


✓ ENTRENAMIENTO MODELO - Proyecto Final

HardVision

Autor: Iván Falcón Monzón

```
# -----
# 1. Montar Google Drive
# -----
from google.colab import drive
drive.mount('/content/drive')
# El dataset de entrenamiento esta en Google Drive compartido porque es muy grande para Github: https://drive.google.com/drive/folders/1eT3MkyZ9aWkNkTwwNsLaj6
```

 Mounted at /content/drive

✓ Importar librerías necesarias


```
# -----
# 2. Instalación de librerías necesarias
# -----

# Este bloque de código se encarga de instalar las librerías de Python requeridas para el proyecto.

# Instalar la librería 'ultralytics'.
# 'ultralytics' es la librería oficial que proporciona la implementación de los modelos YOLO (You Only Look Once),
# incluyendo YOLOv8, que es el modelo que vamos a entrenar más abajo.
# Contiene funciones para entrenamiento, validación, inferencia y exportación de modelos YOLO.
!pip install ultralytics

# Instalar la librería 'openimages'.
# 'openimages' es una herramienta que facilita el acceso y la descarga de subconjuntos del dataset Open Images.
# Este dataset es una colección muy grande de imágenes con anotaciones de objetos.
!pip install openimages

# Instalar la librería 'fiftyone'.
# 'fiftyone' es una librería de código abierto para construir, evaluar y visualizar datasets
# de visión por computadora. Es extremadamente útil para:
# - Cargar y explorar datasets de imágenes y anotaciones.
# - Visualizar datos de entrenamiento, validación y predicciones.
# - Encontrar y corregir errores en las anotaciones.
# - Filtrar y manipular subconjuntos de datos.
# Es una herramienta útil para la gestión y depuración del dataset durante el ciclo de vida del entrenamiento de un modelo.
!pip install fiftyone
```

 **Mostrar salida oculta**

✓ Importar imagenes para Unknown

```
# -----
# 3. Importar imagenes de Open images v6
# -----

import os # Importa el módulo 'os' para interactuar con el sistema operativo, como crear directorios y manejar rutas de archivos.
import fiftyone.zoo as foz # Importa el módulo 'zoo' de fiftyone con el alias 'foz'. 'fiftyone.zoo' permite cargar y gestionar datasets preexistentes, como Open Images.
from PIL import Image # Importa la clase 'Image' del módulo 'PIL' (Pillow), que se utiliza para abrir, manipular y guardar imágenes.

# Ruta base donde se guardarán las imágenes descargadas para la clase "unknown".
# Estas imágenes se organizarán dentro del dataset de entrenamiento.
output_dir = "/content/drive/MyDrive/CurEsplABD_ProyectoFinal_IvánFalcónMonzón/dataset/train/images/unknown"

# Define una lista de nombres de clases que se descargarán.
# Estas clases se agruparán bajo la categoría "unknown" para el entrenamiento del modelo.
# Esto es útil en escenarios donde ciertas clases no son el objetivo principal de detección
# pero se quiere que el modelo las reconozca como "no relevantes" o "desconocidas".
classes = ["pizza", "cake", "tv", "frisbee", "dog", "microwave", "suitcase", "backpack", "vase", "mouse"]

# Crea el directorio base especificado en 'output_dir' si no existe.
# 'exist_ok=True' evita que se lance un error si el directorio ya existe.
os.makedirs(output_dir, exist_ok=True)

# Define el tamaño al que se redimensionarán todas las imágenes descargadas.
# Homogeneizar el tamaño de las imágenes es una práctica común en el entrenamiento de modelos de visión
# para asegurar una entrada consistente al modelo. YOLOv8 suele trabajar con 640x640.
img_size = (640, 640)
```

```
# Itera sobre cada clase definida en la lista 'classes'.
for cls in classes:
    # Construye la ruta del directorio específico para la clase actual dentro de 'output_dir'.
    class_dir = os.path.join(output_dir, cls)
    # Crea el directorio de la clase si no existe.
    os.makedirs(class_dir, exist_ok=True)

    # Verifica si ya existen imágenes en la carpeta de la clase actual.
    # Esto es útil para reanudar descargas o evitar descargas duplicadas.
    # Filtra solo archivos que terminan en extensiones de imagen comunes.
    existing_images = [f for f in os.listdir(class_dir) if f.lower().endswith((".jpg", ".jpeg", ".png"))]
    # Si ya hay 100 o más imágenes en la carpeta, se asume que la descarga para esta clase está completa
    # y se salta a la siguiente clase.
    if len(existing_images) >= 100:
        print(f"Clase '{cls}' ya tiene imágenes. Saltando descarga.")
        continue # Pasa a la siguiente iteración del bucle.

print(f"Descargando clase: {cls}")

# Descarga imágenes desde el dataset Open Images V6 usando FiftyOne.
dataset = foz.load_zoo_dataset(
    "open-images-v6", # Especifica el dataset a cargar.
    split="train", # Descarga imágenes del split de entrenamiento.
    label_types=["classifications"], # Solicita solo etiquetas de clasificación (no bounding boxes).
    classes=[cls], # Filtra para descargar solo imágenes que contengan la clase actual.
    max_samples=100, # Limita la descarga a un máximo de 100 imágenes por clase.
    shuffle=True # Mezcla las imágenes antes de seleccionarlas.
)


# Itera sobre cada muestra (imagen) descargada del dataset.
for sample in dataset:
    img = sample.filepath # Obtiene la ruta del archivo de la imagen descargada.
    img_name = os.path.basename(img) # Extrae el nombre del archivo de la ruta.
    dest_path = os.path.join(class_dir, img_name) # Construye la ruta de destino para guardar la imagen.

    # Verifica si la imagen ya existe en la ruta de destino para evitar sobrescribir.
    if os.path.exists(dest_path):
        continue # Si ya existe, salta a la siguiente imagen.

    try:
        # Abre la imagen usando Pillow.
        with Image.open(img) as im:
            im = im.convert("RGB") # Convierte la imagen a formato RGB para asegurar consistencia (algunas pueden ser RGBA...).
            im = im.resize(img_size) # Redimensiona la imagen al tamaño predefinido (640x640).
            im.save(dest_path, "JPEG") # Guarda la imagen redimensionada en la ruta de destino con formato JPEG.
    except Exception as e:
        # Captura y muestra cualquier error que ocurra durante el procesamiento de una imagen.
        print(f"Error con la imagen {img}: {e}")

print(f" {cls} terminado.") # Imprime un mensaje cuando la descarga y procesamiento de una clase ha finalizado.

print("¡Todo listo! Imágenes descargadas y organizadas.") # Mensaje final indicando que todo el proceso ha terminado.
```

