PROYECTO JUGADOR DE DOBBLE

Visión y Percepción Automáticas

Autores:

Iván Fernández Gómez

Elena Pérez Baena

23 de enero de 2024

Índice

Índice de figuras

Índice de cuadros

1. Introducción

En este proyecto, se busca implementar un jugador automático para el juego Dobble mediante el la visión artificial. El objetivo principal es emplear técnicas de visión por computadora y aprendizaje profundo para entrenar un modelo capaz de identificar parejas de cartas en imágenes. Se ha empleado la arquitectura YOLO-NAS para llevar a cabo este propósito.
El juego de Dobble como contexto para este proyecto se basa en su complejidad visual, ya que cada carta posee múltiples símbolos y se busca la identificación de la pareja correcta en un tiempo reducido. La implementación de un modelo preciso para este juego no solo presenta desafíos tecnológicos, sino que también permite explorar aplicaciones prácticas de la visión por computadora en el ámbito de los juegos de mesa.
A lo largo de este trabajo, se describe el proceso de construcción y entrenamiento del modelo, se analizan los resultados obtenidos y se discuten los desafíos enfrentados durante el desarrollo del proyecto.

Finalmente, se presentarán las conclusiones derivadas de este proyecto, destacando los logros alcanzados, las limitaciones identificadas. Además, se incluirá un anexo que contiene todos los códigos utilizados en este proyecto.

2. Dobble

En 1976, Jacques Cottereau dio vida a la idea del juego Dobble al generalizar un famoso rompecabezas matemático. Este acertijo planteaba el desafío de organizar a 15 alumnas en filas de tres durante siete días consecutivos, de modo que al final de la semana ninguna hubiera caminado junto a otra más de una vez. La solución involucraba la creación de lo que se conoce como "bloques equilibrados incompletos", una estructura matemática clave.

Motivado por este logro, Cottereau desarrolló dos juegos basados en estas estructuras, que fueron publicados en revistas especializadas. El primero, llamado "Strange retriever", no alcanzó gran popularidad. Sin embargo, el segundo, conocido como ."El juego de los insectos", sentó las bases para lo que eventualmente se convertiría en el juego Dobble. Esta versión temprana del Dobble consistía en identificar la imagen de un insecto común entre dos cartas.

El renacer del Dobble tuvo lugar en la primavera de 2008, cuando Denis Blanchot descubrió el juego y se asoció con Cottereau para convertirlo en un juego de éxito. Durante el proceso de desarrollo, se centraron en la creación de iconos que permitieran una identificación rápida y fueran fácilmente comprensibles, especialmente para niños.
Tras varios prototipos y pruebas, Blanchot contactó con editoriales, y con la colaboración de Play Factory, el juego Dobble finalmente vio la luz em 2009. Desde entonces, se ha convertido en un fenómeno mundial, cautivando a jugadores de todas las edades con su simplicidad y diversión.

El Dobble, en su versión actual, se compone de un conjunto de cartas circulares, cada una compuesta por diversas imágenes. La característica distintiva del juego radica en que cada par de cartas comparte exactamente un símbolo común, lo que plantea a los jugadores el desafío de identificarlo con rapidez y precisión. Aunque existen diversas modalidades de juego, nuestro enfoque en este proyecto de visión artificial se centra en desarrollar un jugador automático para Dobble.

En esta implementación, el juego se simplifica a la extracción de dos cartas del mazo, mostrándolas al ordenador. La tarea es que, mediante una función, el sistema pueda identificar de manera eficaz cuál es el símbolo que se repite en ambas cartas, proporcionando no sólo su nombre de clase sino también su ubicación, es decir, su bounding box. Este desafío fusiona la emoción del Dobble con la capacidad analítica de la inteligencia artificial, abriendo la puerta a una nueva dimensión de juego interactivo y automatizado.

En nuestro proyecto, comenzaremos trabajando con la edición de animales de las cartas proporcionadas, como se ilustra en la Figura 1, donde cada carta presenta seis clases únicas y, en total, contamos con un conjunto de 32 clases distintas. Posteriormente, con el objetivo de evaluar y comparar los resultados obtenidos, ampliaremos nuestra análisis a otra clase de cartas, en concreto con la edición Disney, cuyas representaciones se presentan en la Figura 2. En este conjunto, cada carta alberga diez clases únicas, lo que suma un total de 91 clases en el conjunto completo.

Figura 1: Cartas de la edición animales

Figura 2: Cartas de la edición Disney

3. Tratamiento de datos

Para crear la base de datos, se ha utilizado la imagen proporcionada que contiene varias tarjetas de la versión del juego de animales. Se ha segmentado cada tarjeta individual en imágenes separadas, generando así un total de 21 imágenes. Como en este caso, tenemos un conjunto inicial limitado a tan sólo 21 imágenes de las tarjetas del juego de animales, la capacidad de aprendizaje de nuestra red neuronal sería prácticamente nula, ya que apenas tiene datos de los que aprender. Por tanto, es esencial generar un conjunto de datos más extenso y diverso para permitir que la red neural capture de manera efectiva la variabilidad con la que pueden aparecer las tarjetas del juego. Por ello, se ha aplicado el método conocido como 'data augmentation'.

Esta técnica es aquella que amplía un conjunto de datos mediante la aplicación de transformaciones aleatorias. La utilidad de esta técnica radica en mejorar el rendimiento de los modelos de machine learning, especialmente en el ámbito de la visión por computadora, como es nuestro caso. La necesidad de un conjunto de datos amplio es crucial para entrenar de manera efectiva las redes neuronales, y la generación de más datos a través de 'data augmentation' contribuye significativamente a este objetivo.

A continuación, se detalla el código desarrollado para implementar la técnica de 'data augmentation' en nuestras imágenes iniciales. El código completo se encuentra disponible en el Anexo 1.

El primer paso consiste en la instalación de todas las librerias necesarias para poder hacer uso de este método, así como descomprimir un archivo .zip donde se incluyen las imágenes originales. Además, estas se guardarán en el directorio, en una carpeta que indica 'images' y dentro de esta, en una que se denomina 'Tarjetas' y se obtendrán todos los nombres de estas para poder recorrerlas y realizar las transformaciones.

```
import zipfile
import os
import shutil
import cv2
import numpy as np
from keras.preprocessing.image import ImageDataGenerator, load_img, img_to_array
import matplotlib.pyplot as plt
from scipy.spatial import distance
import random

zip_ref = zipfile.ZipFile('Tarjetas.zip', 'r')
zip_ref.extractall('images')
zip_ref.close()

imagenes = os.listdir('images/Tarjetas')
```

A continuación, definiremos tanto una función para calcular el histograma de color de una imagen, además que normaliza y aplana el histograma antes de devolverlo, así como la configuración del generador de imágenes. Este realiza diversas transformaciones, como rotación, desplazamiento horizontal y vertical, cizalladura, zoom, volteo horizontal y cambios en la intensidad de brillo (pudiéndose cambiar según lo que necesite el usuario). También, se almacenarán estos para poder revisar si las imagenes generadas presentan el mismo histograma que las ya creadas, pudiendo así verificar si las imagen generada ya existe y crear una diferente hasta que no se repita en todas las imagenes existentes.

```
def calculate_histogram(image, bins=(8, 8, 8)):
    hist = cv2.calcHist([image], [0, 1, 2], None, bins, [0, 256, 0, 256, 0, 256])
    hist = cv2.normalize(hist, hist).flatten()
    return hist

datos de imagen
datagen = ImageDataGenerator(
    rotation_range=360,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    brightness_range=[0.5,1.0])
```

Entonces, ahora se ejecutará el código que generará las imágenes. Por lo tanto, se iterará en todas las imagenes originales y convierte cada imagen a un array y le da forma para que tenga una dimensión adicional que representa el tamaño del lote (batch size). Siguiendo el código, compara el histograma de la imagen aumentada con los histogramas de las imágenes ya generadas. Si la distancia entre el histograma actual y los histogramas existentes es menor que 0.1 (umbral arbitrario), se considera que las imágenes son muy similares y no se almacena. Si no se encuentra ninguna imagen similar, se almacena el histograma y se muestra la imagen aumentada en una figura. Además, se van mostrando las imagenes para intentar verificar que las imagenes están bien transformadas.

```
histograms = []
i = 0
for imagen in imagenes:
    img = load_img('images/Tarjetas/' + imagen)
    data = img_to_array(img)
    data = data.reshape((1,) + data.shape)

    for batch in datagen.flow(data, batch_size=1, save_to_dir='images/Tarjetas_New',
        save_prefix='Tarjeta', save_format='png'):
        histogram = calculate_histogram(batch[0].astype('uint8'))

        for hist in histograms:
            if cv2.compareHist(hist, histogram, cv2.HISTCMP_BHATTACHARYYA) < 0.1:
```

```
                break
        else:
            histograms.append(histogram)
            plt.figure(i)
            imgplot = plt.imshow(batch[0].astype('uint8'))
            i += 1
            if i % 20 == 0:
                break

plt.show()
```

