# TFM Grupo 7: Código Fuente del Modelo SSD MobileNet V2

# Init de las librerias

!pip install tensorflow opencv-python matplotlib --quiet

!pip install roboflow

**Análisis de Contenido**: esta celda recorre la estructura de directorios del dataset especificado y muestra el contenido de cada carpeta, imprimiendo hasta 5 archivos por carpeta.

import os  
# Ruta del dataset  
dataset\_path = "/content/epi-dataset-ssd-1"  
  
# Imprimir el contenido del dataset descargado  
print("Contenido del dataset descargado:")  
for root, dirs, files in os.walk(dataset\_path):  
 print(f"{root}")  
 for file in files[:5]: # Mostrar 5 archivos por carpeta  
 print(f" {file}")

Análisis de Contenido(2) / consistencia con respecto al formato COCO Este script monta Google Drive y verifica la presencia de archivos de anotaciones COCO en las subcarpetas "train", "valid" y "test" del dataset epi dataset ssd proveniendo de la herramienta Roboflow. Imprime un mensaje indicando si las anotaciones fueron encontradas o no en cada subcarpeta.

import os  
# Montar Google Drive  
from google.colab import drive  
drive.mount('/content/drive')  
dataset\_path = "/content/epi-dataset-ssd-1"  
# Definir las rutas de imágenes y etiquetas  
train\_images\_path = os.path.join(dataset\_path, "train", "images")  
train\_annotations\_path = os.path.join(dataset\_path, "train", "annotations", "\_annotations.coco.json")  
valid\_images\_path = os.path.join(dataset\_path, "valid", "images")  
valid\_annotations\_path = os.path.join(dataset\_path, "valid", "annotations", "\_annotations.coco.json")  
# Verificar la presencia de anotaciones COCO en cada carpeta  
for subdir in ["train", "valid", "test"]:  
 annotations\_path = os.path.join(dataset\_path, subdir, "\_annotations.coco.json")  
 if os.path.exists(annotations\_path):  
 print(f"Anotaciones encontradas en {subdir}: {annotations\_path}")  
 else:  
 print(f"No se encontraron anotaciones en {subdir} ! Verifica el formato del dataset.")

Análisis de Contenido (3): Este script verifica la existencia de un archivo de anotaciones COCO y, si existe, lo carga y muestra las primeras cinco anotaciones. Si el archivo no existe, imprime un mensaje de error.

import os  
import json  
  
# Cargar el archivo COCO y mostrar algunas anotaciones  
if os.path.exists(annotations\_path):  
 with open(annotations\_path, "r") as f:  
 coco\_data = json.load(f)  
 print("Ejemplos de anotaciones del archivo COCO:")  
 for i, ann in enumerate(coco\_data["annotations"][:5]): # Mostrar las 5 primeras anotaciones  
 print(f"Anotación {i+1}: {ann}")  
else:  
 print(f"El archivo {annotations\_path} no existe ! Verifica el dataset.")

Análisis de Contenido (4): Esta función carga un archivo de anotaciones COCO, verifica la correspondencia entre los IDs de las clases y sus nombres, y muestra la distribución de anotaciones por clase. Luego, se llama a la función para verificar un archivo específico de anotaciones de entrenamiento.

import json  
def verificar\_clases\_coco(annotations\_path):  
 """  
 Verifica la correspondencia de los IDs de las clases COCO y muestra su distribución.  
 """  
 with open(annotations\_path, "r", encoding="utf-8") as f:  
 coco\_data = json.load(f)  
  
 categories = {cat["id"]: cat["name"] for cat in coco\_data["categories"]}  
 class\_counts = {}  
  
 for ann in coco\_data["annotations"]:  
 class\_id = ann.get("category\_id")  
 if class\_id in categories:  
 class\_counts[class\_id] = class\_counts.get(class\_id, 0) + 1  
 else:  
 print(f"Advertencia: Se encontró un ID de clase desconocido ({class\_id}).")  
  
