqueta real y la etiqueta predicha por el modelo, así podemos ver en qué casos el modelo se equivoca y en cuáles no.

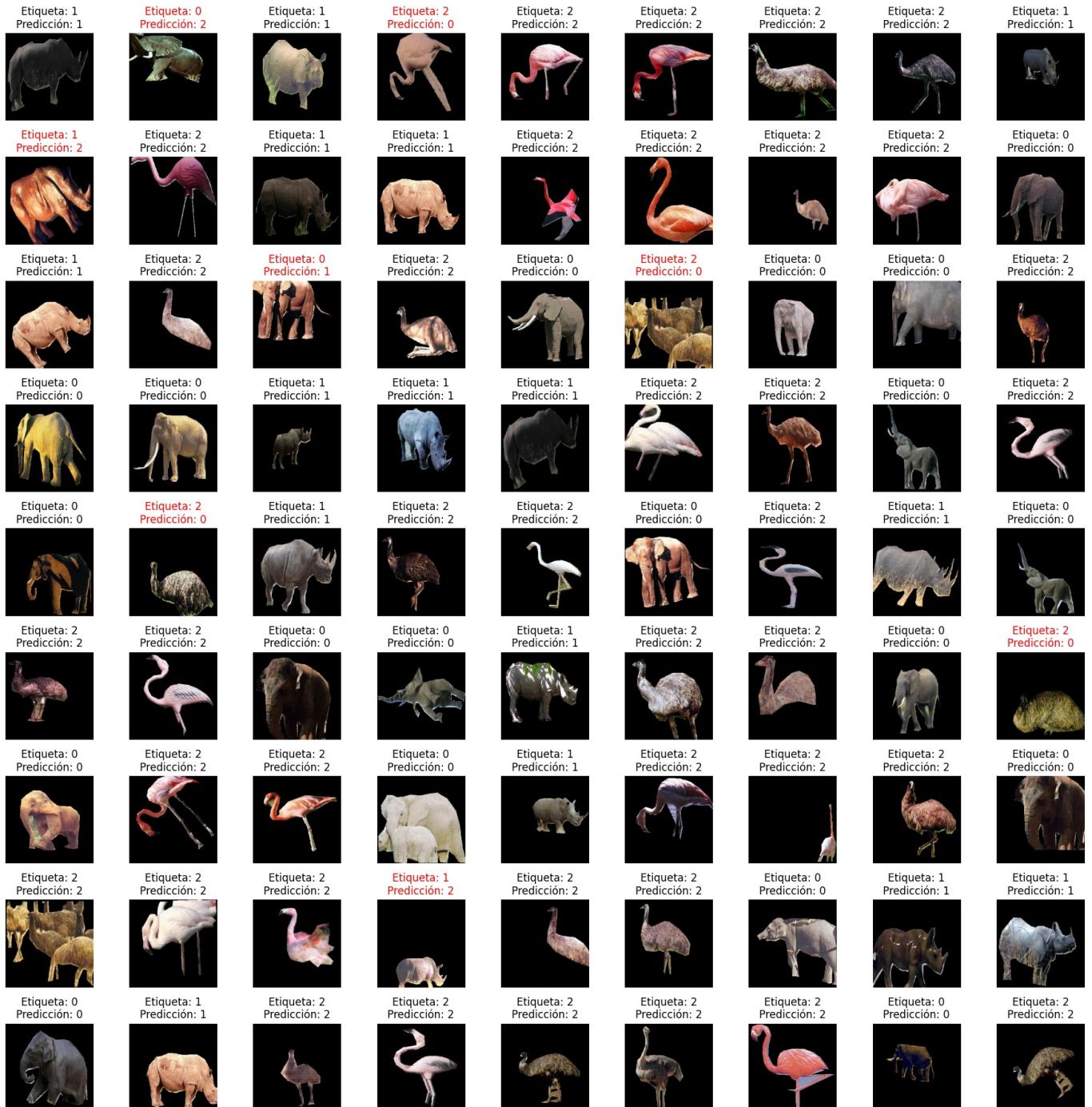
```
In [175]: ## Obtenemos los índices de test (usando el mismo random_state y test_size)
_, test_indices = train_test_split(np.arange(len(imágenes_aumentadas_recortadas)), test_size=0.1, random_state=42)

## Visualizamos las imágenes de test y sus predicciones
fig, ax = plt.subplots(9, 9, figsize=(18, 18))
```

```

for i in range(min(81, len(y_test_hog_aumentadas))):
    idx = test_indices[i]
    ax[i // 9, i % 9].imshow(imagenes_aumentadas_recortadas[idx])
    ## Texto en rojo si la predicción es incorrecta
    if y_test_hog_aumentadas[i] != svm_model_hog_aumentadas.predict([x_test_hog_aumentadas[i]])[0]:
        ax[i // 9, i % 9].set_title(f"Etiqueta: {y_test_hog_aumentadas[i]}\nPredicción: {svm_model_hog_aumentadas.predict([x_test_hog_aumentadas[i]])[0]}")
    else:
        ax[i // 9, i % 9].set_title(f"Etiqueta: {y_test_hog_aumentadas[i]}\nPredicción: {svm_model_hog_aumentadas.predict([x_test_hog_aumentadas[i]])[0]}")
    ax[i // 9, i % 9].axis("off")
plt.tight_layout()
plt.show()