Finalmente, para verificar que al menos se han generado 600 imagenes, se realiza un bucle para generar suficientes imagenes para poder satisfacerlo. El código en sí tiene el mismo procedimiento que el anterior.

```
while len(os.listdir('images/Tarjetas_New')) < 600:
    imagen = random.choice(imagenes)

    img = load_img('images/Tarjetas/' + imagen)
    data = img_to_array(img)
    data = data.reshape((1,) + data.shape)

    for batch in datagen.flow(data, batch_size=1, save_to_dir='images/Tarjetas_New',
        save_prefix='Tarjeta', save_format='png'):
        histogram = calculate_histogram(batch[0].astype('uint8'))

        for hist in histograms:
            if cv2.compareHist(hist, histogram, cv2.HISTCMP_BHATTACHARYYA) < 0.1:
                break
        else:
            histograms.append(histogram)
            plt.figure(i)
            imgplot = plt.imshow(batch[0].astype('uint8'))  # Mostrar la imagen
                transformada
            i += 1
            if i % 2 == 0:
                break

    plt.show()
```

Este código genera un total de 3.234 imágenes nuevas. Dado que ahora es necesario etiquetar manualmente todas las clases de animales presentes en las tarjetas, que suman 31 clases en total, se ha optado por seleccionar aleatoriamente 528 imágenes para su etiquetado. Para simplificar y agilizar este proceso, se han empleado la herramienta Roboflow.

Roboflow es una plataforma que proporciona herramientas y servicios diseñados para simplificar el trabajo con imágenes y modelos de aprendizaje profundo, centrándose principalmente en la preparación de datos para tareas de visión por computadora. En nuestro

proyecto, se ha creado un proyecto en Roboflow donde se han cargado todas las imágenes generadas. De este conjunto, se han seleccionado aleatoriamente 528 imágenes para su etiquetado.

Esta plataforma facilita el proceso de etiquetado al permitir la creación rápida de bounding boxes y su correspondiente asignación etiquetas de manera eficiente. Posteriormente, con las imágenes de las tarjetas etiquetadas, se ha generado nuestro dataset. Para ello, aplicamos un preprocesamiento que incluye la auto-orientación y reescalado de las imágenes a 640x640 píxeles. Además, la herramienta nos brinda la capacidad de aplicar otra vez 'data augmentation' adicionalmente.

Como resultado, se han creado dos versiones del dataset. La primera versión contiene las 528 imágenes etiquetadas, sin aplicar data augmentation adicional. La segunda versión incluye las mismas imágenes ya etiquetadas, pero con la aplicación de la técnica de data augmentation proporcionada por Roboflow. En esta última versión, obtenemos un conjunto ampliado con un total de 1.268 imágenes, mejorando así la diversidad y robustez de nuestro conjunto de datos para el entrenamiento de modelos de visión por computadora.

Adicionalmente, la herramienta ofrece la facilidad de realizar la división de la base de datos en conjuntos de entrenamiento, validación y prueba. En esta configuración, hemos seguido la recomendación estándar de asignar el 70 % de las imágenes al conjunto de entrenamiento, el 20 % al conjunto de validación y el 10 % al conjunto de test para la primera versión. En cuanto a la segunda versión, se optó por una distribución del 88 % para el conjunto de entrenamiento, el 8 % para el conjunto de validación y el 4 % para el conjunto de test, para manterner un mayor número de imágenes en la fase de entrenamiento. Los detalles de la distribución resultante del número de imágenes en cada conjunto para ambas versiones del dataset se encuentran en la Tabla 1 a continuación.

	Versión 1	**Versión 2**
Entrenamiento	370	1110
Validación	106	106
Test	52	52
Total	520 imágenes	1.268 imágenes

Cuadro 1: Distribución de imágenes de animales en conjuntos para ambas versiones del dataset.

Finalmente, la herramienta ofrece la posibilidad de exportar las variantes del dataset en formato .zip o en forma de código. En la integración con nuestro código de entrenamiento para el modelo YOLO-NAS, se ha seleccionado la opción de exportar el dataset en el formato YOLO v5 PyTorch. Este formato resulta compatible con la estructura requerida para la división de datos en el modelo YOLO-NAS. A continuación, se presenta la estructura de código generada.

```
!pip install roboflow

from roboflow import Roboflow
rf = Roboflow(api_key='********************')
project = rf.workspace('dobble-fm3a4').project('dobble_bb_rect')
dataset = project.version(@yg@1@).download('yolov5')
```

Todo este proceso será repetido de la misma manera, para la versión de los personajes Disney.

4. Yolo-NAS

YOLO (You Only Look Once) es una técnica ampliamente utilizada en detección de objetos en visión por computadora. Desarrollado en 2015 por Joseph Redmon, Santosh Divvala, Ross Girshick y Ali Farhadi, YOLO destaca por su capacidad para detectar objetos en tiempo real con precisión y velocidad excepcionales. A diferencia de otros algoritmos, YOLO utiliza una única red neuronal convolucional para predecir tanto la clase como la ubicación de los objetos en una imagen, lo que le confiere eficiencia superior frente a métodos de múltiples etapas.

El algoritmo se construye mediante el empleo de una red neuronal convolucional (CNN) entrenada con un conjunto de datos etiquetados. Este conjunto consiste en imágenes con etiquetas que indican la ubicación (bounding box) y la clase de los objetos presentes.

La CNN se divide en dos componentes principales: extracción de características y detección de objetos. La primera utiliza capas convolucionales para extraer características distintivas, generando un mapa utilizado en la fase de detección.

En la fase de detección, el mapa se usa para identificar objetos en la imagen, dividiéndola en una cuadrícula de celdas. Se predice la probabilidad de que un objeto esté presente en cada celda y, en aquellas con probabilidad significativa, se hacen predicciones sobre la ubicación y clase del objeto.

Después del entrenamiento con el conjunto de datos etiquetados, la CNN se utiliza para detectar objetos en nuevas imágenes, devolviendo una lista con bounding box y clase.

YOLO presenta la peculiaridad de tener un preentrenamiento inicializando una red clasificadora con las primeras capas convolucionales. Esta red se entrena en un conjunto grande y diverso, como ImageNet, para reconocer características visuales. Los pesos aprendidos se transfieren a la red YOLO, mejorando su rendimiento, especialmente con pocos datos de entrenamiento.

Desde la publicación del primer YOLO en 2015, se han desarrollado varias versiones avanzadas del algoritmo. Las más detacadas estos últimos años son YOLOv5, YOLOv8 y YOLO-NAS.

YOLOv5, desarrollado por Ultralytics, ha ganado popularidad por su arquitectura modificada y su capacidad para adaptarse a diferentes requisitos de hardware. Ofrece características avanzadas, incluyendo mejoras en la estabilidad de los bounding boxes y diversas técnicas de aumento de datos.

YOLOv8, también desarrollado por Ultralytics, comparte similitudes con YOLOv5. Adopta un modelo para procesar tareas de detección de objetos de manera más eficiente. Utiliza funciones de activación como sigmoid y softmax para puntuaciones de objectness y probabilidades de clases, respectivamente.

Por otra parte, YOLO-NAS, lanzado por Deci en mayo de 2023, se destaca por su enfoque en la detección de objetos pequeños, mejora la precisión de localización y eficiencia en el rendimiento por cálculo, siendo adecuado para aplicaciones en tiempo real. Introduce módulos de cuantificación consciente (QSP y QCI) para minimizar la pérdida de precisión durante la cuantificación post-entrenamiento. Su arquitectura se logra automáticamente mediante AutoNAC, una tecnología de búsqueda de arquitectura neural patentada, y ofrece tres versiones (YOLO-NASS, YOLO-NASM y YOLO-NASL) que varían en profundidad y posición de bloques cuantitativos, siendo sus nombres una referencia a Small, Medium y Large. La fase de preentrenamiento incluye datos automáticamente etiquetados, auto-diseño, y grandes conjuntos de datos.

En este proyecto, se ha optado por entrenar el modelo YOLO-NASL, una variante especializada de YOLO-NAS. Esta elección se fundamenta en su enfoque particular hacia la detección de objetos pequeños, así como en las mejoras que presenta en la precisión de los bounding boxes y su eficiencia en el rendimiento. La comparativa entre estas tres arquitecturas se puede apreciar en la Figura 3.

Además, YOLO-NAS, al igual que otros modelos de detección de objetos, puede beneficiarse de un proceso de preentrenamiento mediante el conjunto de datos COCO (Common Objects in Context). COCO es un extenso conjunto de datos que engloba más de 330,000 imágenes, todas ellas anotadas para realizar tareas de detección de objetos, segmentación y descripción.

La etapa de preentrenamiento en COCO proporciona a YOLO-NAS la capacidad de aprender a reconocer una amplia variedad de categorías de objetos antes de ser ajustado con un conjunto de datos específico. Este proceso de preentrenamiento emerge como una estrategia eficaz para mejorar considerablemente el rendimiento del modelo, especialmente en situaciones donde la disponibilidad de datos de entrenamiento es limitada.