 print("\nCorrespondencia de clases COCO:")  
 for class\_id, name in categories.items():  
 print(f" - ID {class\_id}: {name} ({class\_counts.get(class\_id, 0)} anotaciones)")  
  
# Ruta del archivo de anotaciones  
train\_annotations\_path = "/content/epi-dataset-ssd-1/train/\_annotations.coco.json"  
# Verificación del archivo train\_annotations\_path  
verificar\_clases\_coco(train\_annotations\_path)  
  
**Pipeline de preprocesamiento de los datos para el Entrenamiento de un Modelo SSD**

Este script prepara los datos para entrenar el modelo SSD, asegurandose de que las imágenes y anotaciones estan listas para ser utilizadas.

Carga de anotaciones COCO: El script abre el archivo JSON que contiene las anotaciones y extrae la información esencial (nombre de las imágenes, dimensiones y anotaciones de los objetos).

Filtrado de clases válidas: Solo se conservan las clases entre 1 y 5, mientras que las demás anotaciones son ignoradas.

Normalización de coordenadas de las bounding boxes: Se convierten los valores a coordenadas relativas para que sean compatibles con los modelos de IA.

Gestión de errores e inconsistencias: Se verifica la existencia de los archivos de imagen, se controlan dimensiones incorrectas y se eliminan anotaciones problemáticas, además de revisar valores anómalos en las bounding boxes después de la normalización.

Preprocesamiento de imágenes: Se cargan y redimensionan las imágenes a un tamaño estándar de 224x224 píxeles, normalizando los valores de los píxeles entre 0 y 1.

Creación de etiquetas en un formato utilizable por el modelo: Se convierten las clases a one-hot encoding y se truncan o rellenan las bounding boxes para garantizar un número fijo de anotaciones por imagen (MAX\_BOXES).

Verificación final de los datos: Se muestran estadísticas sobre las imágenes y anotaciones cargadas, asegurando la coherencia de las bounding boxes para evitar errores en el entrenamiento.

Este pipeline garantiza una calidad óptima de los datos, identificando y corrigiendo errores potenciales antes del entrenamiento del modelo. Esto permite mejorar la fiabilidad de las predicciones y optimizar el rendimiento del modelo YOLO u otra red de detección de objetos.

import numpy as np  
import json  
import os  
import cv2  
import tensorflow as tf  
  
# Definición de constantes  
IMG\_SIZE = 224  
NUM\_CLASSES = 5 # Solo se permiten 5 clases válidas  
MAX\_BOXES = 10  
def cargar\_datos\_coco(images\_path, annotations\_path):  
 # Carga el archivo JSON de anotaciones en formato COCO  
 with open(annotations\_path, "r") as f:  
 coco\_data = json.load(f)  
  
 images = []  
 labels\_class = []  
 labels\_bbox = []  
  
 # Mapeo de ID de imagen a nombres de archivo y tamaños  
 image\_id\_to\_filename = {img["id"]: img["file\_name"] for img in coco\_data["images"]}  
 image\_id\_to\_size = {img["id"]: (img["width"], img["height"]) for img in coco\_data["images"]}  
  
 # Diccionario para almacenar las anotaciones organizadas por imagen  
 labels\_dict = {img\_id: [] for img\_id in image\_id\_to\_filename.keys()}  
  
 # Contador de anotaciones eliminadas  
 eliminadas = 0  
  
 # Procesar las anotaciones  
 for annotation in coco\_data["annotations"]:  
 image\_id = annotation["image\_id"]  
 class\_id = annotation["category\_id"]  
  
 # Ignorar la clase 0 (chaleco-y-casco-ajax)  
 if class\_id == 0:  
 eliminadas += 1  
 continue  
  
 # Solo aceptar clases entre 1 y 5  
 if class\_id not in [1, 2, 3, 4, 5]:  
 continue  
  
 # Ajustar las clases para que sean de [0, 4] antes del one-hot encoding  
 class\_id -= 1 # Convierte [1,2,3,4,5] en [0,1,2,3,4]  
  