```



- **Aclaración:**

Las imágenes se visualizan cuando ya se ha aplicado el aumento de datos y el recorte de la figura del animal. Las imágenes mostradas, obviamente, son diferentes a las originales.

Extra: Breve prueba con otros clasificadores

Random Forest sobre los vectores anteriores.

```
In [178]: rf_model_hog_aumentadas = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model_hog_aumentadas.fit(x_train_hog_aumentadas, y_train_hog_aumentadas)
```

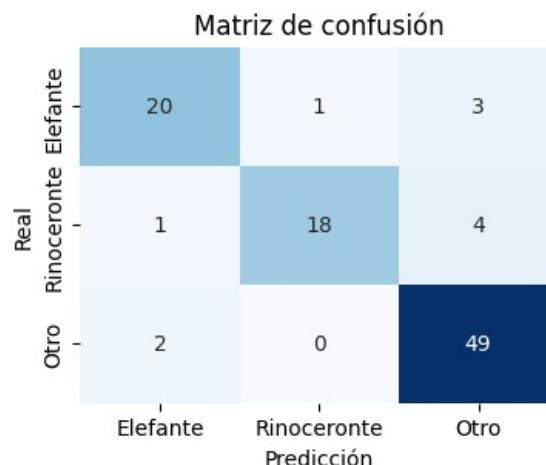
```
## Evaluación en el conjunto de test
score_rf_hog_aumentadas = rf_model_hog_aumentadas.score(x_test_hog_aumentadas, y_test_hog_aumentadas)
```

```

print(f"Test accuracy: {score_rf_hog_aumentadas}")
## Matriz de confusión
cm_rf_hog_aumentadas = confusion_matrix(y_test_hog_aumentadas, rf_model_hog_aumentadas.predict(x_test_hog_aumentadas))
confusion_matrix_plot(cm_rf_hog_aumentadas)

```

Test accuracy: 0.8877551020408163



Modelo de red neuronal convolucional.

```

In [181]: ## Creamos el modelo
model_mlp_hog_aumentadas = modelo_simple(input_shape=(caracteristicas_hog.shape[1],))

## Entrenamos el modelo
history_mlp_hog_aumentadas = model_mlp_hog_aumentadas.fit(x_train_hog_aumentadas, y_train_hog_aumentadas, epochs=100)
## Evaluación en el conjunto de test
score_mlp_hog_aumentadas = model_mlp_hog_aumentadas.evaluate(x_test_hog_aumentadas, y_test_hog_aumentadas)
print(f"Test accuracy: {score_mlp_hog_aumentadas[1]}")
## Matriz de confusión
y_pred_mlp_hog_aumentadas = model_mlp_hog_aumentadas.predict(x_test_hog_aumentadas)
y_pred_mlp_hog_aumentadas = np.argmax(y_pred_mlp_hog_aumentadas, axis=1)
cm_mlp_hog_aumentadas = confusion_matrix(y_test_hog_aumentadas, y_pred_mlp_hog_aumentadas)
confusion_matrix_plot(cm_mlp_hog_aumentadas)