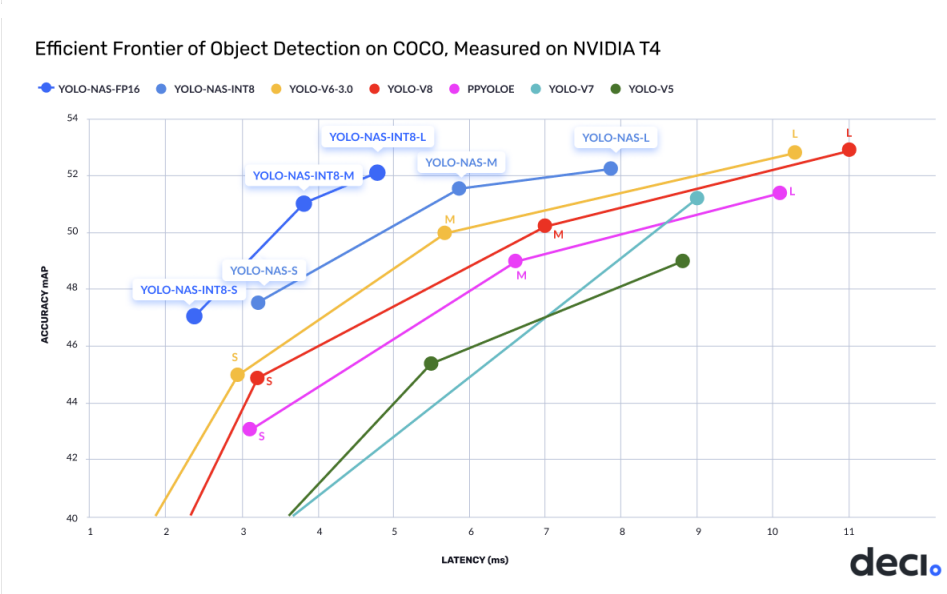

Figura 3: Comparativa entre las diferentes versiones de YOLO

A continuación, se describe detalladamente el proceso paso a paso para entrenar y probar la red neuronal YOLO-NAS y obtener el modelo que clasifica los animales de las cartas del juego Dobble. Es importante saber que dada las librerías usadas por el código y el costo computacional que supone el entrenamiento, este código debe ejecutarse en Google Colab.

Primero, se elige el directorio de trabajo y se procede a eliminar cualquier archivo o subdirectorio presente en él. En caso de que el directorio previo no existiera, se informa de esta situación en la pantalla. Esto evita una acumulación de archivos dentro del directorio de trabajo.

```
import os
import shutil

directorio = '/content/Dobble_BB_rect-5'

if os.path.exists(directorio):
    shutil.rmtree(directorio)
else:
    print(f'El directorio {directorio} no existe')
```

Posteriormente, se debe aplicar un entorno de ejecución T4 GPU o A100 si es posible, para de esta manera trabajar con una GPU de NVIDIA. Podemos comprobar que los cambios se han aplicado correctamente con el siguiente código, donde pedimos primero que muestre la información sobre la GPU usada, y después cuántos GB de RAM se están usando para el programa.

```
gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
if gpu_info.find('failed') >= 0:
  print('Not connected to a GPU')
else:
  print(gpu_info)

from psutil import virtual_memory
ram_gb = virtual_memory().total / 1e9
print('Your runtime has {:.1f} gigabytes of available RAM\n'.format(ram_gb))

if ram_gb < 20:
  print('Not using a high-RAM runtime')
else:
  print('You are using a high-RAM runtime!')
```

Después se descargan todas las librerías que van a ser necesarias durante la ejecución.

```
!pip install -q git+https://github.com/Deci-AI/super-gradients.git
```

```
!pip install -q numpy
!pip install -q supervision
!pip install -q roboflow
!pip install -q supervision
!pip install -q torch
!pip install -q super-gradients==3.5.0.
```

Posteriormente, se debe reiniciar la sesión para que se apliquen las librerías intaladas. Comprobamos que esto se ha hecho con éxito comprobando la versión de super-gradients con el siguiente comando.

```
import super_gradients
print(super_gradients.__version__)
```

Después, para que resulte más fácil encontrar las carpetas que se crean dentro de Google Colab, se crea un directorio *HOME* y se entra en él con las siguientes líneas de código.

```
import os
HOME = os.getcwd()
print(HOME)

%cd {HOME}
```

A continuación, importamos el dataset creado en Roboflow, tal y como se mostró anteriormente. Además, se copian y se muestran el directorio en el que este se ubica y y las clases que contiene. Seguidamente, con estos datos definimos cómo están ordenador los parámetros del dataset.

```
from roboflow import Roboflow
rf = Roboflow(api_key='**********************')
project = rf.workspace('dobble-fm3a4').project('dobble_bb_rect')
dataset = project.version(1).download('yolov5')

LOCATION = dataset.location
print("location:", LOCATION)
CLASSES = sorted(project.classes.keys())
print("classes:", CLASSES)

dataset_params = {
    'data_dir': LOCATION,
    'train_images_dir':'train/images',
    'train_labels_dir':'train/labels',
    'val_images_dir':'valid/images',
    'val_labels_dir':'valid/labels',
    'test_images_dir':'test/images',
```

```
    'test_labels_dir':'test/labels',
    'classes': CLASSES
}
```

Continuamos configurando el entrenamiento del modelo. En este fragmento de código, se especifica la arquitectura como YOLO-NASL, el *batchsize*, o tamaño del lote, que representa el número de muestras de entrenamiento utilizadas en una iteración. A mayor *batchsize*, el entrenamiento es más rápido, pero también requiere más carga computacional. En nuestro caso, lo hemos establecido en 8, ya que con un valor mayor, el entrenamiento no es posible.

También se establece *Max_epochs* como el número máximo de épocas de entrenamiento. Una época se refiere a un pase completo a través de todo el conjunto de datos de entrenamiento. El proceso de entrenamiento se detendrá después de alcanzar este número especificado de épocas.

Además, se define el directorio donde se guardan los checkpoints, que son instantáneas del modelo en diferentes puntos del entrenamiento, permitiendo retomar el proceso en caso de interrupciones. Finalmente, hay una línea de código que convierte el nombre del proyecto a minúsculas y reemplaza los espacios con guiones bajos. Esto se realiza para evitar problemas de nombrado del proyecto.

Finalmente, se importa la clase *Trainer* del módulo *'super_gradients.training'* , que gestiona el proceso de entrenamiento, incluyendo el manejo de checkpoints y el registro.

```
MODEL_ARCH = 'yolo_nas_l'
BATCH_SIZE = 32
MAX_EPOCHS = 250
CHECKPOINT_DIR = f'{HOME}/checkpoints'
EXPERIMENT_NAME = project.name.lower().replace(" ", "_")

from super_gradients.training import Trainer

trainer = Trainer(experiment_name=EXPERIMENT_NAME, ckpt_root_dir=CHECKPOINT_DIR)
```

La parte de código con la que se sigue, es con la carga y división de los datos de entrenamiento, de validación y de test, del modelo YOLO-NAS en el formato de detección de objetos COCO. En cada división del conjunto de datos, se proporcionan sus respectivos parámetros específicos, como la ubicación del directorio de datos, el directorio de imágenes, el directorio de etiquetas y las clases. Además, se especifican parámetros del cargador de datos, como el tamaño del lote y el número de trabajadores para la carga de datos paralela.

```
from super_gradients.training.dataloaders.dataloaders import (
    coco_detection_yolo_format_train, coco_detection_yolo_format_val)

train_data = coco_detection_yolo_format_train(
```

```
    dataset_params={
        'data_dir': dataset_params['data_dir'],
        'images_dir': dataset_params['train_images_dir'],
        'labels_dir': dataset_params['train_labels_dir'],
        'classes': dataset_params['classes']
    },
    dataloader_params={
        'batch_size': BATCH_SIZE,
        'num_workers': 2
    }
)

val_data = coco_detection_yolo_format_val(
    dataset_params={
        'data_dir': dataset_params['data_dir'],
        'images_dir': dataset_params['val_images_dir'],
        'labels_dir': dataset_params['val_labels_dir'],
        'classes': dataset_params['classes']
    },
    dataloader_params={
        'batch_size': BATCH_SIZE,
        'num_workers': 2
    }
)

test_data = coco_detection_yolo_format_val(
    dataset_params={
        'data_dir': dataset_params['data_dir'],
        'images_dir': dataset_params['test_images_dir'],
        'labels_dir': dataset_params['test_labels_dir'],
        'classes': dataset_params['classes']
    },
    dataloader_params={
        'batch_size': BATCH_SIZE,
        'num_workers': 2
    }
)
```