 **Mostrar salida oculta**

```
# -----
# 4. Generación automática de etiquetas para YOLO (formato txt)
# -----

# Este bloque de código se encarga de generar archivos de etiquetas en formato YOLO (.txt)
# para las imágenes que fueron descargadas y clasificadas como "unknown".
# Para simplificar, cada imagen se marca como "unknown" (clase 10) con una caja delimitadora
# que ocupa una porción central de la imagen.

import os # Importa el módulo 'os' para interactuar con el sistema de archivos (crear directorios, listar archivos, etc.).
import fiftyone.zoo as foz # Importa fiftyone.zoo, aunque no se usa directamente en este bloque, se mantuvo del código anterior.
from PIL import Image # Importa la clase 'Image' de Pillow para manejar imágenes, aunque en este caso solo se usa para obtener dimensiones.

# Define la ruta raíz donde se guardarán los archivos de etiquetas (.txt).
labels_root = "/content/drive/MyDrive/CurEspIABD_ProyectoFinal_IvánFalcónMonzón/dataset/train/labels/unknown"

# Define el ID de clase para las imágenes "unknown".
# Es crucial que este ID (10 en este caso) coincida con el ID asignado a la clase 'unknown'
# en el archivo de configuración 'data.yaml' de YOLOv8.
unknown_class_id = 10 # Debe coincidir con data.yaml

# Crea el directorio donde se guardarán los archivos de etiquetas si no existe.
# 'exist_ok=True' previene errores si el directorio ya ha sido creado.
os.makedirs(labels_root, exist_ok=True)

# Itera sobre cada subcarpeta dentro de 'output_dir'.
```

```
# 'output_dir' fue definido en el bloque de código anterior y contiene las imágenes descargadas
# organizadas por sus clases originales (pizza, cake, etc.), que ahora consideramos "unknown".
for class_folder in os.listdir(output_dir):
    # Construye la ruta completa a la carpeta de la clase actual.
    class_folder_path = os.path.join(output_dir, class_folder)

    # Verifica si la entrada actual es realmente un directorio. Si no lo es (por ejemplo, es un archivo),
    # se salta a la siguiente iteración del bucle.
    if not os.path.isdir(class_folder_path):
        continue

    # Itera sobre cada archivo dentro de la carpeta de la clase actual.
    for filename in os.listdir(class_folder_path):
        # Asegura que solo se procesen archivos de imagen (en este caso, JPEG).
        if not filename.lower().endswith(".jpg"):
            continue

        # Construye la ruta completa a la imagen actual.
        image_path = os.path.join(class_folder_path, filename)

        # Genera el nombre del archivo de etiqueta correspondiente.
        # Por ejemplo, para 'imagen.jpg', el archivo de etiqueta será 'imagen.txt'.
        label_filename = os.path.splitext(filename)[0] + ".txt"
        label_path = os.path.join(labels_root, label_filename)

        # Verificación de existencia del archivo de etiqueta
        # Si el archivo de etiqueta (.txt) para esta imagen ya existe, se salta su creación.
        # Esto evita reescribir etiquetas innecesariamente y permite reanudar el proceso.
        if os.path.exists(label_path):
            # print(f"Etiqueta para {filename} ya existe. Saltando.") # Descomentar para ver mensajes
            continue

        # Abrir imagen para obtener dimensiones. Aunque en este caso la caja es fija,
        # en escenarios más complejos las dimensiones son necesarias para normalizar las coordenadas YOLO.
        try:
            with Image.open(image_path) as im:
                width, height = im.size # Obtiene el ancho y alto de la imagen.
        except Exception as e:
            # Si hay un error al abrir la imagen, se imprime un mensaje y se salta a la siguiente.
            print(f"Error con imagen: {image_path} - {e}")
            continue

        # Define las coordenadas y dimensiones de la caja delimitadora en formato YOLO normalizado.
        # YOLO usa (x_center, y_center, width, height) donde todos los valores están normalizados
        # entre 0 y 1, relativos al ancho y alto de la imagen.
        x_center = 0.5 # Centro horizontal de la caja (50% del ancho de la imagen).
        y_center = 0.5 # Centro vertical de la caja (50% del alto de la imagen).
        w = 0.6 # Ancho de la caja (60% del ancho de la imagen).
        h = 0.6 # Alto de la caja (60% del alto de la imagen).

        # Abre el archivo de etiqueta en modo escritura ('w').
        # Si el archivo no existe, lo crea; si existe, lo sobrescribe (aunque la verificación anterior lo evita).
        with open(label_path, "w") as f:
            # Escribe la línea de la etiqueta en el archivo. El formato es:
            # <class_id> <x_center> <y_center> <width> <height>
            f.write(f"{unknown_class_id} {x_center} {y_center} {w} {h}\n")

print("Etiquetas generadas automáticamente para clase 'unknown'.")
```

 Etiquetas generadas automáticamente para clase 'unknown'.


```
import os # Importa el módulo 'os' para interactuar con el sistema de archivos, como listar directorios.

# Define la ruta al directorio donde se encuentran las etiquetas generadas para la clase "unknown".
labels_dir = "/content/drive/MyDrive/CurEspIABD_ProyectoFinal_IvánFalcónMonzón/dataset/train/labels/unknown"

# Lista todos los archivos en el directorio 'labels_dir' y filtra aquellos que terminan con ".txt".
# Esto crea una lista de todos los archivos de etiquetas YOLO generados.
labels = [f for f in os.listdir(labels_dir) if f.endswith(".txt")]

# Imprime el número total de archivos de etiquetas encontrados en el directorio.
print(f"Total etiquetas en unknown: {len(labels)}")

# Imprime los primeros 5 nombres de archivos de etiquetas como ejemplo,
# lo que permite una verificación rápida de que se han generado los archivos.
print("Ejemplos:", labels[:5])
```

 Total etiquetas en unknown: 100
Ejemplos: ['c4281407137dacc0.txt', 'e8360570bde54a58.txt', '4dfb763cad6e41fd.txt', 'db2a95d178776844.txt', '8c5373a8ee722f31.txt']