 # Extraer coordenadas de la bounding box  
 xmin, ymin, w, h = annotation["bbox"]  
 xmax = xmin + w  
 ymax = ymin + h  
  
 # Verificar si la imagen tiene dimensiones válidas  
 if image\_id not in image\_id\_to\_size:  
 print(f"Error: Dimensiones no encontradas para image\_id {image\_id}")  
 continue  
 width, height = image\_id\_to\_size[image\_id]  
  
 # Control de errores en dimensiones antes de normalizar  
 if width <= 1 or height <= 1:  
 print(f"Problema detectado: Dimensiones incorrectas (w={width}, h={height}) en imagen {image\_id}")  
  
 # Normalizar coordenadas de la bounding box (evitar división por dimensiones incorrectas)  
 if width > 1 and height > 1:  
 xmin = xmin / width  
 ymin = ymin / height  
 xmax = xmax / width  
 ymax = ymax / height  
 else:  
 print(f"Dimensiones inválidas en imagen {image\_id}: (w={width}, h={height}) - Normalización omitida")  
 xmin, ymin, xmax, ymax = 0, 0, 0, 0 # Valores por defecto para evitar errores  
  
 # Verificación después de la normalización  
 if xmax > 1 or ymax > 1 or xmin < 0 or ymin < 0:  
 print(f"Anomalía detectada: {xmin, ymin, xmax, ymax} en imagen {image\_id}")  
  
 # Agregar la anotación corregida a la lista  
 labels\_dict[image\_id].append([class\_id, xmin, ymin, xmax, ymax])  
  
 # Procesar las imágenes  
 for image\_id, img\_name in image\_id\_to\_filename.items():  
 img\_path = os.path.join(images\_path, img\_name)  
 if not os.path.exists(img\_path):  
 continue  
  
 img = cv2.imread(img\_path)  
 if img is None:  
 continue  
 # Redimensionar la imagen y normalizar valores entre 0 y 1  
 img = cv2.resize(img, (IMG\_SIZE, IMG\_SIZE)) / 255.0  
 images.append(img)  
  
 # Recuperar anotaciones asociadas (limitar a MAX\_BOXES)  
 label = labels\_dict.get(image\_id, [])[:MAX\_BOXES]  
  
 # Completar con valores nulos si es necesario  
 while len(label) < MAX\_BOXES:  
 label.append([-1, 0, 0, 0, 0]) # -1 indica líneas vacías  
  
 label = np.array(label, dtype=np.float32)  
  
 # Extraer clases y bounding boxes  
 class\_labels = label[:, 0] # Clases (incluye -1 potencialmente)  
 bbox\_labels = label[:, 1:] # Bounding boxes  
  
 # Eliminar anotaciones inválidas (-1)  
 valid\_idx = class\_labels >= 0  
 class\_labels = class\_labels[valid\_idx] # Filtrar clases válidas  
 bbox\_labels = bbox\_labels[valid\_idx] # Filtrar bounding boxes válidas  
  
 # Si la lista queda vacía, agregar una clase y bounding box vacía por defecto  
 if len(class\_labels) == 0:  
 class\_labels = np.array([0]) # Clase 0 por defecto  
 bbox\_labels = np.zeros((1, 4), dtype=np.float32) # Bounding box vacía  
  
 # Asegurar que bbox\_labels tenga siempre MAX\_BOXES líneas  
 while len(bbox\_labels) < MAX\_BOXES:  
 bbox\_labels = np.vstack([bbox\_labels, [0, 0, 0, 0]]) # Relleno  
  
 bbox\_labels = bbox\_labels[:MAX\_BOXES] # Truncar si es demasiado grande  
  
 # Convertir clases a enteros y corregir valores fuera de rango  
 class\_labels = np.round(class\_labels).astype(int)  
 class\_labels = np.clip(class\_labels, 0, NUM\_CLASSES - 1)  
  
