Transcripción automática de símbolos escritos a mano en una imagen.

José Miguel Medianero Cobo

TFM Inteligencia Artificial y Deep Leraning.

Índice

Resumen.

Con el afán por avanzar en la digitalización de la empresa EuroReciclajes, **surge la idea de automatizar ciertas tareas administrativas** que hasta ahora las realizan diferentes personas dentro de dicha empresa. La idea de este PFM es desarrollar una funcionalidad que permita la automatización de una tarea en concreto. Esta tarea es la de registrar en la base de datos el código identificativo del elemento del que se ha procedido a realizar su precintado, junto con la fecha en la que se ha realizado dicho precintado. Hasta ahora, la persona que realiza esta tarea, debe revisar un listado de fotos, al menos una por elemento precintado, donde aparecen los datos anteriormente expuestos Es decir el código identificativo y la fecha de precintado.

Un ejemplo puede ser esta foto.

Fig. 1. Imagen un contenedor precintado

Es un trabajo muy repetitivo y monótono, además que es fácil cometer errores de transcripción, por eso la idea de automatizar dicha tarea. **Esta automatización consistiría en la implementación de una funcionalidad a la que se le pase un lote de imágenes correspondientes al precintado de los elementos en cuestión y sea capaz de obtener el código identificativo del elemento y la fecha en la que se ha realizado su precintado.** Si nos basamos en la imagen anterior, se debería obtener el código identificativo 1641/11-2020/A y la fecha 9-10-23. Estos códigos se le pasarían a otra aplicación, que queda fuera del horizonte de este PFM, que sería la encargada de su inserción en la base de datos.

En el procesamiento de la imagen nos encontraremos con ciertas dificultades, como son el ruido en la imagen, diferentes perspectivas, imágenes borrosas, diferentes resoluciones, objetos no planos, texto rotado o vertical, diferentes grafías, etc. Para llevar a cabo esta funcionalidad y poder realizar los diferentes procesamientos que se deban realizar sobre la imagen para luchar contra las diferentes dificultades

expuestas anteriormente, nos basaremos en el concepto de visión artificial, y la desarrollaremos en el lenguaje de programación Python. También utilizaremos librerías como Numpy, Pandas, OpenCV, Matplotlib además de los frameworks TensorFlow y Keras para tener todas las herramientas necesarias para el desarrollo de la funcionalidad.

Palabras Clave

Visión artificial, red neuronal convolucional, dataset personalizado, modelo pre-entrenado, segmentación semántica, detección de objetos y clasificación de objetos.

Objetivo

El objetivo principal de este PFM, es el **desarrollo de uno o más sistemas basados en IA, en concreto en visión artificial, para poder automatizar la tarea de inserción en la base de datos** de los datos del precintado de cada contenedor instalado en las explotaciones ganaderas que trabajan con el sistema de hidrolizadores.

Por otra parte los objetivos secundarios que se persiguen con la implementación de este PFM son:

- Demostrar los conocimientos y competencias adquiridas en machine learning, deep learning, programación en Python y la utilización de librerías como Numpy, Pandas, OpenCV o frameworks como TensorFlow y Keras, además de los diferentes entornos de programación como Jupyter Notebooks y Google Colab entre otros.

- Adquirir o profundizar en conocimientos y competencias que en la parte teórica de este máster no han sido tratadas o no se han tratado de manera lo suficientemente extensa y son totalmente necesarias para la elaboración de este PFM.

Metodología.

Para poder realizar la transcripción de los datos escritos a mano que aparecen en cada imagen y corresponden con una codificación y una fecha seguiremos los siguientes pasos.

- Realizaremos una **segmentación semántica de la zona donde aparecen los datos escritos a mano que queremos transcribir.** De esta manera delimitamos la zona a tratar evitando así otras zonas en la imagen que contengan otros símbolos que pueden interferir con la transcripción. Para realizar dicha segmentación implementaremos un modelo deep learning con una red neuronal convolucional basada en la arquitectura U-NET. Para su construcción usaremos TensorFlow. Este modelo lo entrenaremos con un dataset personalizado que lo construiremos a partir de una selección de imágenes etiquetadas con la herramienta *LabelImg.*

- Sobre la zona segmentada realizaremos una **detección de objetos**. Denominamos objetos a cada uno de los símbolos que componen el texto escrito a mano en el interior de la zona segmentada. Para realizar dicha detección de objetos **utilizaremos un modelo pre-entrenado** llamado YOLO V8. Es un modelo de deep learning que está compuesto por una red convolucional y realiza la detección de objetos en tan solo una pasada sobre la imagen en la que trabaja. El modelo lo entrenaremos con un dataset personalizado que lo construiremos a partir de una selección de imágenes etiquetadas con la herramienta *Roboflow.*

- Una vez detectados los objetos que componen los datos de identificación del contenedor y la fecha, los **clasificaremos** para determinar la clase a la que pertenece cada objeto y poder construir así el código del contenedor y su fecha de precintado para su posterior almacenamiento en la base de datos.

1.- Segmentación semántica

1.1.- Definición de segmentación semántica

La **segmentación semántica** en deep learning consiste en **clasificar cada píxel de una imagen asociándolo a una única clase o categoría.** Las clases o categorías vienen definidas según las necesidades que tengamos a la hora de realizar la segmentación y dependen del problema a tratar. A cada clase se le asocia una etiqueta con nombre único. Para realizar la segmentación debemos tener claro a priori las clases o categorías en las que queremos clasificar los píxeles de la imagen.

En nuestro caso utilizaremos la segmentación semántica para la clasificación de los píxeles de la imagen en dos categorías distintas:

> ***clase precinto***: que son los píxeles que forman la zona de la etiqueta del precinto, la zona que queremos segmentar.
>
> ***clase fondo***: que es el resto de píxeles que no pertenecen a la clase precinto, la zona que queremos eliminar.