Entrenamiento del modelo

Este entrenamiento con el modelo de YOLOv8 en Google Colab con el entorno de ejecución de GPU-T4 suele tardar entre 45-50 minutos

```
# -----
# 5. Entrenamiento del modelo YOLOv8
# -----

# Importa la clase YOLO de la librería ultralytics.
# Esta clase es la interfaz principal para cargar, entrenar, validar y usar modelos YOLOv8.
from ultralytics import YOLO

# Carga un modelo YOLOv8 pre-entrenado.
# "yolov8n.pt" se refiere a la versión "nano" de YOLOv8, que es la más pequeña y rápida,
# ideal para entornos con recursos limitados o para pruebas iniciales.
# Los modelos pre-entrenados han sido previamente entrenados en grandes datasets (como COCO)
# y ya han aprendido a detectar una amplia variedad de objetos. Esto permite un "fine-tuning"
# más rápido y eficiente para el dataset.
model = YOLO("yolov8n.pt")

# Inicia el proceso de entrenamiento del modelo.
# Se pasan varios argumentos clave para configurar el entrenamiento:
model.train(
    # 'data': Ruta al archivo de configuración YAML de tu dataset.
    # Este archivo 'data.yaml' es crucial porque define:
    # - Las rutas a las carpetas de imágenes de entrenamiento y validación.
    # - Las rutas a las carpetas de etiquetas de entrenamiento y validación.
    # - Los nombres de las clases que el modelo aprenderá a detectar.
    data="content/drive/MyDrive/CurEspIABD_ProyectoFinal_IvánFalcónMonzón/dataset/data.yaml",


    # 'epochs': Número de épocas (iteraciones completas sobre todo el dataset de entrenamiento).
    # Más épocas pueden llevar a un mejor rendimiento, pero también aumentan el tiempo de entrenamiento
    # y el riesgo de overfitting (que el modelo memorice los datos de entrenamiento en lugar de aprender patrones generales).
    epochs=50,

    # 'imgsz': Tamaño de la imagen de entrada para el modelo durante el entrenamiento.
    # Las imágenes se redimensionarán a este tamaño antes de ser alimentadas a la red.
    # Un tamaño de 640x640 es un estándar común para YOLOv8.
    imgsz=640,

    # 'batch': Tamaño del lote (batch size).
    # Número de imágenes procesadas simultáneamente por el modelo antes de actualizar sus pesos.
    # Un batch size más grande puede acelerar el entrenamiento si tienes suficiente VRAM en la GPU,
    # pero un batch size más pequeño puede ayudar a la generalización en algunos casos.
    batch=16,

    # 'workers': Número de procesos (workers) a usar para la carga de datos.
    # Un mayor número de workers puede acelerar la preparación de los datos de entrada,
    # especialmente si el cuello de botella está en la lectura y preprocesamiento de imágenes.
    workers=2,

    # 'name': Nombre para el directorio donde se guardarán los resultados del entrenamiento (pesos, métricas, gráficos).
    # Esto ayuda a organizar tus experimentos si entrenas múltiples modelos o variaciones.
    name="resultados"
)
```

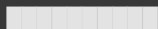
 [Mostrar salida oculta](#)

Desglose de los Resultados del Entrenamiento

Proceso de Entrenamiento (Salida durante las Épocas)

Epoch GPU_mem box_loss cls_loss dfl_loss Instances Size 50/


50 2.69G 0.6362 0.5164 0.9723 10 640: 100%|

 178/178 [00:53<00:00, 3.32it/s]

Esta línea es un resumen del progreso de una época (o un estado intermedio) del entrenamiento.

Epoch: Indica el número de la época de entrenamiento actual.

- GPU_mem: Muestra la cantidad de memoria de la GPU (en este caso, 2.69 GB) que está siendo utilizada por el proceso de entrenamiento. Esto es útil para monitorear el consumo de recursos.
- box_loss: Es la pérdida de la caja delimitadora (Bounding Box Loss). Mide qué tan bien el modelo está prediciendo la ubicación y el tamaño de los objetos. Un valor más bajo indica que las cajas predichas se ajustan mejor a las cajas reales. En este caso, 0.6362.

- `cls_loss`: Es la pérdida de clasificación (Classification Loss). Mide qué tan bien el modelo está clasificando correctamente los objetos dentro de las cajas. Un valor más bajo indica una mejor clasificación. Aquí, 0.5164.
- `dfl_loss`: Es la pérdida DFL (Distribution Focal Loss). Esta es una pérdida específica de YOLOv8 que ayuda a la regresión de las cajas al enfocarse en la calidad de la distribución de los puntos que definen los límites de la caja. Un valor más bajo es mejor. En este caso, 0.9723.
- `Instances`: El número de instancias (objetos) detectadas o procesadas en este paso/época (10).
- `Size`: El tamaño de las imágenes que se están procesando (640x640 píxeles). 100%  178/178 [00:53<00:00, 3.32it/s]: Esta es una barra de progreso. Indica que se han procesado 178 de 178 lotes (batches) de datos, completando el 100% de la época en 53 segundos, con una tasa de 3.32 iteraciones por segundo.

Class Images Instances Box(P R mAP50 mAP50-95): 100%  3/3 [00:01<00:00, 1.97it/s] all 83 869 0.925 0.907 0.921 0.795

Esta sección resume las métricas de validación por clase (aunque aquí solo se muestra el promedio "all" para todas las clases) después de una época de entrenamiento.

- `all`: Representa las métricas promediadas sobre todas las clases detectables por tu modelo.
- `Images`: El número de imágenes en el conjunto de validación procesado (83 imágenes).
- `Instances`: El número total de instancias (objetos reales) en esas imágenes de validación (869 objetos).
- `Box(P)` (Precision): La precisión promedio para todas las clases. Indica qué proporción de las detecciones del modelo fueron correctas. Un valor de 0.925 (92.5%) es excelente.
- `R` (Recall): El recall (o exhaustividad) promedio para todas las clases. Indica qué proporción de los objetos reales en las imágenes fueron detectados por el modelo. Un valor de 0.907 (90.7%) también es muy bueno.
- `mAP50`: El mean Average Precision (mAP) con un umbral de Intersection over Union (IoU) del 50%. Es decir, una detección se considera correcta si su IoU con la verdad terrestre es al menos 0.5. Un mAP50 de 0.921 (92.1%) es un rendimiento muy sólido.
- `mAP50-95`: El mAP promediado sobre diferentes umbrales de IoU, desde 0.5 hasta 0.95 (en pasos de 0.05). Esta es una métrica más estricta y representativa del rendimiento general. Un mAP50-95 de 0.795 (79.5%) indica que el modelo no solo detecta los objetos, sino que también los localiza con bastante precisión.

```
maps: array([ 0.63008, 0.67697, 0.91444, 0.88291, 0.
```

```
64041, 0.86881, 0.83743, 0.87256, 0.73619, 0.84934, 0.
```

```
79091])
```