 # Inicializar one-hot encoding  
 class\_labels\_one\_hot = np.zeros((MAX\_BOXES, NUM\_CLASSES), dtype=np.float32)  
  
 # Llenar de manera segura las clases válidas  
 for i, class\_id in enumerate(class\_labels[:MAX\_BOXES]):  
 if class\_id >= 0: # Verificar que el índice es válido  
 class\_labels\_one\_hot[i, class\_id] = 1.0  
  
 # Agregar datos procesados a las listas finales  
 labels\_class.append(class\_labels\_one\_hot)  
 labels\_bbox.append(bbox\_labels)  
  
 return (  
 np.array(images, dtype=np.float32),  
 np.array(labels\_class, dtype=np.float32),  
 np.array(labels\_bbox, dtype=np.float32)  
 )  
  
# Cargar y corregir los datos ANTES del entrenamiento  
X\_train, y\_train\_class, y\_train\_bbox = cargar\_datos\_coco(train\_images\_path, train\_annotations\_path)  
X\_valid, y\_valid\_class, y\_valid\_bbox = cargar\_datos\_coco(valid\_images\_path, valid\_annotations\_path)  
  
# Assurer que toutes les valeurs sont bien comprises entre 0 et 1  
y\_valid\_bbox = np.clip(y\_valid\_bbox, 0, 1)  
  
# Verificación profunda de las bounding boxes después de la carga  
print("\n\*\*Diagnóstico profundo de bounding boxes\*\*")  
print("\nPreprocesamiento terminado con éxito 🚀")

**Fase de aumento de datos para el entrenamiento del Modelo SSD**

Este script implementa un pipeline de aumento de datos para equilibrar el dataset antes del entrenamiento de un modelo SSD, generando más ejemplos de clases subrepresentadas como "casco" y "chaleco". Se usa ImageDataGenerator para aplicar transformaciones como rotaciones, desplazamientos y volteos, sin alterar las bounding boxes ni etiquetas. Las imágenes aumentadas se agregan al dataset original, actualizando las variables X\_train, y\_train\_class y y\_train\_bbox. Finalmente, se verifica el nuevo tamaño del dataset para asegurar que el aumento de datos se ha aplicado correctamente y mejorar la precisión del modelo.

from tensorflow.keras.preprocessing.image import ImageDataGenerator  
import numpy as np  
  
# Création du générateur d'augmentation d'images  
datagen = ImageDataGenerator(  
 rotation\_range=20,  
 width\_shift\_range=0.2,  
 height\_shift\_range=0.2,  
 horizontal\_flip=True,  
 fill\_mode="nearest"  
)  
  
# Identifier les classes sous-représentées  
classes\_to\_augment = [1, 2] # Casque et Chaleco  
augmentation\_factor = 2 # Multiplier par 2 le nombre d'exemples pour équilibrer  
  
for class\_id in classes\_to\_augment:  
 class\_indices = np.where(np.argmax(y\_train\_class, axis=-1)[:, 0] == class\_id)[0] # Sélectionner indices de la classe  
 class\_images = X\_train[class\_indices] # Extraire les images concernées  
 class\_bboxes = y\_train\_bbox[class\_indices] # Extraire les bounding boxes associées  
 class\_labels = y\_train\_class[class\_indices] # Extraire les labels associés  
  
 # Générer des images augmentées  
 augmented\_images = []  
 augmented\_bboxes = []  
 augmented\_labels = []  
  
 for img, bbox, label in zip(class\_images, class\_bboxes, class\_labels):  
 for \_ in range(augmentation\_factor): # Augmenter chaque image plusieurs fois  
 transformed\_img = datagen.random\_transform(img)  
 augmented\_images.append(transformed\_img)  
 augmented\_bboxes.append(bbox) # Les bounding boxes restent inchangées  
 augmented\_labels.append(label) # Les labels restent inchangés  
  