De esta forma **conseguimos aislar la zona de la etiqueta del precinto del resto de la imagen** para evitar posibles interferencias o ruidos a la hora de reconocer la información escrita en dicha zona de la imagen.

1.2.- Preparación de los datos.

El primer paso que debemos dar es el de preparar los datos que con posterioridad utilizaremos para entrenar, evaluar y probar la precisión de nuestro modelo de red neuronal . Para ello utilizaremos una batería de imágenes en las cuales mediante la herramienta LabelImg etiquetaremos las zonas de segmentación.

1.2.1.- Etiquetado de imágenes.

Labelme es una herramienta de anotación o etiquetado de imágenes para datasets utilizados en visión artificial. Es la herramienta que vamos a utilizar para **crear nuestro dataset personalizado.**

Para instalarla necesitaremos realizar los siguientes pasos.

1.- Abrir la consola de Conda

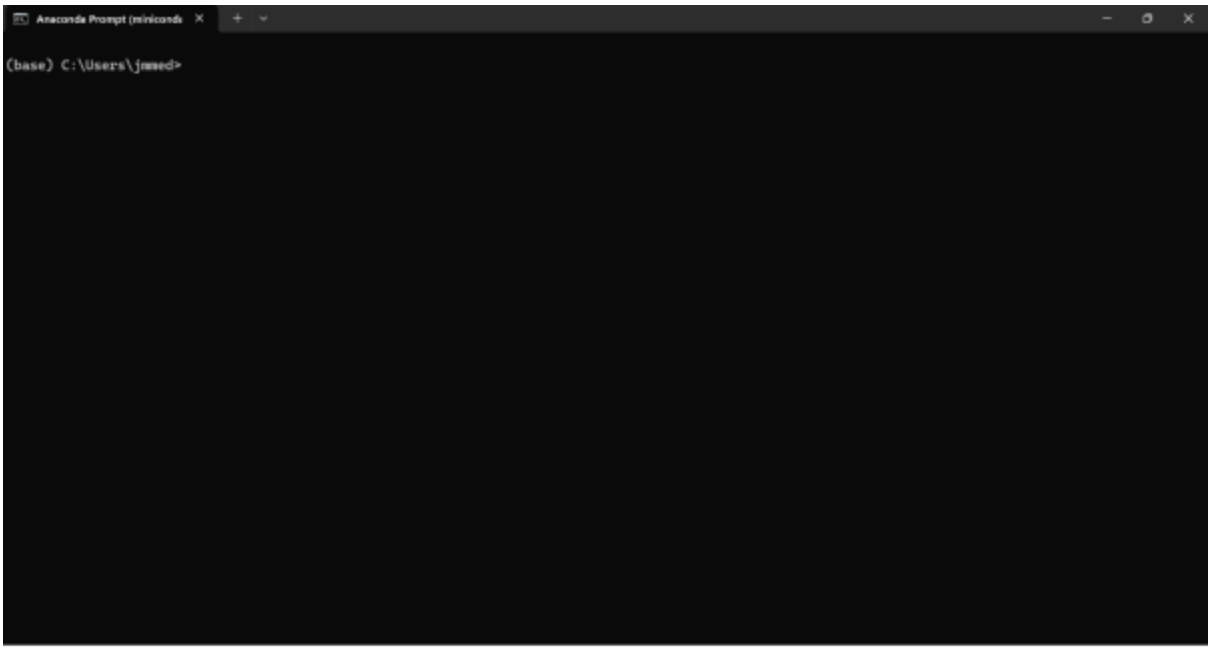


Fig. 1.1. Consola Conda

2.- Crear un entorno de desarrollo llamado Labelme

Fig. 1.2. Consola Conda. Creando Labelme

3.- Activar el entorno de desarrollo Labelme

Fig 1.3. Activando Labelme

4.- Instalar el paquete Labelme con el gestor de paquetes de Python pip

Fig 1.4. Instalando paquetes de Labelme

5.- Por último ejecutar el comando labelme desde la línea de comandos de Conda dentro del entorno Labelme.

Fig 1.5. Abriendo Labelme

Tras realizar todos estos pasos, se abrirá una ventana donde se ejecutará la herramienta de etiquetado Labelme.