Este es un array que probablemente contiene los valores de mAP por clase (en el orden en que las clases están definidas internamente, no necesariamente el orden de nombres de abajo), y el último valor (0.79091) es el mAP promedio global. Como tienes 11 clases (names tiene 11 entradas), este array muestra el mAP para cada una de tus clases, y el último valor es el promedio.

```
names: {0: 'atx_12v', 1: 'atx_power', 2: 'cpu', 3: 'cpu-slot', 4:
```

```
'fan-bracket', 5: 'm.2-ssd-slot', 6: 'pcie-slot', 7: 'ram', 8: 'ram-slot', 9:
```

```
'ssd', 10: 'unknown'}
```

Este es un diccionario que mapea los IDs numéricos de las clases a sus nombres legibles. Es crucial para interpretar las predicciones del modelo. Vemos que tienes 11 clases, incluyendo la clase unknown con ID 10, tal como la configuraste.

Métricas finales de rendimiento del modelo validado en un formato más estructurado:

- `metrics/precision(B)`: Precisión final de 0.9246.
- `metrics/recall(B)`: Recall final de 0.9065.
- `metrics/mAP50(B)`: [mAP@0.5](#) final de 0.9206.
- `metrics/mAP50-95(B)`: [mAP@0.5:0.95](#) final de 0.7909.
- `fitness`: El valor final de fitness compuesto 0.8038.

Estos valores son muy consistentes con los que se vieron en la tabla de resumen durante el entrenamiento, confirmando un excelente rendimiento.

En Resumen

Los resultados de entrenamiento muestran que tu modelo YOLOv8n ha alcanzado un rendimiento muy fuerte para la tarea de detección de objetos, con una alta precisión y recall, y un excelente mAP tanto a un umbral bajo (0.5) como a través de un rango de umbrales estrictos


(0.5:0.95). La pérdida de las cajas, clasificación y DFL disminuyeron a valores razonables, lo que indica un aprendizaje efectivo.

✓ Comprobaciones del modelo entrenado

```
# Cargar el modelo entrenado
# Instancia la clase YOLO y carga los pesos del modelo que fue entrenado previamente.
# La ruta "/content/runs/detect/resultados/weights/best.pt" es la ubicación predeterminada
# donde Ultralytics guarda el modelo con el mejor rendimiento (medido por mAP)
# al final del entrenamiento cuyo nombre de ejecución fue "yolov8n_unknown_train".
model = YOLO("/content/runs/detect/resultados/weights/best.pt")

# Hacer predicciones sobre una imagen o carpeta
# Utiliza el modelo cargado para realizar inferencias (predicciones de objetos).
# 'source': Define la entrada para la predicción.
# ¡IMPORTANTE!: La ruta "/ruta/a/imagenes/de/prueba" es un marcador de posición y DEBE ser reemplazada
# por la ruta real a la imagen o carpeta de imágenes que deseas probar.
# Por ejemplo:
# - Para una imagen específica: 'source="/content/drive/MyDrive/dataset/test/imagen_desconocida.jpg"'
# - Para una carpeta con varias imágenes: 'source="/content/drive/MyDrive/dataset/test_images/'
# 'imgsz': Especifica el tamaño de la imagen al que se redimensionarán las entradas para la inferencia.
# Debe coincidir con el 'imgsz' utilizado durante el entrenamiento (640 en este caso).
# 'conf': Establece el umbral de confianza. Solo las detecciones con una probabilidad superior
# a este valor (0.3 o 30% en este caso) serán consideradas y mostradas. Ajusta este valor
# para controlar la sensibilidad de las detecciones (menor valor = más detecciones, posiblemente más falsos positivos).
results = model.predict(source="/content/drive/MyDrive/CurEspIABD_ProyectoFinal_IvánFalcónMonzón/dataset/test/images", imgsz=640, conf=0.3)

# Mostrar visualmente
# Accede al primer objeto de resultados (asumiendo que se procesa una sola imagen o que solo se quiere ver la primera).
# El método '.show()' visualiza la imagen con las cajas delimitadoras y etiquetas de las detecciones realizadas.
# Esto abrirá una ventana emergente o mostrará la imagen directamente en la salida de Colab.
results[0].show()
```

 **Mostrar salida oculta**

Cada línea representa el resultado de la detección de objetos en una imagen específica.

- image X/56: Indica que esta es la imagen número X de un total de 56 imágenes que se están procesando.
- [/content/drive/MyDrive/CurEspIABD_ProyectoFinal_IvánFalcónMonzón/dataset/test/images/screenshot_384.jpg.rf.08456c533f2caaf264c86ddd749238.jpg](#): Esta es la ruta completa de la imagen que ha sido procesada. Puedes ver que las imágenes provienen de una carpeta de "test" (prueba) dentro de tu Google Drive.
- 640x640: Confirma que la imagen fue redimensionada a 640x640 píxeles antes de ser pasada al modelo para la inferencia, lo cual es consistente con el tamaño de entrenamiento.
- 1 cpu-slot, 1 m.2-ssd-slot, 1 pcie-slot, 4 ram-slots: Esta es la parte más importante. Enumera los objetos que el modelo ha detectado en la imagen. Por cada detección, muestra:
 - La cantidad de objetos detectados de esa clase (por ejemplo, "1 cpu-slot", "4 ram-slots").
 - El nombre de la clase detectada (por ejemplo, cpu-slot, m.2-ssd-slot, pcie-slot, ram-slots).
- 7.5ms (o similar): Este es el tiempo que tardó el modelo en realizar la inferencia (es decir, en procesar esa imagen y generar las detecciones). Los valores están en milisegundos (ms), lo que indica que el modelo es bastante rápido en sus predicciones.
- Resumen de la última línea: Speed
 - Speed: 2.3ms preprocess, 8.9ms inference, 1.8ms postprocess per image at shape (1, 3, 640, 640)
 - Esta línea final es un resumen de la velocidad promedio del proceso de inferencia para todo el lote de imágenes (en este caso, las 56 imágenes de prueba).
- 2.3ms preprocess: Tiempo promedio para el preprocesamiento de cada imagen (como redimensionar y normalizar).
- 8.9ms inference: Tiempo promedio que el modelo tarda en hacer la predicción real por cada imagen. Esta es la parte central de la detección.
- 1.8ms postprocess: Tiempo promedio para el postprocesamiento de las salidas del modelo (aplicar umbrales de confianza, Non-Maximum Suppression para eliminar cajas duplicadas, etc.) para obtener las detecciones finales. per image at shape (1, 3, 640, 640):
 - Indica que estas velocidades son por imagen, con un formato de entrada de (batch_size=1, canales_rgb=3, altura=640, ancho=640).