```

Epoch 1/100
22/22 1s 17ms/step - accuracy: 0.5542 - loss: 0.9431 - val_accuracy: 0.7429 - val_loss: 0.6067
Epoch 2/100
22/22 0s 10ms/step - accuracy: 0.8386 - loss: 0.4268 - val_accuracy: 0.8514 - val_loss: 0.4406
Epoch 3/100
22/22 0s 8ms/step - accuracy: 0.9524 - loss: 0.2110 - val_accuracy: 0.8629 - val_loss: 0.3676
Epoch 4/100
22/22 0s 9ms/step - accuracy: 0.9849 - loss: 0.1225 - val_accuracy: 0.8800 - val_loss: 0.3570
Epoch 5/100
22/22 0s 9ms/step - accuracy: 0.9997 - loss: 0.0566 - val_accuracy: 0.8800 - val_loss: 0.3227
Epoch 6/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 0.0274 - val_accuracy: 0.8914 - val_loss: 0.3243
Epoch 7/100
22/22 0s 8ms/step - accuracy: 1.0000 - loss: 0.0164 - val_accuracy: 0.8971 - val_loss: 0.3244
Epoch 8/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 0.0102 - val_accuracy: 0.8914 - val_loss: 0.3403
Epoch 9/100
22/22 0s 8ms/step - accuracy: 1.0000 - loss: 0.0071 - val_accuracy: 0.8914 - val_loss: 0.3297
Epoch 10/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 0.0052 - val_accuracy: 0.8857 - val_loss: 0.3405
Epoch 11/100
22/22 0s 8ms/step - accuracy: 1.0000 - loss: 0.0041 - val_accuracy: 0.8857 - val_loss: 0.3440
Epoch 12/100
22/22 0s 8ms/step - accuracy: 1.0000 - loss: 0.0032 - val_accuracy: 0.8857 - val_loss: 0.3481
Epoch 13/100
22/22 0s 8ms/step - accuracy: 1.0000 - loss: 0.0025 - val_accuracy: 0.8800 - val_loss: 0.3639
Epoch 14/100
22/22 0s 8ms/step - accuracy: 1.0000 - loss: 0.0022 - val_accuracy: 0.8743 - val_loss: 0.3684

Epoch 15/100
22/22 0s 8ms/step - accuracy: 1.0000 - loss: 0.0017 - val_accuracy: 0.8857 - val_loss: 0.3646

Epoch 16/100
22/22 0s 10ms/step - accuracy: 1.0000 - loss: 0.0015 - val_accuracy: 0.8800 - val_loss: 0.3717

Epoch 17/100
22/22 0s 8ms/step - accuracy: 1.0000 - loss: 0.0013 - val_accuracy: 0.8800 - val_loss: 0.3762

Epoch 18/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 0.0010 - val_accuracy: 0.8800 - val_loss: 0.3749

Epoch 19/100
22/22 0s 10ms/step - accuracy: 1.0000 - loss: 9.5831e-04 - val_accuracy: 0.8743 - val_loss: 0.3832

Epoch 20/100
22/22 0s 11ms/step - accuracy: 1.0000 - loss: 8.0869e-04 - val_accuracy: 0.8800 - val_loss: 0.3847

Epoch 21/100
22/22 0s 14ms/step - accuracy: 1.0000 - loss: 7.3769e-04 - val_accuracy: 0.8800 - val_loss: 0.3866

Epoch 22/100
22/22 0s 14ms/step - accuracy: 1.0000 - loss: 6.5078e-04 - val_accuracy: 0.8800 - val_loss: 0.3885

Epoch 23/100
22/22 0s 12ms/step - accuracy: 1.0000 - loss: 6.1684e-04 - val_accuracy: 0.8800 - val_loss: 0.3940

Epoch 24/100
22/22 0s 13ms/step - accuracy: 1.0000 - loss: 5.1423e-04 - val_accuracy: 0.8800 - val_loss: 0.3958

Epoch 25/100
22/22 0s 13ms/step - accuracy: 1.0000 - loss: 5.0160e-04 - val_accuracy: 0.8800 - val_loss: 0.3989