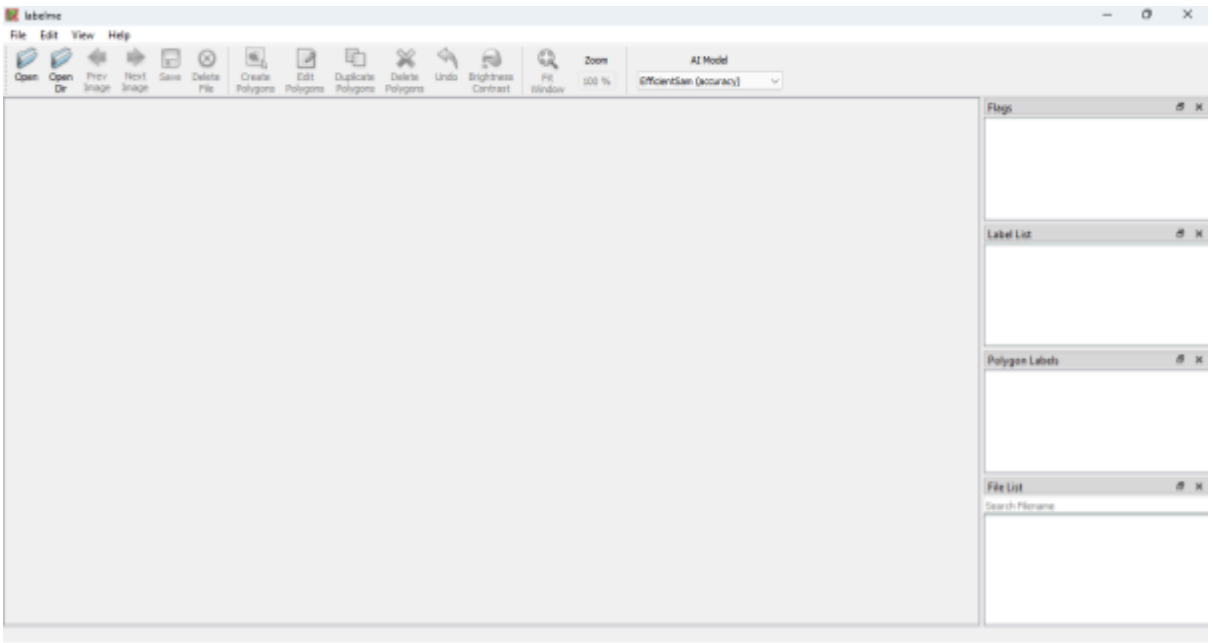


Fig. 1.6. Escritorio de Labelme

Con esta herramienta seleccionaremos una imagen de las que compondrá nuestro dataset personalizado. Mediante la opción Create Polygons seleccionaremos la región de la imagen cuyos píxeles corresponderán con la clase precinto, como hemos explicado con anterioridad.

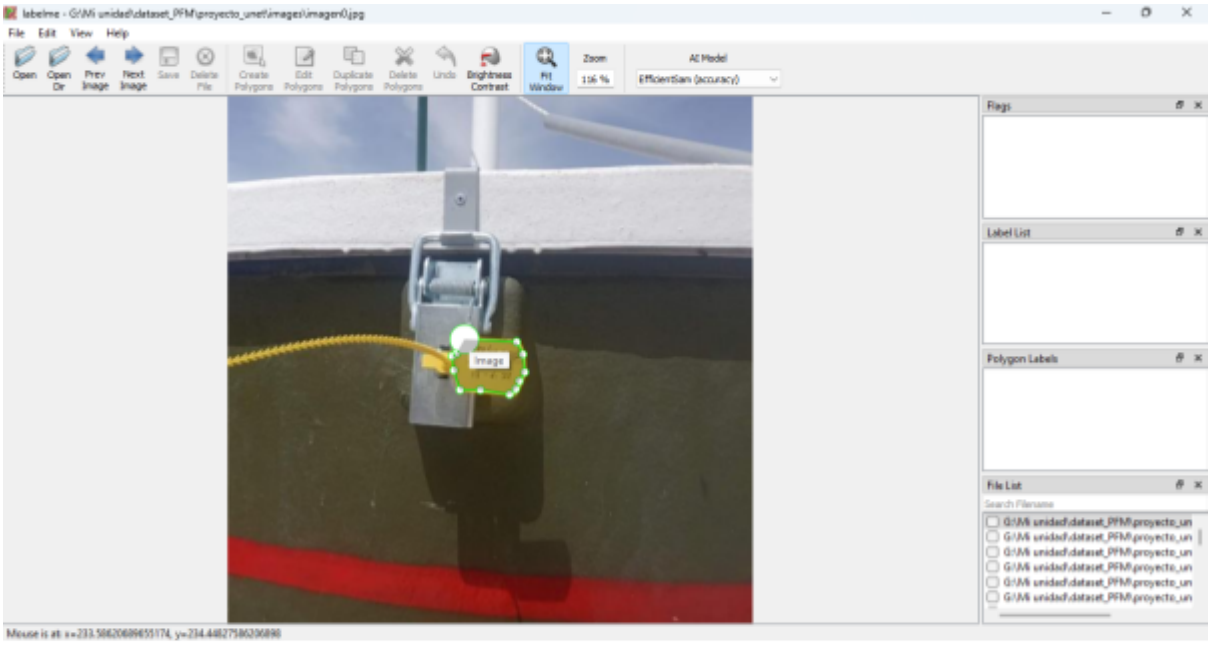


Fig. 1.7. Etiquetando segmentación

Cuando cerremos los puntos del polígono seleccionado se abrirá una ventana para etiquetar esa región con uno de los nombre de las clases a las que puede pertenecer cada píxel de la imagen. En nuestro caso precinto.

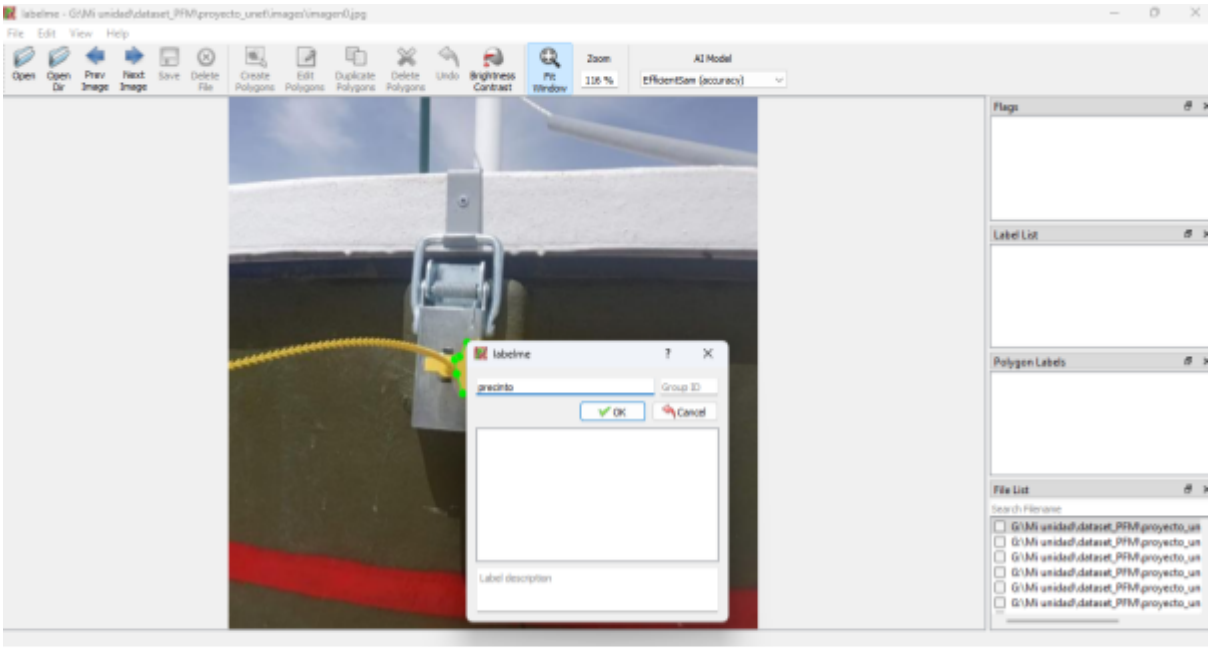


Fig. 1.8. Guardando etiquetado

Una vez etiquetada la región pulsamos en la opción Save para guardar todo.