 # Convertir en numpy arrays  
 augmented\_images = np.array(augmented\_images)  
 augmented\_bboxes = np.array(augmented\_bboxes)  
 augmented\_labels = np.array(augmented\_labels)  
  
 # Ajouter les nouvelles images et leurs labels  
 X\_train = np.concatenate([X\_train, augmented\_images], axis=0)  
 y\_train\_class = np.concatenate([y\_train\_class, augmented\_labels], axis=0)  
 y\_train\_bbox = np.concatenate([y\_train\_bbox, augmented\_bboxes], axis=0)  
  
print(f"📌 Nouvelle taille du dataset après Data Augmentation : {X\_train.shape[0]}")  
  
**Entrenamiento del Modelo de Detección de Objetos SSD MobileNetV2**

Importamos tensorflow y numpy para manejar redes neuronales y operaciones matemáticas. Definición de Constantes: Establecemos el tamaño de las imágenes (IMG\_SIZE), el número de clases (NUM\_CLASSES) y el máximo de objetos por imagen (MAX\_BOXES). Definición del Modelo:

Utilizamos MobileNetV2 como modelo base, ajustado para la detección de objetos. Añadimos capas adicionales para manejar las salidas de clases y bounding boxes. Compilación del Modelo: Compilamos el modelo con funciones de pérdida adecuadas para clasificación y regresión de bounding boxes. Configuramos un planificador de tasa de aprendizaje para ajustar la tasa durante el entrenamiento. Verificaciones Previas:

Verificamos las dimensiones de los datos de entrenamiento y validación para asegurarnos de que coinciden con lo esperado. Comprobamos los índices de clase y los valores de los bounding boxes. Entrenamiento del Modelo:

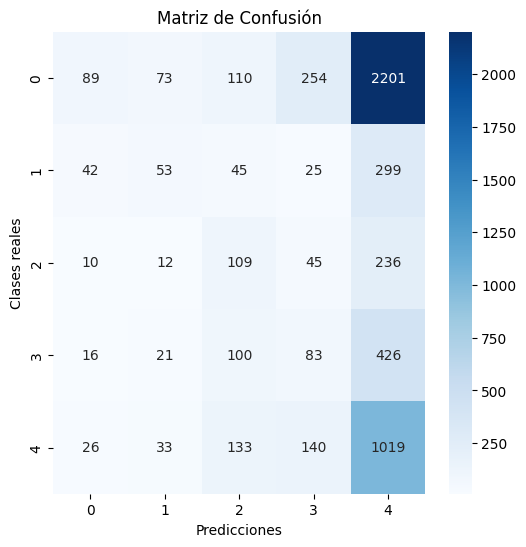
Entrenamos el modelo con los datos de entrenamiento y validación, utilizando el planificador de tasa de aprendizaje. Guardamos el modelo entrenado para su uso posterior. Este script prepara y entrena un modelo de detección de objetos utilizando MobileNetV2, asegurándose de que los datos estén correctamente procesados y listos para el entrenamiento.

import tensorflow as tf  
import numpy as np  
from collections import Counter  
  
# Definición de constantes  
IMG\_SIZE = 224  
NUM\_CLASSES = 5  
MAX\_BOXES = 10  
  
# Definición del modelo MobileNetV2  
from tensorflow.keras.applications import MobileNetV2  
from tensorflow.keras.layers import Dense, Flatten, Reshape, Dropout  
from tensorflow.keras.models import Model  
from tensorflow.keras.regularizers import l2  
  