A continuación, se configuran los hiperparámetros del entrenamiento. La primera línea hace una inspección del dataset y muestra información relevante sobre las transformaciones aplicadas al conjunto de datos de entrenamiento, como si las imágenes generadas deben contener al menos un objeto, las dimensiones de entrada de las imágenes o la probabilidad de aplicar algunas transformaciones.

```
train_data.dataset.transforms
```

Después, se carga el modelo de detección de objetos. Utiliza la función *get* del módulo models de *super_gradients.training*, pasando la arquitectura del modelo, el número de clases en el conjunto de datos y los pesos preentrenados en el conjunto de datos COCO.

```
from super_gradients.training import models

model = models.get(
    MODEL_ARCH,
    num_classes=len(dataset_params['classes']),
    pretrained_weights="coco"
)
```

Posteriormente, se define la función de pérdida que se utilizará durante el entrenamiento. En este caso, se utiliza PPYoloELoss. Además, configura las métricas que se calcularán en el conjunto de validación y se proporciona un callback de post-predicción que realiza acciones después de realizar las predicciones.

```
from super_gradients.training.losses import PPYoloELoss
from super_gradients.training.metrics import DetectionMetrics_050
from super_gradients.training.models.detection_models.pp_yolo_e import
    PPYoloEPostPredictionCallback
```

En este trozo se configuran varios parámetros generales de entrenamiento, como el modo silencioso, el uso de precisión mixta, la tasa de aprendizaje inicial, el optimizador, etc. Estos parámetros afectan cómo se lleva a cabo el entrenamiento del modelo.

```
train_params = {
    'silent_mode': False,
    "average_best_models":True,
    "warmup_mode": "linear_epoch_step",
    "warmup_initial_lr": 1e-6,
    "lr_warmup_epochs": 3,
    "initial_lr": 5e-4,
    "lr_mode": "cosine",
    "cosine_final_lr_ratio": 0.1,
    "optimizer": "Adam",
    "optimizer_params": {"weight_decay": 0.0001},
    "zero_weight_decay_on_bias_and_bn": True,
    "ema": True,
    "ema_params": {"decay": 0.9, "decay_type": "threshold"},
    "max_epochs": MAX_EPOCHS,
    "mixed_precision": False,
    "loss": PPYoloELoss(
        use_static_assigner=False,
        num_classes=len(dataset_params['classes']),
        reg_max=16
```

```
    ),
    "valid_metrics_list": [
        DetectionMetrics_050(
            score_thres=0.1,
            top_k_predictions=300,
            num_cls=len(dataset_params['classes']),
            normalize_targets=True,
            post_prediction_callback=PPYoloEPostPredictionCallback(
                score_threshold=0.01,
                nms_top_k=1000,
                max_predictions=300,
                nms_threshold=0.7
            )
        )
    ],
    "metric_to_watch": 'mAP@0.50'
}
```

Ya que está todo definido, se inicia el proceso de entrenamiento del modelo según los datos configurados.

```
trainer.train(
    model=model,
    training_params=train_params,
    train_loader=train_data,
    valid_loader=val_data
)
```

Según la configuración del modelo el entrenamiento puede tardar más o menos. Y al terminar la fase de entrenamiento, se guardan los resultados y los checkpoints del modelo en una carpeta .zip. Ya que esta carpeta tiene un peso de más de 2GB, la subiremos a Drive, para poder descargarla de forma más sencilla. Posteriormente se muestran los resultados del entrenamiento del modelo en TensorFlow.

```
%load_ext tensorboard
%tensorboard --logdir {CHECKPOINT_DIR}/{EXPERIMENT_NAME}

import locale
locale.getpreferredencoding = lambda: "UTF-8"

!zip -r yolo_nas.zip {CHECKPOINT_DIR}/{EXPERIMENT_NAME}

from google.colab import drive
import shutil

drive.mount('/content/drive')
```

```
archivo_zip_colab = '/content/yolo_nas.zip'
ruta_destino_drive = '/content/drive/MyDrive/yolo_nas.zip'
shutil.copy(archivo_zip_colab, ruta_destino_drive)
```

5. Función *Jugador*

Una vez entrenado el modelo con una base de datos de Roboflow en Google Colab, se realizó una función llamada "Jugador", con la que se pretende a la vez que probar este modelo, interactuar con el usuario introduciendo una imagen que contenga las dos cartas, y esta función devolverá tanto en formato .jpg, como por pantalla la imagen con las anotaciones correspondientes con el resultado de la detección.

En esta detección y el muestreo del resultado de este jugador, se considerarán diferentes anotaciones:

- Mostrar el bounding box de las dos figuras detectadas como resultado ganador del juego encima de estas en la imagen.
- Señalar el nombre de la clase que corresponde a la detección, pudiendo verificar que el jugador detectó correctamente según su entrenamiento.
- Indicar el porcentaje de fiabilidad con el que indica que esa figura corresponde a una clase. Así, al igual que en el caso anterior, se puede observar cuando de seguro está el jugador de que esa figura pertenece a esa clase.

Como última consideración a mencionar, como el modelo no presenta una precisión del 100 % a la hora de jugar, se nos plantean diferentes casuísticas a la hora de empezar el juego, como puede ser una falsa detección que acaba resultando en una falsa pareja, tanto en ambas cartas como en la misma.

Por ello, se ha realizado un arreglo para que acabe mostrando únicamente una pareja si es que la detecta. Ahora se dispone a la explicación del código.

5.1. Construcción del modelo importado

Inicialmente, fuera de la función de *Jugador*, se hace un inicio del modelo, ya que si se pretende exportar del entrenamiento desde el entorno donde se ha realizado (Google Colab)y volver a cargarse en un fichero a parte, se necesita una reconfiguración con diferentes elementos del entrenamiento. El código que se mostrará más adelante, corresponde a los ficheros .ipynb realizados en Google Colab.

Primero se han de instalar las librerias necesarias e importar estas y las funciones específicas de estas.

```
!pip install -q git+https://github.com/Deci-AI/super-gradients.git
!pip install -q numpy
!pip install -q supervision
!pip install -q roboflow
!pip install -q supervision
!pip install -q torch
!pip install -q super-gradients==3.5.0.
```

```

from google.colab import drive
import zipfile
import os
from roboflow import Roboflow
import torch
import super_gradients
from super_gradients.training import models
```

Más tarde, se procederá a importar el modelo desde Google Drive, ya que en el entorno de trabjo, se encontró la limitación del espacio máximo de subida de archivos (limitado a 25MB, necesitando para cargar el modelo de ejemplo unos 2.3GB). Entonces se accederá al directorio de Drive, se descargará y se descomprimirá para poder acceder a este más tarde. El directorio indicado en la variable "file_path"debe ser cambiado según el usuario y donde se haya guardado.

```
drive.mount('/content/drive')

file_id = 'yolo_nas_Disney.zip'
file_path = '/content/drive/MyDrive/' + file_id

with zipfile.ZipFile(file_path, 'r') as zip_ref:
    zip_ref.extractall("/content/")
```

A continuación, estas líneas de código son utilizadas en Google Colab para imprimir y cambiar el directorio de trabajo. Sirven para organizar y acceder a los archivos de nuestro cuaderno. Además se cargará la base de datos con la que fue entrenado el modelo, para dar datos de referncia como lo son las clases de la base de datos y la localización del dataset.

```
HOME = os.getcwd()
print(HOME)
%cd {HOME}

rf = Roboflow(api_key='********************')
project = rf.workspace('dobble-fm3a4').project('dobble_bb_rect')
dataset = project.version(1).download('yolov5')
```

Se preparan estos parámetros para la construcción del modelo, y así poder utilizar este pre-entrenado, indicando también la arquitectura del modelo además de otros parámetros para poder construir este.

```
LOCATION = dataset.location
print("location:", LOCATION)
CLASSES = sorted(project.classes.keys())
print("classes:", CLASSES)
```

```
5
6  dataset_params = {
7      'data_dir': LOCATION,
8      'classes': CLASSES
9  }
10
11 MODEL_ARCH = 'yolo_nas_l'
12 CHECKPOINT_DIR = f'{HOME}/checkpoints'
13 EXPERIMENT_NAME = project.name.lower().replace(" ", "_")
```

Por último, se define el modelo con todos los parámetros obtenidos anteriormente y una vez construido este a partir del entrenamiento realizado en cualquier momento para una base de datos con sus clases, está preparadao para utilizarse. Se deberá cambiar la variable 'checkpoint_path' dependiendo de como se haya guardado el modelo, principalmente el número de RUN. Además se podrá elegir el modelo obtenido del entrenamiento, teniendo 3 posibilidades cambiando el último elemento del directorio a:

1. average_model.pth
2. ckpt_best.pth
3. ckpt_latest.pth

```
1  DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
2
3  best_model = models.get(
4      MODEL_ARCH,
5      num_classes=len(dataset_params['classes']),    checkpoint_path=f"/content/content/
           checkpoints/dobble_bb_rect/RUN_20240121_120128_043707/ckpt_best.pth"
6  ).to(DEVICE)
```

5.2. Construcción de la función

Una vez configurado nuestro jugador para el dataset del entrenamiento, se mostrará la función que nos permite interactuar con el modelo entrenado y comprobar sus detecciones y con cuanta precisión las realiza. En este punto, se han tomado diferentes decisiones teniendo en cuenta las posibilidades mencionadas al principio de la sección.