Epoch 26/100
22/22 0s 12ms/step - accuracy: 1.0000 - loss: 4.1955e-04 - val_accuracy: 0.8800 - val_loss: 0.4017

Epoch 27/100
22/22 0s 12ms/step - accuracy: 1.0000 - loss: 3.8758e-04 - val_accuracy: 0.8857 - val_loss: 0.4068

Epoch 28/100
22/22 0s 10ms/step - accuracy: 1.0000 - loss: 3.6334e-04 - val_accuracy: 0.8857 - val_loss: 0.4123

Epoch 29/100
22/22 0s 10ms/step - accuracy: 1.0000 - loss: 3.1237e-04 - val_accuracy: 0.8857 - val_loss: 0.4169

Epoch 30/100
22/22 0s 10ms/step - accuracy: 1.0000 - loss: 2.7593e-04 - val_accuracy: 0.8800 - val_loss: 0.4211

Epoch 31/100
22/22 0s 10ms/step - accuracy: 1.0000 - loss: 2.6910e-04 - val_accuracy: 0.8800 - val_loss: 0.4233

Epoch 32/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 2.3378e-04 - val_accuracy: 0.8800 - val_loss: 0.4237

Epoch 33/100
22/22 0s 10ms/step - accuracy: 1.0000 - loss: 2.2055e-04 - val_accuracy: 0.8800 - val_loss: 0.4277

Epoch 34/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 2.1610e-04 - val_accuracy: 0.8800 - val_loss: 0.4307

Epoch 35/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 1.8950e-04 - val_accuracy: 0.8800 - val_loss: 0.4323

Epoch 36/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 1.7731e-04 - val_accuracy: 0.8800 - val_loss: 0.4348

Epoch 37/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 1.6707e-04 - val_accuracy: 0.8800 - val_loss: 0.4360

Epoch 38/100
22/22 0s 10ms/step - accuracy: 1.0000 - loss: 1.5938e-04 - val_accuracy: 0.8800 - val_loss: 0.4390

Epoch 39/100
22/22 0s 10ms/step - accuracy: 1.0000 - loss: 1.3584e-04 - val_accuracy: 0.8800 - val_loss: 0.4408

Epoch 40/100
22/22 0s 10ms/step - accuracy: 1.0000 - loss: 1.2263e-04 - val_accuracy: 0.8800 - val_loss: 0.4442

Epoch 41/100
22/22 0s 10ms/step - accuracy: 1.0000 - loss: 1.2563e-04 - val_accuracy: 0.8800 - val_loss: 0.4442

Epoch 42/100
22/22 0s 10ms/step - accuracy: 1.0000 - loss: 1.1265e-04 - val_accuracy: 0.8800 - val_loss:

0.4473
Epoch 43/100
22/22 0s 10ms/step - accuracy: 1.0000 - loss: 1.0904e-04 - val_accuracy: 0.8800 - val_loss: 0.4482
Epoch 44/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 1.1330e-04 - val_accuracy: 0.8800 - val_loss: 0.4499
Epoch 45/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 1.0438e-04 - val_accuracy: 0.8800 - val_loss: 0.4525
Epoch 46/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 8.7462e-05 - val_accuracy: 0.8800 - val_loss: 0.4556
Epoch 47/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 8.4170e-05 - val_accuracy: 0.8800 - val_loss: 0.4561
Epoch 48/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 8.8417e-05 - val_accuracy: 0.8800 - val_loss: 0.4595
Epoch 49/100
22/22 0s 10ms/step - accuracy: 1.0000 - loss: 7.7087e-05 - val_accuracy: 0.8800 - val_loss: 0.4591
Epoch 50/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 6.9775e-05 - val_accuracy: 0.8800 - val_loss: 0.4626
Epoch 51/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 7.5183e-05 - val_accuracy: 0.8800 - val_loss: 0.4638
Epoch 52/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 6.5784e-05 - val_accuracy: 0.8800 - val_loss: 0.4660
Epoch 53/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 6.9703e-05 - val_accuracy: 0.8800 - val_loss: 0.4678
Epoch 54/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 5.8427e-05 - val_accuracy: 0.8800 - val_loss: 0.4687
Epoch 55/100
22/22 0s 10ms/step - accuracy: 1.0000 - loss: 6.5551e-05 - val_accuracy: 0.8800 - val_loss: 0.4712
Epoch 56/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 4.9966e-05 - val_accuracy: 0.8800 - val_loss: 0.4728
Epoch 57/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 5.4538e-05 - val_accuracy: 0.8800 - val_loss: 0.4743
Epoch 58/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 5.1617e-05 - val_accuracy: 0.8800 - val_loss: 0.4759
Epoch 59/100
22/22 0s 11ms/step - accuracy: 1.0000 - loss: 4.6111e-05 - val_accuracy: 0.8800 - val_loss: 0.4768
Epoch 60/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 4.6576e-05 - val_accuracy: 0.8800 - val_loss: 0.4792
Epoch 61/100
22/22 0s 10ms/step - accuracy: 1.0000 - loss: 5.0699e-05 - val_accuracy: 0.8800 - val_loss: 0.4808
Epoch 62/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 4.2730e-05 - val_accuracy: 0.8800 - val_loss: 0.4826
Epoch 63/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 4.2970e-05 - val_accuracy: 0.8800 - val_loss: 0.4834
Epoch 64/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 3.7383e-05 - val_accuracy: 0.8800 - val_loss: 0.4847
Epoch 65/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 4.4189e-05 - val_accuracy: 0.8800 - val_loss: 0.4873
Epoch 66/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 3.6519e-05 - val_accuracy: 0.8800 - val_loss: 0.4874
Epoch 67/100
22/22 0s 10ms/step - accuracy: 1.0000 - loss: 3.6975e-05 - val_accuracy: 0.8800 - val_loss: 0.4894
Epoch 68/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 3.7958e-05 - val_accuracy: 0.8800 - val_loss: 0.4906
Epoch 69/100
22/22 0s 9ms/step - accuracy: 1.0000 - loss: 3.4148e-05 - val_accuracy: 0.8800 - val_loss: 0.4920
Epoch 70/100