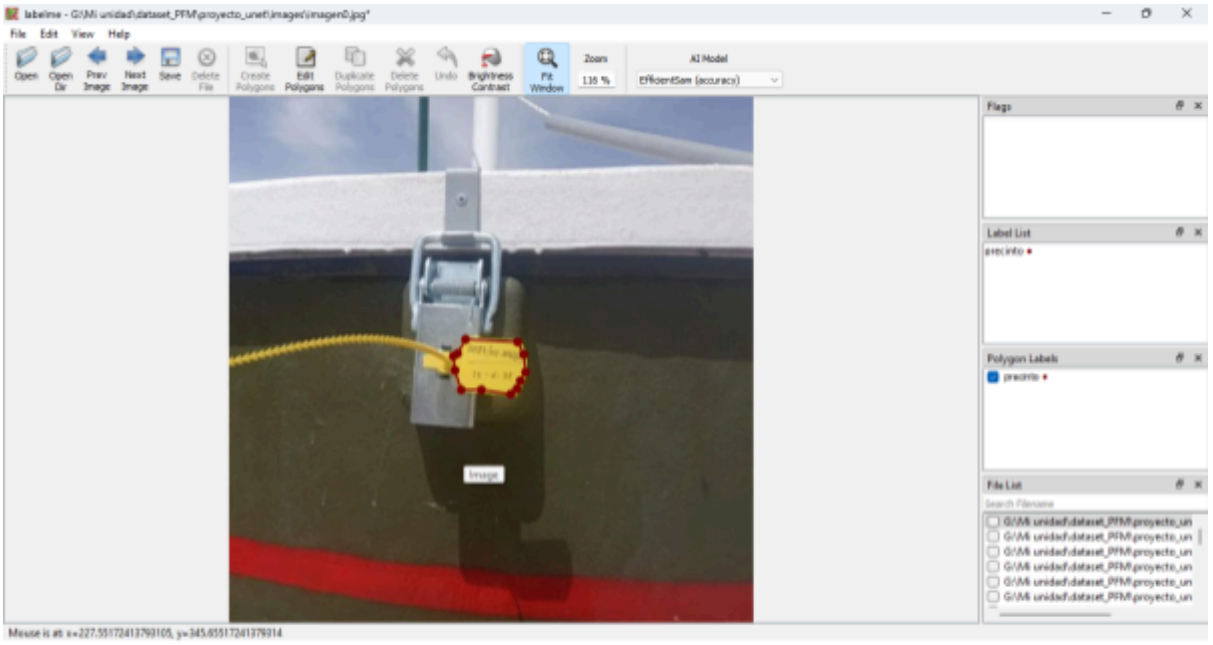


Fig. 1.9. Muestra de etiquetado.

En este momento se generará un archivo .JSON en formato COCO con el etiquetado de la imagen.

Nombre	Fecha	Tipo	Tamaño	Etiquetas
imagen0	01/08/2024 10:27	Archivo JPG	66 KB	
imagen0	09/08/2024 17:48	Archivo de origen JSON	32 KB	

Fig. 1.10. Archivo JSON con el etiquetado de las imágenes

1.2.2.- Formato de los datos para la segmentación.

COCO (Common Objects in Context) es un **formato de etiquetado de imágenes para clasificación, detección y segmentación, que utiliza un formato JSON para su codificación.** El formato COCO es comúnmente utilizado en deep learning.

En nuestro caso, el archivo .JSON en el que se especifica el formato COCO correspondiente al etiquetado de la segmentación, tiene el siguiente aspecto:

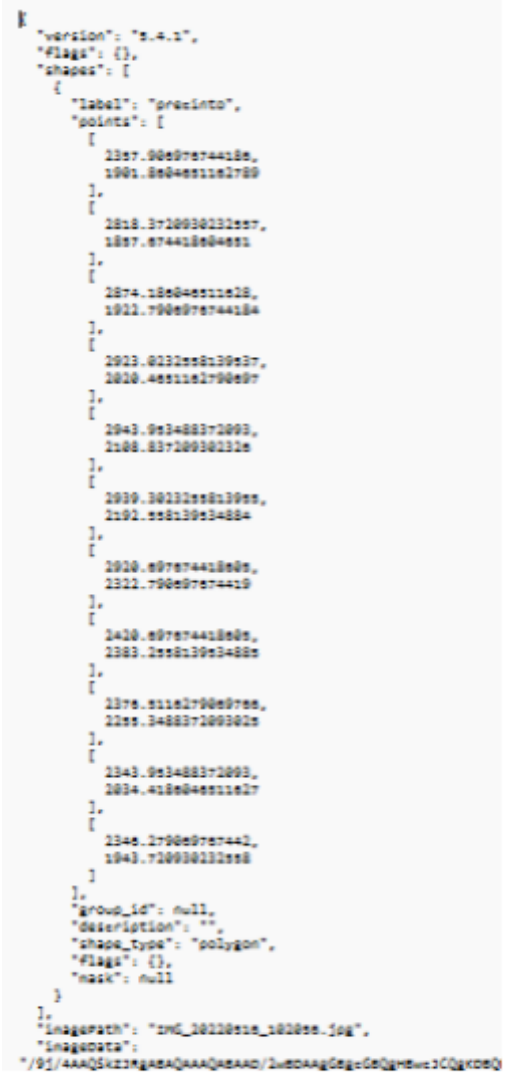


Fig. 1.11. Ejemplo del contenido de un archivo con formato COCO

Como se puede observar, en el archivo se muestran diferentes campos como la versión, los flags. etc. A nosotros solamente nos interesa el campo shapes que es donde se especifica el nombre de la etiqueta que vamos segmentar y la lista con los puntos que delimitan la zona a segmentar.