Para empezar se debe tener clara la definición de la función, las librerias necesarias para su ejecución, así como su nombre como los parámetros de entrada necesarios. Como se ve en el código adjuntado abajo, la función se denomina como *jugador* y esta tiene como entradas:

- Una imagen, que para este caso se espera que incluya las dos tarjetas (preferiblemente en formato .jpg)
- Las clases que presenta el dataset, pudiéndose adquirir de este o introducirlas manualmente como una lista con los nombres ordenados en orden alfabético.

- El modelo con el que se va a jugar. En este caso, se define anteriormente.

```
import supervision as sv
import cv2
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from collections import Counter
import time


def jugador(imagen, clases, modelo):
```

Siguiendo con la función, ahora se deberá detectar las diferentes figuras en la imagen adjuntada a la función. Para ello, se ejecuta el código adjuntado posteriormente, pudiendo destacar que se inicia un cronómetro a modo de medir en cuanto tiempo el modelo es capaz de detectar la pareja ganadora en las cartas. Además, el porcentaje de confianza que ha tenido el modelo a detectar (se ha seleccionado 40 % de forma predetermianda pudiendose cambiar según los requerimientos y la aplicación). Por el resto, simplemente se manda al modelo a predecir la apariencia de las figuras en la imagen con la fiabilidad definida anteriormente, además de obtener los bounding boxes, la confianza de cada detección, y la id (número entero) de la clase detectada.

```
def jugador(imagen, clases, modelo):
    inicio = time.time()
    confidence_threshold = 0.4

    result = list(modelo.predict(imagen, conf=confidence_threshold))[0]

    detections = sv.Detections(
        xyxy=result.prediction.bboxes_xyxy,
        confidence=result.prediction.confidence,
        class_id=result.prediction.labels.astype(int)
    )

    identified_classes = detections.class_id
```

Una vez que se obtienen los índices de las clases detectadas, se filtra por apariciones en la lista total de detecciones, siendo tal que si se han repetido más de 1 vez, se considera como posible ganador, y si obtendrá el índice en la lista de detecciones para poder localizar posteriormente su bounding box correspondiente.

```
    conteo = Counter(identified_classes)

    numeros_repetidos = [elemento for elemento, frecuencia in conteo.items() if
        frecuencia > 1]
```

```
    indices_repetidos = [[indice for indice, valor in enumerate(identified_classes) if
        valor == numero] for numero in numeros_repetidos]
    indices_aplanados = [indice for sublist in indices_repetidos for indice in sublist]

    imagen = cv2.imread(imagen)
```

A continuación, se verificaran las clases repetidas y se calculará la media de la confianza sobre la detección de estas. De este modo, se ha pensado que aunque el modelo por algún posible error en la detección tenga varias clases repetidas, solo se muestre la de mayor confianza en la detección, suponiendo que está debe ser siempre la mejor dentro de todas las detecciones.

```
    clases_repetidas = []

    medias_confianzas = {}

    for idx in indices_aplanados:
        class_id = identified_classes[idx]
        conf = detections.confidence[idx]

        if class_id not in medias_confianzas:
            medias_confianzas[class_id] = [conf]
        else:
            medias_confianzas[class_id].append(conf)

    clase_con_mayor_media = max(medias_confianzas, key=lambda x: sum(medias_confianzas[x
        ]) / len(medias_confianzas[x]))
```

Por último, el mostrado de la imagen con las etiquetas, como son el bounding box de la detección, la clase predicha por el modelo y el porcentaje de confianza sobre la detección. Además del guardado de la imagen en el directorio donde se ejecuta la función y el tiempo que ha tardado la función en hacer la detección y el mostrar la imagen con las anotaciones.

```
    fig, ax = plt.subplots(figsize=(10, 10))
    ax.grid(False)
    ax.imshow(cv2.cvtColor(imagen, cv2.COLOR_BGR2RGB))

    for idx in indices_aplanados:
        bbox, conf, class_id = detections.xyxy[idx], detections.confidence[idx],
            identified_classes[idx]

        if class_id == clase_con_mayor_media:
            rect = patches.Rectangle((bbox[0], bbox[1]), bbox[2]-bbox[0], bbox[3]-bbox[1
                ], linewidth=1, edgecolor='r', facecolor='none')
            ax.add_patch(rect)
            ax.text(bbox[0], bbox[1], f'Class: {clases[class_id]}, Conf: {conf:.2f}',
```

```
                color='white', bbox={'facecolor': 'red', 'alpha': 0.5})

    plt.savefig('imagen_con_anotaciones.jpg')

    plt.show()

    final = time.time()

    print("He ganado en: " + str(final-inicio) + "segundos")

    return [clases[class_id] for class_id in numeros_repetidos if class_id ==
        clase_con_mayor_media]
```

Finalmente, se adjunta una prueba de codigo para realizar la llamada de la función, asumiendo que CLASSES y best_model están definidos en algún lugar antes de llamar a la función.

```
image = 'Imag.jpg'
numeros = jugador(image, CLASSES, best_model)
print("Clases que se repiten con mayor media de confianza:", numeros)
```

6. Resultados

Hemos obtenido dos modelos para cada una de las ediciones del juego mencionadas anteriormente y planeamos compararlos utilizando dos métricas clave: mAP (Mean Average Precision) y la pérdida total (Ppyoloeloss/loss). La mAP es una métrica estándar en la evaluación de modelos de detección de objetos en visión por computadora, representando el promedio de las precisiones medias calculadas para cada clase de objeto. Una mAP más alta indica un mejor rendimiento en la detección de objetos. En cuanto a la métrica Ppyoloeloss/loss, se refiere al cálculo de la pérdida total del modelo YOLO-NAS, esta métrica se utiliza para evaluar la eficiencia y la precisión del modelo durante el entrenamiento.

En relación con la edición de los animales, hemos obtenido dos modelos con configuraciones de entrenamiento diferentes. El primer modelo se generó mediante un batch size de 4, 100 épocas y alrededor de 500 imágenes. El mejor modelo de este entrenamiento exhibe un impresionante mAP de 0.9917 y una pérdida total de 1.1552. Por otro lado, el segundo modelo se formó utilizando un batch size de 32, 100 épocas y aproximadamente 1200 imágenes. En este caso, el mejor modelo alcanza un mAP ligeramente superior de 0.9933, aunque con una pérdida total un poco mayor de 1.1703.

Estos resultados tienen sentido, ya que el segundo modelo se entrenó con un batch size más grande, permitiendo procesar un mayor número de imágenes por época de entrenamiento. Además fue entrenado con casi un tercio de menor cantidad de imágenes. Esta capacidad para manejar un mayor volumen de datos por época podría haber contribuido a su mejora en la métrica mAP, a pesar de la pérdida total ligeramente mayor, indicando así una eficiencia y generalización superiores en la detección de animales.

En relación con la segunda edición, la de los personajes Disney, se llevaron a cabo dos entrenamientos con configuraciones distintas. En el primer entrenamiento, se utilizó un batch size de 32, 100 épocas y se trabajó con alrededor de 500 imágenes. El modelo resultante de este proceso exhibió un mAP de 0.98 y una pérdida total de 1.2312. Por otro lado, el segundo entrenamiento, también con un batch size de 32 y 100 épocas, contó con un conjunto de datos más amplio, alrededor de 1500 imágenes. El segundo modelo generado en esta instancia destacó con un mAP superior de 0.99 y una pérdida total más baja de 1.1215.

Estos resultados indican que, en este caso, el aumento en el tamaño del conjunto de datos ha contribuido a una mejora sustancial en la precisión del modelo, como se refleja en el incremento del mAP y la disminución de la pérdida total en comparación con el primer modelo. Aunque cabe destacar que incluso el primero era bastante bueno.

En la Tabla 2, se detallan de manera más clara los resultados obtenidos. En estos resultados, se destaca que los segundos modelos entrenados en cada edición presentan un rendimiento superior. Esto era previsible, dado que estos modelos fueron entrenados con un mayor número de imágenes. En vista de estos resultados, hemos decidido seleccionar estos segundos modelos para llevar a cabo las pruebas en la función *Jugador*.

Edición	Batch Size	mAP	Pérdida Total
Animales - Modelo 1	4	0.9917	1.1552
Animales - Modelo 2	32	0.9933	1.1703
Disney - Modelo 1	32	0.98	1.2312
Disney - Modelo 2	32	0.99	1.1215

Cuadro 2: Resultados de los modelos entrenados para cada edición.

Una vez hayamos cargado los mejores modelos entrenados conforme a las instrucciones previas, procederemos a evaluar su desempeño en la función *Jugador*. Inicialmente, realizaremos la carga de los modelos desde Google Drive, siguiendo el procedimiento detallado anteriormente. Esta acción garantiza que los modelos estén disponibles en el entorno de ejecución de Google Colab. Posteriormente, estaremos listos para cargar tanto archivos con extensión .jpg como .png en la función.