# Inicialización del modelo base  
base\_model = MobileNetV2(weights="imagenet", include\_top=False, input\_shape=(IMG\_SIZE, IMG\_SIZE, 3))  
  
x = Flatten()(base\_model.output)  
x = Dense(512, activation="relu", kernel\_regularizer=l2(0.001))(x)  
x = Dropout(0.5)(x) # Aumento del Dropout para evitar overfitting  
  
x\_class = Dense(MAX\_BOXES \* NUM\_CLASSES, activation="softmax")(x)  
x\_class = Reshape((MAX\_BOXES, NUM\_CLASSES), name="class\_output")(x\_class)  
  
x\_bbox = Dense(MAX\_BOXES \* 4, activation="linear")(x)  
x\_bbox = Reshape((MAX\_BOXES, 4), name="bbox\_output")(x\_bbox)  
  
# Creación del modelo final  
model = Model(inputs=base\_model.input, outputs={"class\_output": x\_class, "bbox\_output": x\_bbox})  
  
# Compilación del modelo con ajuste del learning rate y funciones de pérdida  
model.compile(  
 optimizer=tf.keras.optimizers.Adam(learning\_rate=0.0001), # Reducción del learning rate  
 loss={  
 "class\_output": tf.keras.losses.CategoricalCrossentropy(from\_logits=False),  
 "bbox\_output": tf.keras.losses.Huber()  
 },  
 metrics={  
 "class\_output": "accuracy",  
 "bbox\_output": "mean\_absolute\_error"  
 }  
)  
  
# Configuración del Learning Rate Scheduler  
lr\_scheduler = tf.keras.callbacks.ReduceLROnPlateau(  
 monitor='val\_loss',  
 factor=0.5,  
 patience=2,  
 verbose=1  
)  
  
# ===============================================  
# \*\*Entrenamiento del modelo\*\*  
# ===============================================  
history = model.fit(  
 X\_train, {"bbox\_output": y\_train\_bbox, "class\_output": y\_train\_class},  
 validation\_data=(X\_valid, {"bbox\_output": y\_valid\_bbox, "class\_output": y\_valid\_class}),  
 epochs=10,  
 batch\_size=4,  
 verbose=1,  
 callbacks=[lr\_scheduler]  
)  
**Generación y Visualización de la Matriz de Confusión para la Evaluación del Modelo**

Este script genera una matriz de confusión para evaluar el rendimiento del modelo de detección de objetos en la clasificación de las clases. permite identificar errores comunes en la clasificación, analizar el rendimiento del modelo y detectar posibles desbalances en la predicción de ciertas clases

from sklearn.metrics import confusion\_matrix  
import seaborn as sns  
import matplotlib.pyplot as plt  
import numpy as np  
  
# Extraer las clases reales y las predicciones  
y\_true = np.argmax(y\_valid\_class, axis=-1).flatten() # Clases reales  
pred\_classes = np.argmax(model.predict(X\_valid)["class\_output"], axis=-1).flatten() # Predicciones del modelo  
  
# Generar la matriz de confusión  
cm = confusion\_matrix(y\_true, pred\_classes)  
  
# Mostrar la matriz de confusión  
plt.figure(figsize=(6,6))  
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=range(NUM\_CLASSES), yticklabels=range(NUM\_CLASSES))  
plt.xlabel("Predicciones")  
plt.ylabel("Clases reales")  
plt.title("Matriz de Confusión")  
plt.show()

18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 31ms/step



**Evaluación del modelo SSD**

Esta función evalúa el modelo SSD en un conjunto de validación. Compara las predicciones con los valores reales y calcula métricas importantes como:

* Precisión (Precisión de las detecciones correctas)
* Recall (Capacidad del modelo para detectar todos los objetos)
* mAP (Media de la precisión en diferentes umbrales)
* FPS (Velocidad de inferencia en imágenes por segundo)

También genera un gráfico que muestra los resultados obtenidos.

import numpy as np  
import time  
import tensorflow as tf  
import matplotlib.pyplot as plt  
  
# Función para evaluar el modelo SSD con NMS  
def evaluate\_ssd(model, X\_valid, y\_valid\_class, y\_valid\_bbox, iou\_threshold=0.5, score\_threshold=0.1):  
 start\_time = time.time()  
  