1.2.3.- Generación de máscaras de segmentación.

Actualmente tenemos una serie de imágenes y una serie de ficheros .json en formato COCO que ha proporcionado LabelImg, donde se especifican los parámetros de la segmentación en cada una de las diferentes imágenes.

Para poder generar las máscaras que usaremos para entrenar el modelo que tenemos que construir para realizar la segmentación, debemos hacer una **preparación de los datos**.

Dicha preparación consiste en **transformar los parámetros incluidos en el archivo .json en una imagen que contenga la máscara de segmentación**.

Lo primero que haremos será leer el nombre de archivo de todas las imágenes que forman nuestra selección y construiremos un dataframe de Pandas donde asociaremos por filas el nombre de archivo de la imagen con el nombre de archivo .json asociado a la imagen.

Añadimos la ruta de entrada de los datos originales y las rutas de salida para las imagenes y las máscaras.

```
#Inicializacion de datos
# Establecemos las rutas de entrada y salida


path_in = '/content/gdrive/MyDrive/dataset_PFM/proyecto_unet/dataset_original_precintos/'
path_out_images= '/content/gdrive/MyDrive/dataset_PFM/proyecto_unet/images/'
path_out_masks = '/content/gdrive/MyDrive/dataset_PFM/proyecto_unet/masks/'
```

Haz doble clic (o pulsa Intro) para editar

```
[ ] # Cargamos las imagenes de la ruta especifficada
    imagenes = CargarImagenes(path_in)

    # Creamos un DF donde relacionamos cada imagen con su archivo json

    df_imagenes = pd.DataFrame()

    for i in range(len(imagenes)):
        df_imagenes.loc[i,'imgs']=imagenes[i]
        df_imagenes.loc[i,'msks'] = imagenes[i].replace('.jpg', '.json')

    df_imagenes.head()
```

	imgs	msks
0	IMG_20220613_121059.jpg	IMG_20220613_121059.json
1	IMG_20220516_102249.jpg	IMG_20220516_102249.json
2	IMG_20220620_110423.jpg	IMG_20220620_110423.json
3	IMG_20220606_103514.jpg	IMG_20220606_103514.json
4	IMG_20220620_104318.jpg	IMG_20220620_104318.json

Fig. xx. Detalle del dataframe con las imágenes y su máscaras.

Una vez que hemos determinado los nombres de las imágenes, de los ficheros .json y de la relación que hay entre ellos, pasaremos a **construir una carpeta con las**

imágenes y otra carpeta con las máscaras resultantes a partir de los .json. También **construiremos un archivo .csv con la relación entre las imágenes y las máscaras** para poder realizar su tratamiento con mayor sencillez y sin necesidad de cargar todas las imágenes a la vez en memoria con el excesivo consumo de recursos que conlleva.

En el proceso de generación de máscaras, aprovecharemos y **recortaremos las imágenes y las máscaras a un aspecto cuadrado y las redimensionaremos a un formato de 512x512 píxeles.**

El código a ejecutar es el siguiente:

```
df_archivos_mascara = pd.DataFrame(columns=['imgs', 'msks'])

for i in range(len(df_imagenes)):

    # Cargamos la imagen original para obtener su forma
    #imagen_entrada = cargarImagen(path_in + df_imagenes.loc[i,0])
    imagen_entrada = cv2.imread(path_in + df_imagenes.loc[i,'imgs'])

    alto, ancho = imagen_entrada.shape[:2]

    # Para generar su mascara, creamos una matriz de ceros
    #con la forma de la imagen original
    mascara= np.zeros(shape=(alto,ancho,1), dtype=np.uint8)

    # Cargamos los puntos del área seleccionada para generar la mascara
    puntos = np.array(
        redondearPuntos(obtenerPuntos(
            path_in + df_imagenes.loc[i,'msks'])))

    # Generamos el poligono cerrado que define la lista de puntos
    # obtenida del json y lo rellenamos con el valor 1 en cada capa
    # que corresponde con el código de la clase 'precinto'
    cv2.fillPoly(mascara, pts=[puntos], color=(1,1,1))

    # Creamos un nuevo DF donde relaccionamos cada imagen original
    # con su mascara
    df_archivos_mascara.loc[i,'imgs'] = f'imagen{i}.jpg'
    df_archivos_mascara.loc[i,'msks'] = f'mascara{i}.jpg'
```

```
# Recortamos la imagen original al rededor de la seccion
# de la mascara para obrtener imagenes cuadradas que la contengan
coordenadasX = min_max_X(puntos)
coordenadasY = min_max_Y(puntos)
ancho_mascara = coordenadasX[1]-coordenadasX[0]
alto_mascara = coordenadasY[1]-coordenadasY[0]
```

```
if ancho > alto:
    # Recortamos horizontalmente conservando todo el alto de
    # la imagen original
    margen= (alto-ancho_mascara) // 2
    inicio = 0
    fin = 0
    margen_dcho = ancho-(coordenadasX[1]+margen)
    margen_izdo = coordenadasX[0]-margen
    if (margen_dcho>=0) and (margen_izdo>=0): #Imagen centrada
        inicio = coordenadasX[0]-margen
        fin = inicio + alto

    elif margen_dcho<0: #Imagen desplazada a la derecha
        inicio= ancho -alto
        fin = ancho

    elif margen_izdo<0: #imagen desplazada a la izquierda
        inicio = 0
        fin = alto
    imagen_recortada = imagen_entrada[:,inicio:fin]
    mascara_recortada = mascara[:,inicio:fin]
```

```
else:
    #recortamos verticalmente conservando todo el ancho de la imagen
    # original
    margen= (ancho-alto_mascara) // 2
    inicio = 0
    fin = 0
    margen_inf = alto-(coordenadasY[1]+margen)
    margen_sup = coordenadasY[0]-margen

    if (margen_inf>=0) and (margen_sup>=0):
        inicio = coordenadasY[0]-margen
        fin = inicio + ancho

    elif margen_inf<0:
        inicio = alto - ancho
        fin = alto

    elif margen_sup<0:
        inicio = 0
        fin = ancho

    imagen_recortada = imagen_entrada[inicio:fin, :]
    mascara_recortada = mascara[inicio:fin,:]
```

```
    #Redimensionamos a una imagen 512x512 para su posterior tratamiento en u-net

    mascara_recortada_original = mascara_recortada

    imagen_recortada = redimensionar(imagen_recortada, 512,512)
    mascara_recortada = redimensionar(mascara_recortada, 512, 512)

    cv2.imwrite(path_out_images + f'imagen{i}.jpg', imagen_recortada)
    cv2.imwrite( path_out_masks + f'mascara{i}.jpg', mascara_recortada)
# Generamos un archivo csv que contiene las relacciones entre las imagenes y sus
# correspondientes mascaras

df_archivos_mascara.to_csv(path_out_masks + 'rel_archivos.csv')
```