Interpretación General

Estos resultados indican que el modelo está funcionando correctamente, identificando y clasificando los diferentes componentes de las placas base. Las velocidades son muy buenas, lo que sugiere que el modelo es eficiente para aplicaciones en tiempo real o con un alto volumen de procesamiento.

```
# Muestra los nombres de las clases que el modelo ha sido entrenado para detectar.
# El atributo 'names' de un objeto YOLO contiene un diccionario donde las claves son los IDs de las clases
# y los valores son los nombres legibles de esas clases.
# Esto es fundamental para entender qué es lo que el modelo puede identificar.
print(model.names)
```

```
🔗 {0: 'atx_12v', 1: 'atx_power', 2: 'cpu', 3: 'cpu-slot', 4: 'fan-bracket', 5: 'm.2-ssd-slot', 6: 'pcie-slot', 7: 'ram', 8: 'ram-slot', 9: 'ssd', 10: 'unknown'}
```

Este diccionario es fundamental porque establece la correspondencia entre los IDs numéricos (índices) que el modelo utiliza internamente para sus predicciones, y los nombres legibles de las clases de objetos que ha sido entrenado para detectar.

- Claves (números 0 a 10): Son los identificadores únicos que el modelo asigna a cada tipo de objeto. Cuando el modelo detecta algo, devuelve el ID numérico de la clase.
- Valores (nombres como 'atx_12v', 'cpu', 'unknown'): Son las etiquetas humanas que corresponden a esos IDs numéricos. Por ejemplo:

Si el modelo detecta un objeto y le asigna el ID 2, que se refiere a un 'cpu'.

Si detecta algo con el ID 10, es de la clase 'unknown', que es como esta configurada en la generación de etiquetas. En el archivo data.yaml.

Este mapeo es crucial para la interpretación de los resultados de las predicciones del modelo, ya que te permite entender qué tipo de objeto ha sido detectado en cada imagen.

✓ Curvas de pérdida del entrenamiento

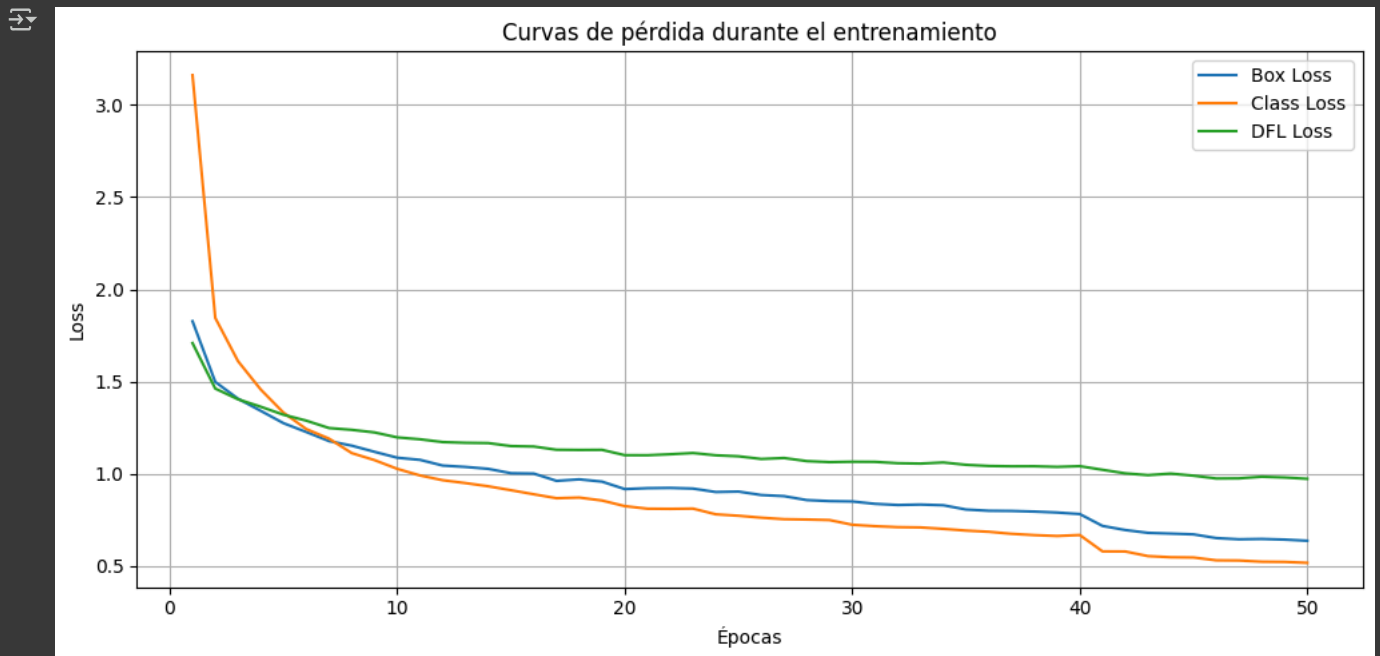
Este script de Python está diseñado para visualizar las curvas de pérdida del entrenamiento de un modelo YOLOv8 a partir del archivo de resultados generado por Ultralytics.

```
import pandas as pd # Importa la librería pandas, fundamental para el manejo y análisis de datos tabulares, especialmente útil para leer archivos CSV.
import matplotlib.pyplot as plt # Importa el módulo pyplot de matplotlib, que es la librería estándar para la creación de gráficos y visualizaciones en Python.

# Cargar y limpiar columnas
# Carga el archivo CSV de resultados del entrenamiento en un DataFrame de pandas.
# La ruta '/content/runs/detect/resultados/results.csv' es la ubicación por defecto donde Ultralytics guarda
# las métricas de entrenamiento (pérdidas, mAP, etc.) para una ejecución llamada 'train2'.
df = pd.read_csv('/content/runs/detect/resultados/results.csv')
# Limpia los nombres de las columnas eliminando espacios en blanco al principio y al final.
# Esto asegura que puedas acceder a las columnas por sus nombres sin problemas de coincidencia de espacios.
df.columns = df.columns.str.strip()

# Extraer datos
# Extrae las columnas relevantes del DataFrame para la graficación.
# 'epoch': El número de época de entrenamiento.
# 'train/box_loss': La pérdida de la caja delimitadora durante el entrenamiento. Mide la precisión de las coordenadas y dimensiones de las cajas predichas.
# 'train/cls_loss': La pérdida de clasificación durante el entrenamiento. Mide la precisión con la que el modelo clasifica los objetos dentro de las cajas.
# 'train/dfl_loss': La pérdida de DFL (Distribution Focal Loss). Es una pérdida específica de YOLOv8 que mejora la precisión de la regresión de la caja delimitadora.
epochs = df['epoch']
loss_box = df['train/box_loss']
loss_cls = df['train/cls_loss']
loss_dfl = df['train/dfl_loss']

# Graficar
# Crea una nueva figura para el gráfico con un tamaño específico (10 pulgadas de ancho por 5 de alto).
plt.figure(figsize=(10, 5))
# Dibuja la curva de la pérdida de la caja delimitadora ('Box Loss') en función de las épocas.
plt.plot(epochs, loss_box, label='Box Loss')
# Dibuja la curva de la pérdida de clasificación ('Class Loss').
plt.plot(epochs, loss_cls, label='Class Loss')
# Dibuja la curva de la pérdida de DFL ('DFL Loss').
plt.plot(epochs, loss_dfl, label='DFL Loss')
# Establece el título del gráfico.
plt.title('Curvas de pérdida durante el entrenamiento')
# Establece la etiqueta del eje X (épocas).
plt.xlabel('Épocas')
# Establece la etiqueta del eje Y (valor de la pérdida).
plt.ylabel('Loss')
# Muestra una leyenda que identifica cada curva.
plt.legend()
# Añade una cuadrícula al gráfico para facilitar la lectura de los valores.
plt.grid(True)
# Ajusta automáticamente los parámetros de la subtrama para que la figura quepa en el área del dibujo.
plt.tight_layout()
# Muestra el gráfico en pantalla.
plt.show()
```