22/22 ━━━━━━ 0s 9ms/step - accuracy: 1.0000 - loss: 3.1547e-05 - val_accuracy: 0.8800 - val_loss: 0.4924
Epoch 71/100
22/22 ━━━━━━ 0s 9ms/step - accuracy: 1.0000 - loss: 3.0293e-05 - val_accuracy: 0.8800 - val_loss: 0.4944
Epoch 72/100
22/22 ━━━━━━ 0s 9ms/step - accuracy: 1.0000 - loss: 3.0509e-05 - val_accuracy: 0.8800 - val_loss: 0.4960
Epoch 73/100
22/22 ━━━━━━ 0s 10ms/step - accuracy: 1.0000 - loss: 2.7873e-05 - val_accuracy: 0.8800 - val_loss: 0.4957
Epoch 74/100
22/22 ━━━━━━ 0s 9ms/step - accuracy: 1.0000 - loss: 2.7456e-05 - val_accuracy: 0.8800 - val_loss: 0.4986
Epoch 75/100
22/22 ━━━━━━ 0s 9ms/step - accuracy: 1.0000 - loss: 2.3998e-05 - val_accuracy: 0.8800 - val_loss: 0.4989
Epoch 76/100
22/22 ━━━━━━ 0s 9ms/step - accuracy: 1.0000 - loss: 2.6678e-05 - val_accuracy: 0.8800 - val_loss: 0.4997
Epoch 77/100
22/22 ━━━━━━ 0s 9ms/step - accuracy: 1.0000 - loss: 2.3770e-05 - val_accuracy: 0.8800 - val_loss: 0.5017
Epoch 78/100
22/22 ━━━━━━ 0s 10ms/step - accuracy: 1.0000 - loss: 2.1110e-05 - val_accuracy: 0.8800 - val_loss: 0.5030
Epoch 79/100
22/22 ━━━━━━ 0s 9ms/step - accuracy: 1.0000 - loss: 2.2777e-05 - val_accuracy: 0.8800 - val_loss: 0.5044
Epoch 80/100
22/22 ━━━━━━ 0s 9ms/step - accuracy: 1.0000 - loss: 1.9346e-05 - val_accuracy: 0.8800 - val_loss: 0.5058
Epoch 81/100
22/22 ━━━━━━ 0s 9ms/step - accuracy: 1.0000 - loss: 2.1810e-05 - val_accuracy: 0.8800 - val_loss: 0.5051
Epoch 82/100
22/22 ━━━━━━ 0s 9ms/step - accuracy: 1.0000 - loss: 2.1287e-05 - val_accuracy: 0.8800 - val_loss: 0.5074
Epoch 83/100
22/22 ━━━━━━ 0s 9ms/step - accuracy: 1.0000 - loss: 2.0553e-05 - val_accuracy: 0.8800 - val_loss: 0.5086
Epoch 84/100
22/22 ━━━━━━ 0s 9ms/step - accuracy: 1.0000 - loss: 2.1410e-05 - val_accuracy: 0.8800 - val_loss: 0.5100
Epoch 85/100
22/22 ━━━━━━ 0s 9ms/step - accuracy: 1.0000 - loss: 1.7508e-05 - val_accuracy: 0.8800 - val_loss: 0.5104
Epoch 86/100
22/22 ━━━━━━ 0s 9ms/step - accuracy: 1.0000 - loss: 1.6875e-05 - val_accuracy: 0.8800 - val_loss: 0.5122
Epoch 87/100
22/22 ━━━━━━ 0s 9ms/step - accuracy: 1.0000 - loss: 1.6147e-05 - val_accuracy: 0.8800 - val_loss: 0.5138
Epoch 88/100
22/22 ━━━━━━ 0s 9ms/step - accuracy: 1.0000 - loss: 1.6331e-05 - val_accuracy: 0.8800 - val_loss: 0.5145
Epoch 89/100
22/22 ━━━━━━ 0s 9ms/step - accuracy: 1.0000 - loss: 1.7432e-05 - val_accuracy: 0.8800 - val_loss: 0.5150
Epoch 90/100
22/22 ━━━━━━ 0s 8ms/step - accuracy: 1.0000 - loss: 1.5829e-05 - val_accuracy: 0.8800 - val_loss: 0.5166
Epoch 91/100
22/22 ━━━━━━ 0s 9ms/step - accuracy: 1.0000 - loss: 1.7509e-05 - val_accuracy: 0.8800 - val_loss: 0.5166
Epoch 92/100
22/22 ━━━━━━ 0s 9ms/step - accuracy: 1.0000 - loss: 1.5911e-05 - val_accuracy: 0.8800 - val_loss: 0.5185
Epoch 93/100
22/22 ━━━━━━ 0s 8ms/step - accuracy: 1.0000 - loss: 1.6094e-05 - val_accuracy: 0.8800 - val_loss: 0.5187
Epoch 94/100
22/22 ━━━━━━ 0s 9ms/step - accuracy: 1.0000 - loss: 1.4284e-05 - val_accuracy: 0.8800 - val_loss: 0.5203
Epoch 95/100
22/22 ━━━━━━ 0s 10ms/step - accuracy: 1.0000 - loss: 1.4607e-05 - val_accuracy: 0.8800 - val_loss: 0.5214
Epoch 96/100
22/22 ━━━━━━ 0s 11ms/step - accuracy: 1.0000 - loss: 1.3855e-05 - val_accuracy: 0.8800 - val_loss: 0.5219
Epoch 97/100
22/22 ━━━━━━ 0s 10ms/step - accuracy: 1.0000 - loss: 1.3305e-05 - val_accuracy: 0.8800 - val_loss: 0.5232

```
Epoch 98/100
22/22 - 0s 11ms/step - accuracy: 1.0000 - loss: 1.3029e-05 - val_accuracy: 0.8800 - val_loss: 0.5234
Epoch 99/100
22/22 - 0s 12ms/step - accuracy: 1.0000 - loss: 1.3186e-05 - val_accuracy: 0.8800 - val_loss: 0.5246
Epoch 100/100
22/22 - 0s 11ms/step - accuracy: 1.0000 - loss: 1.2652e-05 - val_accuracy: 0.8800 - val_loss: 0.5257
4/4 - 0s 14ms/step - accuracy: 0.8946 - loss: 0.9011
Test accuracy: 0.8979591727256775
4/4 - 0s 21ms/step
```