Una vez ejecutado el código de la imagen anterior, se habrán generado en la ruta *'...../images/'* una serie de archivos en formato .jpg y en la ruta *'...../masks/'* una serie de archivos de máscara también en formato .jpg. En esta carpeta también se ha guardado el archivo *'rel_archivos.csv'* que contiene la relación del nombre de imagen con el nombre de su máscara asociada.

Tras la realización de este paso, **ya tendríamos preparados los datos para su utilización** en la red neuronal.

1.3.- Segmentación mediante redes neuronales.

1.3.1.- Redes neuronales convolucionales

Un enfoque habitual para **abordar la segmentación semántica consiste en crear un modelo basado en una red con arquitectura de red neuronal convolucional** (CNN).

Una red neuronal convolucional (CNN), es un **tipo especializado de algoritmo de deep learning diseñado principalmente para tareas que requieren reconocimiento de objetos, como la clasificación, la detección y la segmentación de imágenes.**

El concepto de red neuronal convolucional está basado en el funcionamiento de las neuronas que forman la corteza visual de un cerebro humano. Dicha corteza visual está compuesta por dos tipos de células neuronales, las simples y las complejas.

Cada neurona simple ve una pequeña porción del campo visual y se encarga de detectar líneas o ejes orientados de cierta forma o posición. Para cubrir el campo visual completo tenemos muchas neuronas localizadas en esta zona de la corteza cerebral.

Por otro lado, las neuronas complejas se encargan de agrupar lo que han detectado un grupo de neuronas simples. También detectan líneas o ejes pero a las neuronas complejas no les importa ni su orientación ni posición, pudiendo así detectar características más complejas que las neuronas simples.

En la corteza visual se asocian muchas capas de neuronas simples y complejas para poder así detectar conceptos cada vez más complejos y al final unirlos y construir la imagen que vemos.

En las redes neuronales convolucionales, a semejanza del modelo biológico, tendremos también dos tipos de neuronas organizadas en capas de convolución y capas de agrupación.

Las capas de convolución consiguen detectar características básicas o patrones dentro de la imágen, mediante aplicación de operaciones matriciales sobre los píxeles que forman dicha imagen.

Por otro lado, **las capas de agrupación tienen dos cometidos, reducir el tamaño de la imagen y resaltar las características más importantes para detectar así patrones complejos**. Actualmente se utiliza la capa de agrupación máxima.

En la siguiente imagen se puede ver en forma de esquema la similitud del modelo de red convolucional y el modelo biológico comentado anteriormente.

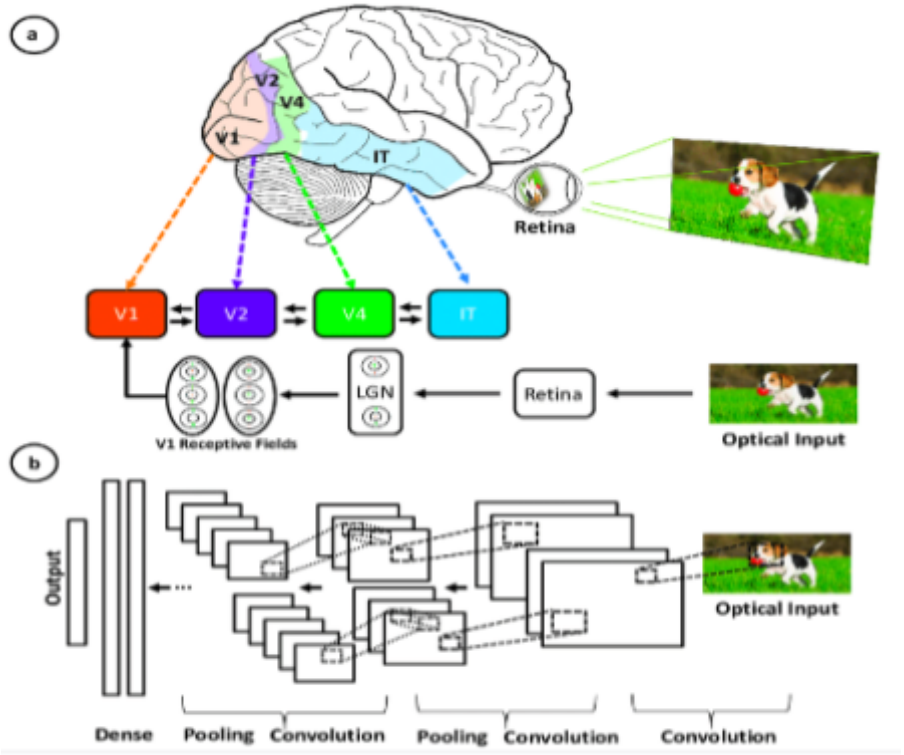


Fig.1.12. Esquema del funcionamiento de una red convolucional.