Análisis de las Curvas de Pérdida El gráfico muestra la evolución de tres tipos de pérdida:

1. Box Loss (Azul): Pérdida relacionada con la precisión de las cajas delimitadoras (ubicación y tamaño de los objetos detectados).
2. Class Loss (Naranja): Pérdida relacionada con la precisión de la clasificación de los objetos dentro de las cajas.
3. DFL Loss (Verde): Pérdida específica de YOLOv8 que mejora la calidad de la regresión de las cajas.

Conclusión sobre el Gráfico de Pérdida:

El gráfico de curvas de pérdida muestra un entrenamiento exitoso y estable. La disminución constante de las pérdidas es que el modelo está aprendiendo eficazmente a realizar las tareas de detección y clasificación. La caída observada en Class Loss alrededor de la época 40 es un buen signo de optimización.

✓ Métricas de rendimiento mAP (mean Average Precision)

Este fragmento de código Python tiene como objetivo visualizar las métricas de rendimiento mAP (mean Average Precision) de un modelo YOLOv8 a lo largo del entrenamiento, utilizando los datos de un archivo CSV de resultados.

```
import matplotlib.pyplot as plt # Importa el módulo pyplot de matplotlib para crear gráficos.
# Se asume que 'epochs' y el DataFrame 'df' ya han sido cargados
# y limpiados en el bloque de código anterior, incluyendo la línea:
# df = pd.read_csv('runs/detect/train2/results.csv')
# df.columns = df.columns.str.strip()
# epochs = df['epoch']

# Crea una nueva figura para el gráfico con un tamaño específico (10 pulgadas de ancho por 5 de alto).
plt.figure(figsize=(10, 5))

# Dibuja la curva de mAP@0.5.
# 'df['metrics/mAP50(B)']' accede a la columna del DataFrame que contiene los valores de mAP con un umbral IoU de 0.5.
# '(B)' en el nombre de la columna a menudo se refiere a las métricas calculadas en el conjunto de "Bounding boxes".
plt.plot(epochs, df['metrics/mAP50(B)'], label='mAP@0.5')

# Dibuja la curva de mAP@0.5:0.95.
# 'df['metrics/mAP50-95(B)']' accede a la columna con los valores de mAP promediados sobre diferentes umbrales IoU.
plt.plot(epochs, df['metrics/mAP50-95(B)'], label='mAP@0.5:0.95')

# Establece el título del gráfico.
plt.title('mAP durante el entrenamiento')

# Establece la etiqueta del eje X (épocas).
plt.xlabel('Épocas')

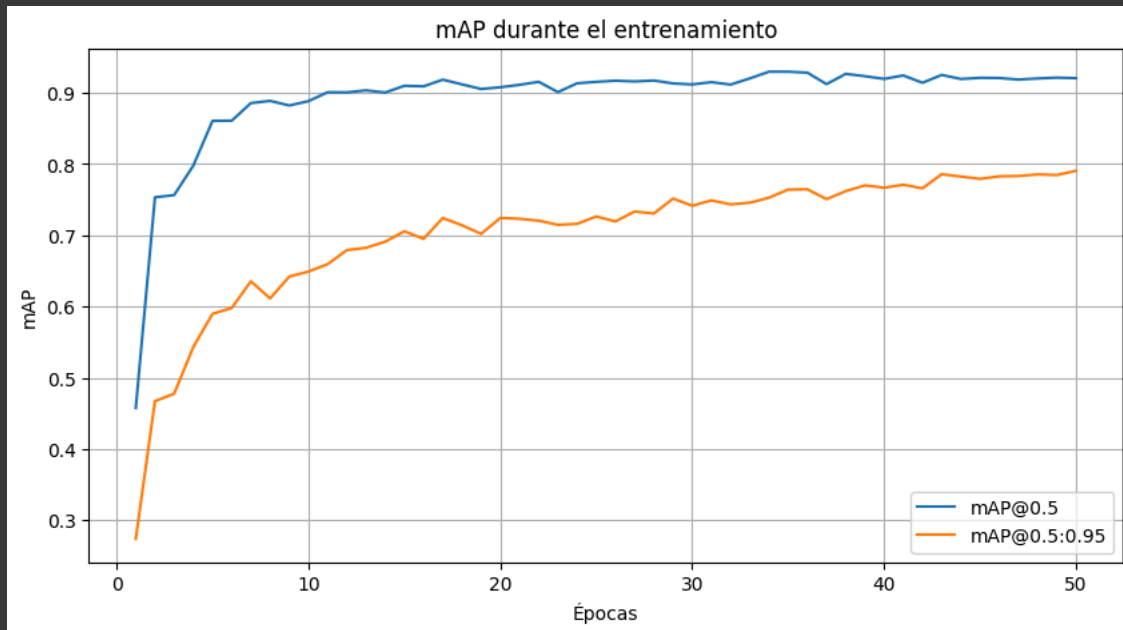
# Establece la etiqueta del eje Y (valor de mAP).
plt.ylabel('mAP')

# Muestra una leyenda para identificar las dos curvas de mAP.
plt.legend()

# Añade una cuadrícula al gráfico para facilitar la lectura de los valores.
plt.grid(True)
```



```
# Muestra el gráfico en pantalla.
plt.show()
```