Para evaluar el mejor modelo de la edición de animales, empleamos tres imágenes provenientes de la carpeta de pruebas de la base de datos. Como se puede apreciar en las Figuras 4 y 5, el modelo demuestra una capacidad notable para predecir con precisión las clases, presentando una alta confianza en sus predicciones. En contraste, en la Figura6, se observa que el modelo no es infalible y puede incurrir en confusiones, especialmente cuando las figuras comparten similitudes en color. Además, en este caso particular, donde la predicción es incorrecta, la confianza del modelo es significativamente baja en comparación. Cabe destacar que estas imágenes muestran una calidad no óptima, lo que resalta aún más la robustez y buen rendimiento del modelo al obtener resultados positivos incluso en condiciones menos ideales.

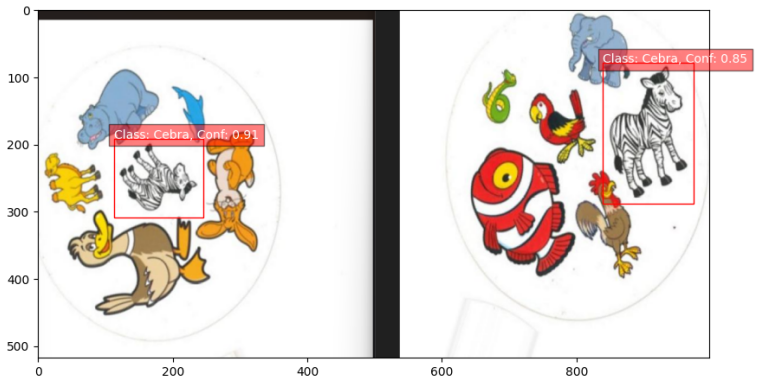

Figura 4: Resultado 1 para la versión Animales

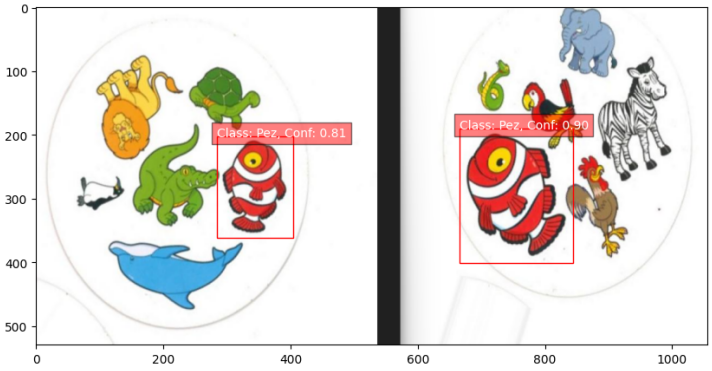

Figura 5: Resultado 2 para la versión Animales

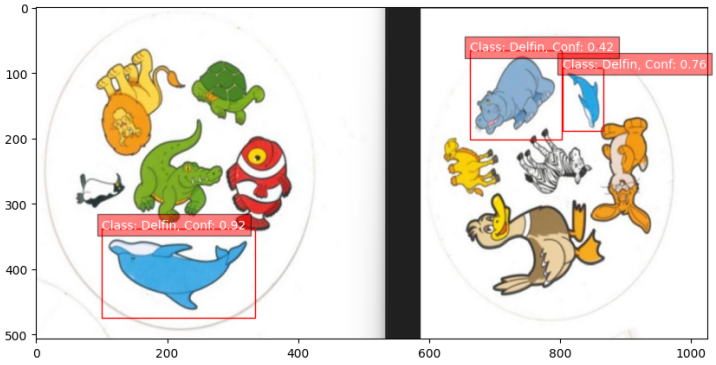

Figura 6: Resultado 3 para la versión Animales

Al cargar el mejor modelo para la versión Disney y realizar pruebas con cuatro imágenes nuevas, que no proceden del dataset, observamos los siguientes resultados. Las dos primeras imágenes se detectan de manera muy precisa, demostrando la eficacia del modelo YOLO-NAS. La tercera imagen también se detecta, aunque con un nivel de confianza ligeramente inferior por parte del modelo. Sin embargo, la última imagen presenta dificultades, ya que el modelo YOLO-NAS confunde dos personajes en lugar de identificarlos correctamente. Estos resultados sugieren que, en general, el modelo funciona de manera notable, logrando excelentes resultados, especialmente en imágenes de alta calidad como las Figuras 7 y 8, que fueron escaneadas. Sin embargo, para las Figuras 9 y 10, capturadas mediante fotografías, se observa que el modelo puede fallar si la calidad de la imagen no es óptima y cuando los personajes son muy parecidos. A pesar de esto, el modelo continúa demostrando un rendimiento notable, resaltando su capacidad para trabajar eficientemente.

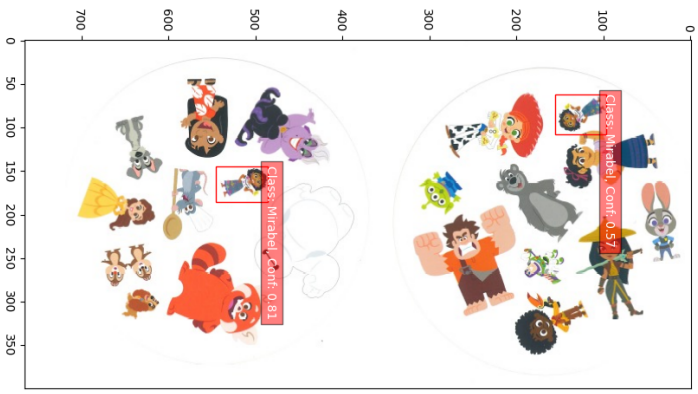

Figura 7: Resultado 1 para la versión Disney

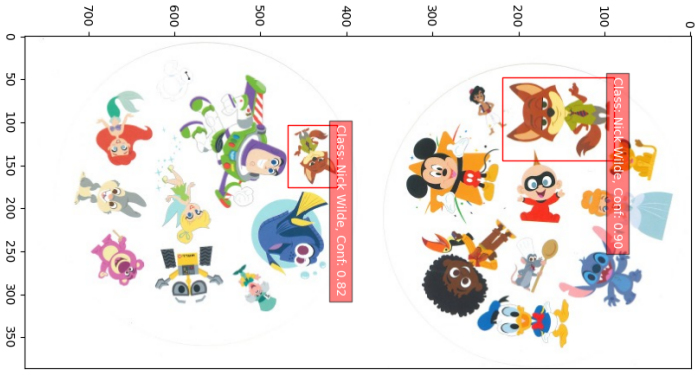

Figura 8: Resultado 2 para la versión Disney

Figura 9: Resultado 3 para la versión Disney

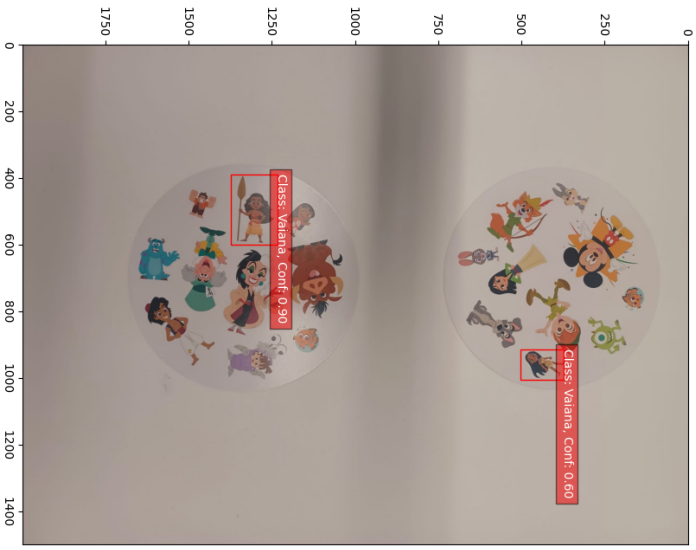

Figura 10: Resultado 4 para la versión Disney

7. Problemas

En esta sección, se abordan los desafíos que surgieron durante la ejecución del proyecto. Inicialmente, se intentó entrenar la red neuronal en nuestros propios ordenadores utilizando Spyder como compilador de Python. No obstante, nos enfrentamos a un problema crucial, ya que las librerías de *super-gradients* no se ejecutan correctamente en sistemas operativos distintos a Linux.

Dado este contratiempo, se exploró la posibilidad de entrenar otros modelos YOLO, como YOLOv8, YOLOv6 y YOLOv5. Desafortunadamente, nos encontramos con los mismos problemas relacionados con las librerías. Ante esta situación, se tomó la decisión de trasladar la ejecución de los códigos a Google Colab, un servicio en la nube gratuito proporcionado por Google. Este entorno, sin necesidad de configuración adicional, ofrecía acceso a GPU, incluyendo las de NVIDIA, requeridas para el proceso de entrenamiento, y permitía compartir fácilmente el código entre colaboradores.