1.3.2- Arquitectura U-NET.

Una vez definido lo que es una red neuronal convolucional pasaremos a explicar la arquitectura de la red neuronal U-NET, que es la que se ha elegido para construir el segmentador semántico del proyecto.

U-NET es un modelo de red neuronal, basado en redes convolucionales, dedicado a problemas de segmentación semántica. Este modelo fue desarrollado originalmente por Olaf Ronneberger, Phillip Fischer y Thomas Brox en 2015 para la segmentación de imágenes médicas.

La arquitectura de U-NET consta de **dos fases**.

La primera fase es la de la **contracción o codificador** y se usa para captar el contexto de la imagen.

Esta fase consiste de **una sucesión de capas de convolución y de capas de agrupación tipo asociación máxima o max pooling, que permiten crear un mapa de características de una imagen y reducir su tamaño para así disminuir el número de parámetros de la red.**

La segunda fase es la de **expansión, también llamada decodificador.** Esta fase permite una **localización precisa mediante la convolución transpuesta**. Es necesario que en cada etapa de expansión se tenga la salida de la etapa correspondiente de la fase de contracción para que la red pueda ir reconstruyendo las imágenes a sus tamaños correspondientes en cada etapa y así no perder información. Al finalizar la etapa de expansión, el tamaño de salida que nos proporciona U-NET es igual al de entrada.

Este tipo de red neuronal **permite obtener una mayor precisión que modelos convencionales aun cuando tengamos un volumen de datos no muy grande para su entrenamiento.** Estas dos características hacen que las redes U-NET se utilicen de forma habitual en procesos de segmentación semántica.

En el siguiente esquema podemos ver claramente cómo se dividen las fases de contracción y expansión y las etapas que las forman, al igual que como se utiliza la salida de cada etapa de contracción en la fase de expansión para reconstruir el tamaño original de la imagen en cada etapa y no perder información en el proceso.

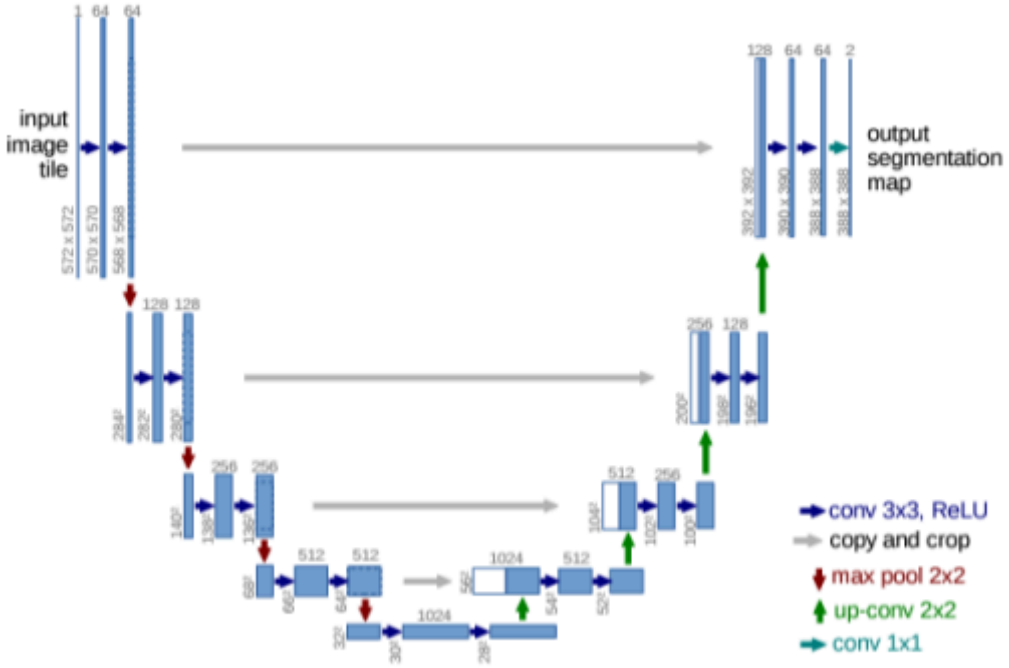


Fig. 1.13 Esquema de la arquitectura U-Net

1.3.3.- Modelo de segmentación.

Para poder **entrenar, validar y probar el modelo necesitamos preparar los datos que forman nuestro dataset** personalizado que hemos construido con anterioridad**.**

Lo primero que debemos hacer es abrir el fichero con los metadatos que relacionan las imágenes y sus máscaras.

```
path_metadata = "/content/gdrive/MyDrive/dataset_PFM/proyecto_unet/masks/rel_archivos.csv"

df = pd.read_csv(path_metadata)
```

Una vez leídos los metadatos, los mezclaremos para evitar que los resultados del entrenamiento de la red neuronal estén condicionados al orden de los datos que proporcionamos. Además calculamos el tamaño de los datasets de entrenamiento, validación y prueba.

```
# Mezclamos las filas del dataframe de manera aleatoria
df = df.sample(frac=1, random_state=23)

# Calculamos el numero total de imagenes y mascaras, asi como el numero de
# elmentos de los datasets de entrenamiento, validacion y prueba

N = df.shape[0]

n_train = int(0.8*N)
n_val = int(0.1*N)
n_test = N - n_train - n_val

print('Num total: ', N)
print('Num train: ', n_train)
print('Num_val: ', n_val)
print('Num test: ', n_test)
```

```
Num total:  204
Num train:  163
Num_val:  20
Num test:  21
```