Conclusión sobre el Gráfico de mAP:

El gráfico de mAP confirma que tu modelo ha entrenado de manera muy efectiva. Las métricas de rendimiento son muy altas y consistentes, lo que indica que el modelo es robusto tanto en la identificación de los objetos como en la precisión de sus cajas delimitadoras.

El entrenamiento hasta 50 épocas es positivo, ya que el [mAP@0.5:0.95](#) siguió mejorando hasta el final.

Curvas de Precisión y Recall

Este bloque de código sirve para visualizar las curvas de Precisión y Recall de tu modelo YOLOv8 durante el entrenamiento, utilizando los datos recopilados en el archivo CSV de resultados.

```
import matplotlib.pyplot as plt # Importa el módulo pyplot de matplotlib para crear gráficos.
# Asumimos que 'epochs' y el DataFrame 'df' ya fueron cargados y limpiados en bloques de código anteriores.

# Crea una nueva figura para el gráfico con un tamaño específico (10 pulgadas de ancho por 5 de alto).
plt.figure(figsize=(10, 5))

# Dibuja la curva de Precisión.
# 'df['metrics/precision(B)']' accede a la columna del DataFrame que contiene los valores de precisión.
# El '(B)' indica que las métricas se calcularon sobre las 'Bounding boxes' (cajas delimitadoras).
plt.plot(epochs, df['metrics/precision(B)'], label='Precisión')

# Dibuja la curva de Recall (Exhaustividad).
# 'df['metrics/recall(B)']' accede a la columna del DataFrame que contiene los valores de recall.
plt.plot(epochs, df['metrics/recall(B)'], label='Recall')

# Establece el título del gráfico.
plt.title('Precisión y Recall')

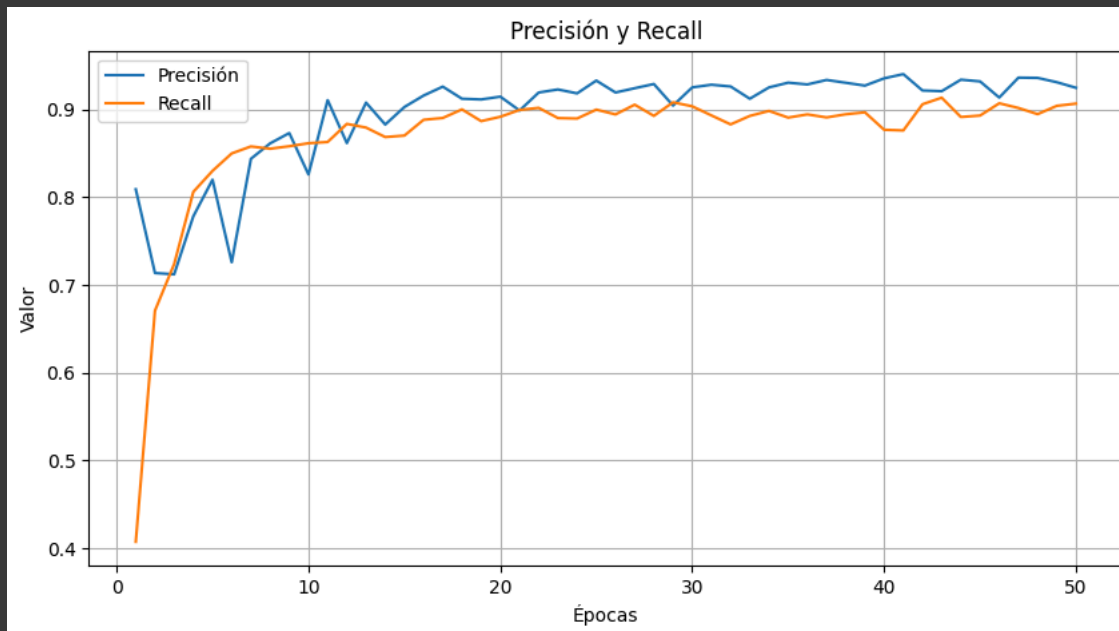
# Establece la etiqueta del eje X como 'Épocas'.
plt.xlabel('Épocas')

# Establece la etiqueta del eje Y como 'Valor', ya que ambas métricas son valores entre 0 y 1.
plt.ylabel('Valor')

# Muestra una leyenda para diferenciar las curvas de Precisión y Recall.
plt.legend()

# Añade una cuadrícula al gráfico para facilitar la lectura de los valores.
plt.grid(True)

# Muestra el gráfico en pantalla.
plt.show()
```



Análisis de las Curvas de Precisión y Recall

El gráfico muestra la evolución de dos métricas complementarias:

1. Precisión (Azul): Proporción de detecciones positivas que fueron correctas. Responde a "¿De todas las detecciones que hizo mi modelo, cuántas eran realmente objetos?" (pocos falsos positivos).
2. Recall (Naranja): Proporción de objetos positivos reales que fueron correctamente identificados. Responde a "¿Cuántos de los objetos que realmente existen detectó mi modelo?" (pocos falsos negativos).