Aunque Google Colab proporcionó el entorno necesario para entrenar la red YOLO-NAS, surgió un nuevo desafío relacionado con la limitación de la memoria RAM. Con el objetivo de realizar un entrenamiento efectivo con un batchsize mínimo de 32, fue necesario disponer de una cantidad significativa de memoria RAM, la cual no estaba disponible en la versión gratuita del servicio. En consecuencia, se optó por contratar el servicio de pago que proporcionaba acceso a casi 90 GB de RAM, permitiendo así ejecutar el entrenamiento con los parámetros deseados. Como consecuencia de este problema, el primer modelo obtenido para la versión de animales, sólo puedo entrenarse con un batchize de 4.

Finalmente, nos enfrentamos a otro inconveniente relacionado con la descarga del modelo entrenado y su posterior incorporación a la función del jugador. La carpeta comprimida (.zip) del modelo tiene un tamaño de aproximadamente 2GB, lo cual impide su descarga directa en Google Colab. Entonces, para poder conseguir descargar dicha carpeta, es necesario cargar el archivo comprimido en Google Drive y, desde allí, realizar la descarga. El proceso de carga del modelo sigue un procedimiento similar en sentido inverso; es decir, se accede a Google Drive para cargar el modelo en Google Colab.

8. Conclusiones

En este proyecto, hemos llevado a cabo el entrenamiento de una red YOLO-NAS con el propósito de clasificar las figuras presentes en las tarjetas del juego Dobble, abarcando dos de sus ediciones: animales y personajes de Disney. Se han obtenido dos modelos distintos para cada edición, seleccionando el mejor de ellos para integrarlo en la función Jugador, la cual identifica el símbolo repetido en dos tarjetas del juego.

A lo largo del desarrollo del proyecto, nos hemos enfrentado a diversos desafíos, como la instalación de las librerías necesarias para el entrenamiento de la red neuronal, así como la gestión de la memoria requerida para ejecutar dicho entrenamiento. A pesar de estos obstáculos, los resultados obtenidos han sido en general muy positivos. Es importante destacar que, aunque los modelos no son perfectos y pueden presentar confusiones en la clasificación de símbolos, especialmente en imágenes de baja calidad, la función *Jugador* demuestra un rendimiento rápido, ejecutándose en aproximadamente 3 segundos para cada modelo.

Es relevante señalar que las pruebas realizadas han mostrado resultados generalmente superiores en la versión de animales. Esto es coherente con la cantidad de símbolos presentes en cada edición; la versión de animales cuenta con 32 símbolos, distribuidos en 6 por tarjeta, mientras que la versión Disney tiene 91 símbolos y 10 por tarjeta. A pesar de estas diferencias, el modelo de la versión Disney sigue proporcionando buenos resultados, lo que nos lleva a considerar este trabajo como exitoso.

9. Anexo

9.1. Data Augmentation

```
import zipfile
import os
import shutil
import cv2
import numpy as np
from keras.preprocessing.image import ImageDataGenerator, load_img, img_to_array
import matplotlib.pyplot as plt
from scipy.spatial import distance
import random


zip_ref = zipfile.ZipFile('Tarjetas.zip', 'r')
zip_ref.extractall('images')
zip_ref.close()

imagenes = os.listdir('images/Tarjetas')

# Calcula el histograma de color de una imagen
def calculate_histogram(image, bins=(8, 8, 8)):
    hist = cv2.calcHist([image], [0, 1, 2], None, bins, [0, 256, 0, 256, 0, 256])
    hist = cv2.normalize(hist, hist).flatten()
    return hist

# Crear el generador de datos de imagen
datagen = ImageDataGenerator(
    rotation_range=360,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    brightness_range=[0.5,1.0])  # Cambia la intensidad de los colores

# Almacena los histogramas de todas las imágenes generadas
histograms = []

# Procesa cada imagen una por una
for imagen in imagenes:
    img = load_img('images/Tarjetas/' + imagen)  # carga la imagen
    data = img_to_array(img)
    data = data.reshape((1,) + data.shape)

    # Generar imágenes aumentadas y guardarlas
    i = 0
```

```
    for batch in datagen.flow(data, batch_size=1, save_to_dir='images/Tarjetas_New',
        save_prefix='Tarjeta', save_format='png'):
        # Calcula el histograma de la imagen generada
        histogram = calculate_histogram(batch[0].astype('uint8'))

        # Compara el histograma con los histogramas de las imágenes ya generadas
        for hist in histograms:
            if cv2.compareHist(hist, histogram, cv2.HISTCMP_BHATTACHARYYA) < 0.1<:  # Si
                la distancia es menor que 0.1, las imágenes son muy similares
                break
        else:  # Si no se encontró ninguna imagen similar, guarda la imagen y su
            histograma
            histograms.append(histogram)
            plt.figure(i)
            imgplot = plt.imshow(batch[0].astype('uint8'))  # Mostrar la imagen
                transformada
            i += 1
            if i % 20 == 0:
                break

    plt.show()

# Verifica si se han generado suficientes imágenes
while len(os.listdir('images/Tarjetas_New')) < 600:
    # Selecciona una imagen aleatoria de la carpeta original
    imagen = random.choice(imagenes)

    img = load_img('images/Tarjetas_Disney/' + imagen)  # carga la imagen
    data = img_to_array(img)
    data = data.reshape((1,) + data.shape)

    # Genera una imagen aumentada y guárdala
    for batch in datagen.flow(data, batch_size=1, save_to_dir='images/Tarjetas_New',
        save_prefix='Tarjeta', save_format='png'):
        # Calcula el histograma de la imagen generada
        histogram = calculate_histogram([0].astype('uint8'))

        # Compara el histograma con los histogramas de las imágenes ya generadas
        for hist in histograms:
            if cv2.compareHist(hist, histogram, cv2.HISTCMP_BHATTACHARYYA) < 0.1: # Si
                la distancia es menor que 0.1, las imágenes son muy similares
                break
        else:  # Si no se encontró ninguna imagen similar, guarda la imagen y su
            histograma
            histograms.append(histogram)
            plt.figure(i)
            imgplot = plt.imshow(batch[0].astype('uint8'))  # Mostrar la imagen
                transformada
            i += 1
```

```
            if i % 2 == 0:
                break

    plt.show()
```

9.2. Yolo-Nas

```
# Comprobamos si hay un directorio ya existente
import os
import shutil

# Especifica el nombre del directorio que quieres vaciar
directorio = '/content/Dobble_Disney_BB_rect-5'

# Verifica si el directorio existe
if os.path.exists(directorio):
    # Elimina todos los archivos y subdirectorios en el directorio especificado
    shutil.rmtree(directorio)
    # Crea de nuevo el directorio vacío
    #os.makedirs(directorio)
else:
    print(f'El directorio {directorio} no existe')

# Verificamos la GPU y RAM disponibles
gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
if gpu_info.find('failed') >= 0:
  print('Not connected to a GPU')
else:
  print(gpu_info)

from psutil import virtual_memory
ram_gb = virtual_memory().total / 1e9
print('Your runtime has {:.1f} gigabytes of available RAM\n'.format(ram_gb))

if ram_gb < 20:
  print('Not using a high-RAM runtime')
else:
  print('You are using a high-RAM runtime!')

# Librerías
!pip install -q git+https://github.com/Deci-AI/super-gradients.git
!pip install -q numpy
!pip install -q supervision
!pip install -q roboflow
!pip install -q supervision
!pip install -q torch
!pip install -q super-gradients==3.5.0.

# IMPORTANTE: REINICIAR SESI N

import super_gradients
print(super_gradients.__version__)