 # Obtener predicciones del modelo  
 preds = model.predict(X\_valid)  
 pred\_bboxes = preds['bbox\_output'] # Cajas delimitadoras predichas  
 pred\_classes = preds['class\_output'] # Clases predichas  
  
 # Asegurar que las bounding boxes están en el rango [0,1]  
 pred\_bboxes = np.clip(pred\_bboxes, 0, 1)  
  
 # Normalizar las clases con softmax  
 pred\_classes = tf.nn.softmax(pred\_classes).numpy()  
  
 # Verificar valores mínimos y máximos de los puntajes  
 print(f"Min/Max de los puntajes predichos : {pred\_classes.min()} / {pred\_classes.max()}")  
  
 # Verificar número total de bounding boxes después de NMS  
 print(f"Número total de bounding boxes después de NMS : {sum(len(b) for b in filtered\_bboxes)}")  
  
 # Cálculo de métricas de rendimiento  
 precision\_list = []  
 recall\_list = []  
 iou\_list = []  
  
 for i in range(len(X\_valid)):  
 gt\_bbox = y\_valid\_bbox[i]  
 gt\_class = np.argmax(y\_valid\_class[i], axis=-1)  
  
 pred\_bbox = filtered\_bboxes[i] if i < len(filtered\_bboxes) else []  
 pred\_class = np.argmax(pred\_classes[i], axis=-1)  
  
 iou\_scores = []  
 correct\_detections = 0  
  
 for j in range(len(pred\_bbox)):  
 if j >= len(gt\_bbox): # 🔥 Evitar errores de índice fuera de rango  
 break  
  
 iou = compute\_iou(pred\_bbox[j], gt\_bbox[j])  
 iou\_scores.append(iou)  
  
 if iou > iou\_threshold and pred\_class[j] == gt\_class[j]:  
 correct\_detections += 1  
  
 iou\_list.append(np.mean(iou\_scores) if iou\_scores else 0)  
 precision = correct\_detections / max(len(pred\_bbox), 1)  
 recall = correct\_detections / max(len(gt\_bbox), 1)  
  
 precision\_list.append(precision)  
 recall\_list.append(recall)  
  
 end\_time = time.time()  
 inference\_time = (end\_time - start\_time) / len(X\_valid) \* 1000 # en ms  
 fps = 1000 / inference\_time  
  
 results = {  
 "Precisión": np.mean(precision\_list),  
 "Recall": np.mean(recall\_list),  
 "mAP": np.mean(iou\_list),  
 "FPS": fps  
 }  
  
 # Generar gráfico de métricas  
 plot\_metrics(results)  
  
 return results  
  
# Ejecutar la evaluación  
results\_ssd = evaluate\_ssd(model, X\_valid, y\_valid\_class, y\_valid\_bbox)  
print("\n=== Resultados de la validación SSD ===")  
for metric, value in results\_ssd.items():  
 print(f"{metric}: {value:.2f}")

**Visualización de Bounding Boxes en Imágenes de Validación**

Este script permite visualizar las imágenes del conjunto de validación con las anotaciones reales y las predicciones del modelo SSD. Es clave para analizar el rendimiento del modelo, detectar errores y ajustar la precisión de las predicciones en escenarios reales

import json  
import cv2  
import matplotlib.pyplot as plt  
import os  
import random  
import numpy as np  
  
# Función para visualizar imágenes con bounding boxes reales y predichas  
def visualize\_validation\_bounding\_boxes(dataset\_path, predictions, num\_images=5, confidence\_threshold=0.5):  
 """  
 Muestra imágenes del conjunto de validación con anotaciones reales (en color) y predicciones (en amarillo).  
 """  
 valid\_images\_path = os.path.join(dataset\_path, "valid", "images")  
 valid\_annotations\_path = os.path.join(dataset\_path, "valid", "annotations", "\_annotations.coco.json")  
  