Conclusión sobre el Gráfico de Precisión y Recall:

El modelo ha logrado una precisión y un recall muy altos, lo que se traduce en un modelo de detección de objetos muy eficaz y fiable. Es capaz de identificar la gran mayoría de los objetos que debería detectar y, al mismo tiempo, las detecciones que realiza son muy pocas veces erróneas.

✓ 1. Descargar el Modelo Entrenado (best.pt)

Este código permite descargar el archivo de pesos del modelo que tuvo el mejor rendimiento durante el entrenamiento (best.pt).

```
from google.colab import files
import os

# Define la ruta al archivo de pesos del mejor modelo.
# Asegúrate de que esta ruta coincida con el nombre de tu ejecución de entrenamiento.
# En tu caso, el nombre de la ejecución fue "yolov8n_unknown_train" dentro de "runs/detect".
model_path = "/content/runs/detect/resultados/weights/best.pt"

# Verifica si el archivo existe antes de intentar descargarlo
if os.path.exists(model_path):
    print(f"Descargando el modelo: {model_path}")
    # Usa la función 'files.download' de Google Colab para iniciar la descarga.
    files.download(model_path)
    print("¡Descarga completada!")
else:
    print(f"Error: El archivo del modelo no se encontró en {model_path}")
    print("Asegúrate de que la ruta sea correcta y que el entrenamiento haya finalizado.")
```



Descargando el modelo: /content/runs/detect/resultados/weights/best.pt
¡Descarga completada!

✓ 2. Guardar la Carpeta runs en Google Drive

La carpeta runs contiene no solo los pesos del modelo, sino también los gráficos de entrenamiento (pérdidas, mAP, etc.), los registros, y otros archivos importantes generados durante el proceso.

```
import shutil
import os
```

```
# Define la ruta de la carpeta 'runs' que quieres guardar.
# Esta carpeta contiene todos los resultados de tus entrenamientos YOLOv8.
source_runs_dir = "/content/runs"

# Define la ruta de destino en tu Google Drive.
# Recomiendo crear una subcarpeta para organizar tus experimentos.
# Por ejemplo, puedes querer guardarlos en una carpeta llamada 'yolov8_experiments' en tu Drive.
destination_drive_dir = "/content/drive/MyDrive/CurEsplABD_ProyectoFinal_IvánFalcónMonzón"

# Asegúrate de que el directorio de destino en Drive exista
os.makedirs(destination_drive_dir, exist_ok=True)

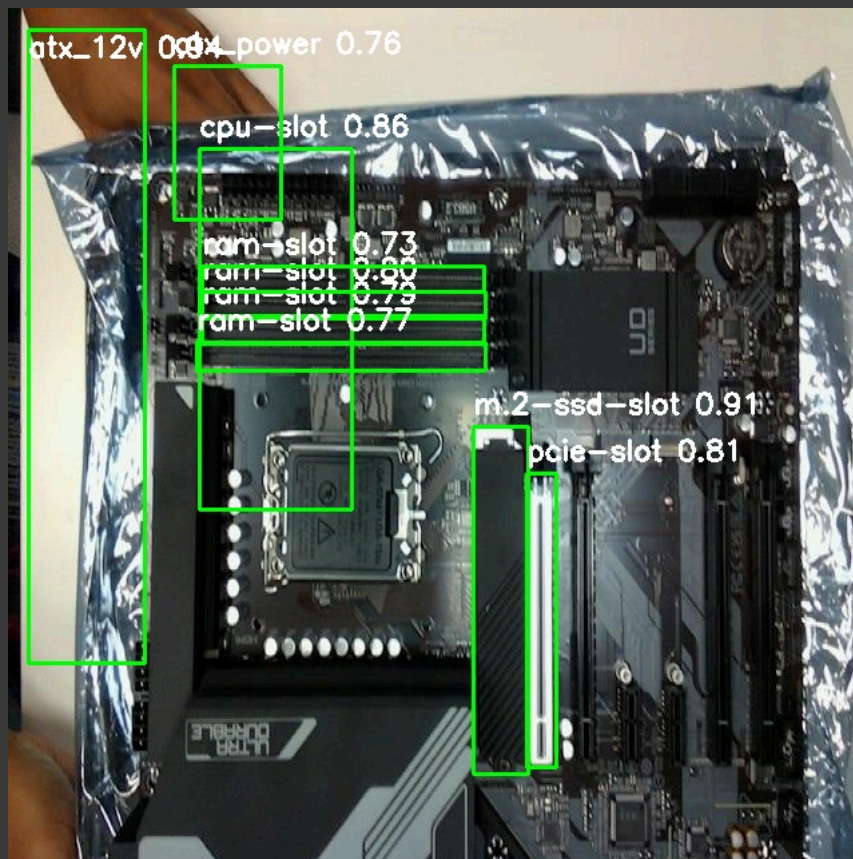
print(f"Copiando la carpeta '{source_runs_dir}' a '{destination_drive_dir}'...")

try:
    # Copia el contenido de la carpeta 'runs' al destino en Google Drive.
    # 'shutil.copytree' copia un directorio completo, incluyendo su contenido.
    # 'dirs_exist_ok=True' permite sobrescribir directorios si ya existen (útil para actualizar resultados).
    shutil.copytree(source_runs_dir, destination_drive_dir, dirs_exist_ok=True)
    print("¡Copia de la carpeta 'runs' a Google Drive completada!")
except Exception as e:
    print(f"Error al copiar la carpeta: {e}")
    print("Verifica que Google Drive esté montado y que las rutas sean correctas.")
```

↗ Copiando la carpeta '/content/runs' a '/content/drive/MyDrive/CurEsplABD_ProyectoFinal_IvánFalcónMonzón'...

¡Copia de la carpeta 'runs' a Google Drive completada!

✓ Otros Resultados Obtenidos



Repositorios

- Dataset inicial: <https://universe.roboflow.com/gradresearch/gradresearch>
- Google Colab: <https://colab.research.google.com/drive/11X6-vMe4TnmLKkDWmBzcQz4uRGPijMRp?usp=sharing>
- Google Drive: https://drive.google.com/drive/folders/1Xzf3wejBUWDtcBH4_ekg6cCaAUSaqsC?usp=sharing
- Github: https://github.com/IvanFalconMonzon/CurEsplABD_ProyectoFinal_IvanFalconMonzon.git
- Presentación: https://www.canva.com/design/DAGoFF8b7GU/6S5vJ4ckhcu-QaonLDRpg/edit?utm_content=DAGoFF8b7GU&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