```

```
import os
HOME = os.getcwd()
print(HOME)

%cd {HOME}

# Cambiamos el dataset según los datos con los que se quiera entrenar
from roboflow import Roboflow
rf = Roboflow(api_key='**********************')
project = rf.workspace('dobble-fm3a4').project('dobble_bb_rect')
dataset = project.version(1).download('yolov5')

LOCATION = dataset.location
print("location:", LOCATION)
CLASSES = sorted(project.classes.keys())
print("classes:", CLASSES)

dataset_params = {
    'data_dir': LOCATION,
    'train_images_dir':'train/images',
    'train_labels_dir':'train/labels',
    'val_images_dir':'valid/images',
    'val_labels_dir':'valid/labels',
    'test_images_dir':'test/images',
    'test_labels_dir':'test/labels',
    'classes': CLASSES
}

# Ajuste de la red
MODEL_ARCH = 'yolo_nas_l'
BATCH_SIZE = 32
MAX_EPOCHS = 250
CHECKPOINT_DIR = f'{HOME}/checkpoints'
EXPERIMENT_NAME = project.name.lower().replace(" ", "_")

from super_gradients.training import Trainer

trainer = Trainer(experiment_name=EXPERIMENT_NAME, ckpt_root_dir=CHECKPOINT_DIR)

from super_gradients.training.dataloaders.dataloaders import (
    coco_detection_yolo_format_train, coco_detection_yolo_format_val)

train_data = coco_detection_yolo_format_train(
    dataset_params={
        'data_dir': dataset_params['data_dir'],
        'images_dir': dataset_params['train_images_dir'],
        'labels_dir': dataset_params['train_labels_dir'],
        'classes': dataset_params['classes']
    },
```

```
    dataloader_params={
        'batch_size': BATCH_SIZE,
        'num_workers': 2
    }
)

val_data = coco_detection_yolo_format_val(
    dataset_params={
        'data_dir': dataset_params['data_dir'],
        'images_dir': dataset_params['val_images_dir'],
        'labels_dir': dataset_params['val_labels_dir'],
        'classes': dataset_params['classes']
    },
    dataloader_params={
        'batch_size': BATCH_SIZE,
        'num_workers': 2
    }
)

test_data = coco_detection_yolo_format_val(
    dataset_params={
        'data_dir': dataset_params['data_dir'],
        'images_dir': dataset_params['test_images_dir'],
        'labels_dir': dataset_params['test_labels_dir'],
        'classes': dataset_params['classes']
    },
    dataloader_params={
        'batch_size': BATCH_SIZE,
        'num_workers': 2
    }
)

# Inspeccionamos el dataset
train_data.dataset.transforms

# Instanciamos el modelo
from super_gradients.training import models

model = models.get(
    MODEL_ARCH,
    num_classes=len(dataset_params['classes']),
    pretrained_weights="coco"
)

#Definimos métricas e hiperparámetros
from super_gradients.training.losses import PPYoloELoss
from super_gradients.training.metrics import DetectionMetrics_050
from super_gradients.training.models.detection_models.pp_yolo_e import
    PPYoloEPostPredictionCallback
```

```
145
146
147 train_params = {
148     'silent_mode': False,
149     "average_best_models":True,
150     "warmup_mode": "linear_epoch_step",
151     "warmup_initial_lr": 1e-6,
152     "lr_warmup_epochs": 3,
153     "initial_lr": 5e-4,
154     "lr_mode": "cosine",
155     "cosine_final_lr_ratio": 0.1,
156     "optimizer": "Adam",
157     "optimizer_params": {"weight_decay": 0.0001},
158     "zero_weight_decay_on_bias_and_bn": True,
159     "ema": True,
160     "ema_params": {"decay": 0.9, "decay_type": "threshold"},
161     "max_epochs": MAX_EPOCHS,
162     "mixed_precision": False,
163     "loss": PPYoloELoss(
164         use_static_assigner=False,
165         num_classes=len(dataset_params['classes']),
166         reg_max=16
167     ),
168     "valid_metrics_list": [
169         DetectionMetrics_050(
170             score_thres=0.1,
171             top_k_predictions=300,
172             num_cls=len(dataset_params['classes']),
173             normalize_targets=True,
174             post_prediction_callback=PPYoloEPostPredictionCallback(
175                 score_threshold=0.01,
176                 nms_top_k=1000,
177                 max_predictions=300,
178                 nms_threshold=0.7
179             )
180         )
181     ],
182     "metric_to_watch": 'mAP@0.50'
183 }
184
185 # Entrenamos el modelo
186 trainer.train(
187     model=model,
188     training_params=train_params,
189     train_loader=train_data,
190     valid_loader=val_data
191 )
192
193 # Observamos y guardamos resultados
```

```
%load_ext tensorboard
%tensorboard --logdir {CHECKPOINT_DIR}/{EXPERIMENT_NAME}

import locale
locale.getpreferredencoding = lambda: "UTF-8"

!zip -r yolo_nas.zip {CHECKPOINT_DIR}/{EXPERIMENT_NAME}

# Montar Google Drive
drive.mount('/content/drive')

# Ruta al archivo ZIP en Colab
archivo_zip_colab = '/content/yolo_nas.zip'

# Ruta de destino en Google Drive
ruta_destino_drive = '/content/drive/MyDrive/yolo_nas.zip'

# Copiar el archivo ZIP de Colab a Google Drive
shutil.copy(archivo_zip_colab, ruta_destino_drive)
```

9.3. Carga del modelo

```
# Librerías
!pip install -q git+https://github.com/Deci-AI/super-gradients.git
!pip install -q numpy
!pip install -q supervision
!pip install -q roboflow
!pip install -q supervision
!pip install -q torch
!pip install -q super-gradients==3.5.0.

from google.colab import drive
import zipfile
import os
from roboflow import Roboflow
import torch
import super_gradients
from super_gradients.training import models

# Cargamos el modelo
drive.mount('/content/drive')

file_id = 'yolo_nas.zip'
file_path = '/content/drive/MyDrive/' + file_id

with zipfile.ZipFile(file_path, 'r') as zip_ref:
zip_ref.extractall("/content/")

HOME = os.getcwd()
print(HOME)
%cd {HOME}

# Cargamos el dataset
rf = Roboflow(api_key='********************')
project = rf.workspace('dobble-fm3a4').project('dobble_bb_rect')
dataset = project.version(1).download('yolov5')

LOCATION = dataset.location
print("location:", LOCATION)
CLASSES = sorted(project.classes.keys())
print("classes:", CLASSES)

dataset_params = {
    'data_dir': LOCATION,
    'classes': CLASSES
}

# Definimos el modelo
MODEL_ARCH = 'yolo_nas_l'
```

```
CHECKPOINT_DIR = f'{HOME}/checkpoints'
EXPERIMENT_NAME = project.name.lower().replace(" ", "_")

DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

best_model = models.get(
    MODEL_ARCH,
    num_classes=len(dataset_params['classes']),    checkpoint_path=f"/content/content/
        checkpoints/dobble_bb_rect/RUN_20240121_120128_043707/ckpt_best.pth"
).to(DEVICE)
```

9.4. Función *Jugador*

```
#Librerías
import supervision as sv
import cv2
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from collections import Counter
import time

# Definición de la función
def jugador(imagen, clases, modelo):

# Función Jugador
def jugador(imagen, clases, modelo):
    inicio = time.time()

    # Configuramos umbral de confianza
    confidence_threshold = 0.4

    # Predecimos las clases en la imagen
    result = list(modelo.predict(imagen, conf=confidence_threshold))[0]

    # Creamos objeto de detecciones
    detections = sv.Detections(
        xyxy=result.prediction.bboxes_xyxy,
        confidence=result.prediction.confidence,
        class_id=result.prediction.labels.astype(int)
    )

    # Obtenemos las clases de las detecciones
    identified_classes = detections.class_id

    # Utilizamos Counter para contar las ocurrencias de cada elemento
    conteo = Counter(identified_classes)

    # Obtenemos los elementos que se repiten con frecuencia y sus índices
    numeros_repetidos = [elemento for elemento, frecuencia in conteo.items() if
        frecuencia > 1]
    indices_repetidos = [[indice for indice, valor in enumerate(identified_classes) if
        valor == numero] for numero in numeros_repetidos]
    indices_aplanados = [indice for sublist in indices_repetidos for indice in sublist]

    imagen = cv2.imread(imagen)

    clases_repetidas = []
    medias_confianzas = {}

    # Calculamos la media de las confianzas por clase
```

```
    for idx in indices_aplanados:
        class_id = identified_classes[idx]
        conf = detections.confidence[idx]

        if class_id not in medias_confianzas:
            medias_confianzas[class_id] = [conf]
        else:
            medias_confianzas[class_id].append(conf)

    # Calculamos la media de las confianzas por clase y seleccionamos la mayor
    clase_con_mayor_media = max(medias_confianzas, key=lambda x: sum(medias_confianzas[x
        ]) / len(medias_confianzas[x]))

    # Mostramos la imagen con Bounding Box
    fig, ax = plt.subplots(figsize=(10, 10))
    ax.grid(False) # Ajustar según sea necesario
    ax.imshow(cv2.cvtColor(imagen, cv2.COLOR_BGR2RGB)) # Convertir de BGR a RGB para
        colores correctos

    for idx in indices_aplanados:
        bbox, conf, class_id = detections.xyxy[idx], detections.confidence[idx],
            identified_classes[idx]

        if class_id == clase_con_mayor_media:
            rect = patches.Rectangle((bbox[0], bbox[1]), bbox[2]-bbox[0], bbox[3]-bbox[1
                ], linewidth=1, edgecolor='r', facecolor='none')
            ax.add_patch(rect)
            ax.text(bbox[0], bbox[1], f'Class: {clases[class_id]}, Conf: {conf:.2f}',
                color='white', bbox={'facecolor': 'red', 'alpha': 0.5})

    plt.savefig('imagen_con_anotaciones.jpg')

    plt.show()

    final = time.time()

    print("He ganado en: " + str(final-inicio) + "segundos")

    return [clases[class_id] for class_id in numeros_repetidos if class_id ==
        clase_con_mayor_media]
```