 # Cargar las anotaciones COCO  
 with open(valid\_annotations\_path, 'r') as f:  
 annotations = json.load(f)  
  
 # Definir colores para cada clase  
 class\_colors = {  
 "persona": (255, 0, 0), # Rojo  
 "casco": (0, 255, 0), # Verde  
 "chaleco": (0, 0, 255), # Azul  
 "no-casco": (255, 255, 0), # Cian  
 "no-chaleco": (255, 165, 0) # Naranja  
 }  
  
 # Extraer información de anotaciones e imágenes  
 images\_info = {img["id"]: img["file\_name"] for img in annotations["images"]}  
 annotations\_info = annotations["annotations"]  
 categories = {cat["id"]: cat["name"].lower().strip() for cat in annotations["categories"]}  
  
 # Seleccionar imágenes aleatoriamente  
 selected\_images = random.sample(list(images\_info.keys()), min(num\_images, len(images\_info)))  
  
 for i, image\_id in enumerate(selected\_images):  
 image\_name = images\_info[image\_id]  
 image\_path = os.path.join(valid\_images\_path, image\_name)  
  
 if not os.path.exists(image\_path):  
 print(f"Imagen no encontrada: {image\_path}")  
 continue  
  
 # Cargar y convertir la imagen  
 image = cv2.imread(image\_path)  
 if image is None:  
 print(f"Error al cargar la imagen: {image\_path}")  
 continue  
 image = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)  
  
 # Obtener anotaciones reales  
 img\_annotations = [ann for ann in annotations\_info if ann["image\_id"] == image\_id]  
  
 # Dibujar bounding boxes reales  
 for ann in img\_annotations:  
 bbox = ann["bbox"] # [x, y, ancho, alto]  
 x, y, w, h = map(int, bbox)  
 class\_id = ann["category\_id"]  
 class\_name = categories.get(class\_id, "unknown")  
 color = class\_colors.get(class\_name, (255, 255, 255)) # Blanco si clase desconocida  
  
 cv2.rectangle(image, (x, y), (x + w, y + h), color, 2)  
 cv2.putText(image, class\_name, (x, y - 5), cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, color, 2)  
  
 # Verificar si existen predicciones  
 if i >= len(predictions):  
 print(f"no hay predicciones disponibles para {image\_name}")  
 continue  
  
 # Dibujar bounding boxes predichas  
 for bbox in predictions[i]:  
 if len(bbox) < 5:  
 continue # Ignorar si la bbox está incompleta  
  
 x\_min, y\_min, x\_max, y\_max, confidence = bbox  
  
 # Filtrar por umbral de confianza  
 if confidence < confidence\_threshold:  
 continue  
  
 # Convertir coordenadas normalizadas a píxeles  
 x1, y1, x2, y2 = (np.array([x\_min, y\_min, x\_max, y\_max]) \*  
 [image.shape[1], image.shape[0], image.shape[1], image.shape[0]]).astype(int)  
  
 # Dibujar en amarillo  
 cv2.rectangle(image, (x1, y1), (x2, y2), (0, 255, 255), 2)  
 cv2.putText(image, "pred", (x1, y1 - 5), cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (0, 255, 255), 2)  
  
 # Mostrar la imagen con anotaciones  
 plt.figure(figsize=(8, 6))  
 plt.imshow(image)  
 plt.title(f"{image\_name} - Predicciones en amarillo")  
 plt.axis("off")  
 plt.show()  
  
# Obtener las predicciones de bounding boxes y clases  
preds = model.predict(X\_valid)  
  
# Corrección: Asegurar que `preds['bbox\_output']` contiene un puntaje de confianza  
bbox\_preds = np.concatenate([preds["bbox\_output"], np.max(preds["class\_output"], axis=-1, keepdims=True)], axis=-1)  
  
# Ejecutar la visualización con correcciones  
visualize\_validation\_bounding\_boxes(dataset\_path, bbox\_preds)  
  