Sabiendo el tamaño de los datasets de entrenamiento, validación y prueba, pasaremos a construir los archivos que contendrán el nombre de las imágenes y el nombre de las máscaras

```
# Seleccinamos del dataframe los nombres de ficheros de imagenes y mascaras
# correspondientes a los datasets de entrenamiento, validacion y pruebas
imgs_train_fnames = df['imgs'].values[0:n_train]
msks_train_fnames = df['msks'].values[0:n_train]

imgs_val_fnames = df['imgs'].values[n_train:n_train+n_val]
msks_val_fnames = df['msks'].values[n_train:n_train+n_val]

imgs_tst_fnames = df['imgs'].values[n_train+n_val:]
msks_tst_fnames = df['msks'].values[n_train+n_val:]
```

A partir de estas listas con los nombres de archivos de imágenes y máscaras correspondientes a entrenamiento, validación y prueba, Generaremos los datasets necesarios.

```
# Construimos los datasets a partir de los archivos de nombres creados con anterioridad
# a partir del dataframe original
dataset_train = tf.data.Dataset.from_tensor_slices((imgs_train_fnames, msks_train_fnames))
dataset_val = tf.data.Dataset.from_tensor_slices((imgs_val_fnames, msks_val_fnames))
dataset_test = tf.data.Dataset.from_tensor_slices((imgs_tst_fnames,msks_tst_fnames))
```

Construiremos una función auxiliar para leer cada imagen y cada máscara de la ruta donde se encuentran, almacenándola así en memoria solamente en el momento en el que se vaya a utilizar. Evitando de esta forma un excesivo uso de recursos del sistema.

En esta función se realiza la **normalización de la imagen a valores entre 0 y 1 para su correcto procesamiento por parte de la red neuronal**. Para ello es necesario convertir los valores de los píxeles de cada canal a float32.

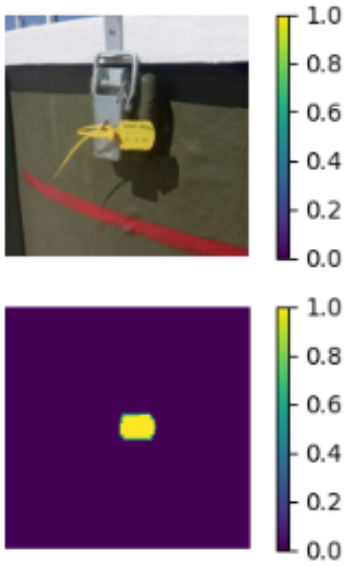
En la **máscara no es necesario hacer esta conversión, porque se ha construido inicialmente con los valores correspondientes a cada clase**. En nuestro caso 0 para el fondo y 1 para la clase precinto. Por tanto nos sirven los valores tipo unit8 originales y no es necesario su conversión a float32.

```
#Ruta donde almacenaremos las imagenes
path_img= '/content/gdrive/MyDrive/dataset_PFM/proyecto_unet/images/'
# Ruta donde almecenaremos las máscaras que construiremos con posterioridad
path_msk = '/content/gdrive/MyDrive/dataset_PFM/proyecto_unet/masks/'
# Funcion preprocesar_imagen: a partir del nombre del archivos de imagen y
# mascara, retorna la imagen y la mascara en el formato preparado para su
# utilizacion por la red convolucional

def preprocesar_imagen(name_img, name_msk):
    #print('Dir imagen: ', path_img)
    #print('Dir mask: ', path_msk)

    #print (' imagen: ', name_img)
    #print('Mascara: ', name_msk)
    #print('Ruta imagen: ',path_img + name_img )
    #print('Ruta mascaa: ',path_msk + name_msk )


    imagen = tf.io.read_file(path_img + name_img)
    mask = tf.io.read_file(path_msk + name_msk)

    # Decodificamos la imagen en 3 canales
    imagen = tf.image.decode_jpeg(imagen, channels=3) #unit8

    # Decodificamos la mascara en 1 solo canal
    mask = tf.image.decode_jpeg(mask,channels=1)
    #Comnvertimos la imagen a float32
    imagen = tf.image.convert_image_dtype(imagen, dtype = tf.float32)

    # Convertimos la mascara a float32
    #mask = tf.image.convert_image_dtype(mask, dtype= tf.float32)

    return imagen, mask
```

Mostraremos a modo de ejemplo una imagen y su máscara preparadas para su utilización por parte de la red neuronal.

```
# preparada para su utilizacion por la red neuronal

import matplotlib.pyplot as plt

plt.figure(figsize=(4,4))

plt.subplot(2,1,1)
plt.imshow(img)
plt.colorbar()
plt.axis('off')

plt.subplot(2,1,2)
plt.imshow(msk)
plt.colorbar()
plt.axis('off')
```

```
(-0.5, 511.5, 511.5, -0.5)
```



En este punto estamos en disposición de **empezar a construir nuestro modelo** a partir de la **arquitectura U-Ne**t.

Y este sería el código que implementa una red neuronal con arquitectura U-Net. Comenzamos con las importaciones de las librerías.

```
from tensorflow import keras
from keras import layers, models
```

```
# Fijamos la semilla del generador aleatorio
tf.random.set_seed(123)

img_size = (512,512,3)
num_clases = 2
```

La parte de la asignación de la entrada y del codificador sería esta:

```
entrada = tf.keras.Input(shape=img_size)

#Entrada

#Codificador
# Capa conv1

conv1 = layers.Conv2D(16, (3,3), activation='relu', padding='same')(entrada)
conv1 = layers.Conv2D(16, (3,3), activation='relu', padding='same')(conv1)
pool1 = layers.MaxPool2D((2,2))(conv1)

#Capa conv2

conv2 = layers.Conv2D(32, (3,3), activation='relu', padding='same')(pool1)
conv2 = layers.Conv2D(32, (3,3), activation='relu', padding='same')(conv2)
pool2 = layers.MaxPool2D((2,2))(conv2)

#Capa conv3

conv3 = layers.Conv2D(64, (3,3), activation='relu', padding='same')(pool2)
conv3 = layers.Conv2D(64, (3,3), activation='relu', padding='same')(conv3)
pool3 = layers.MaxPool2D((2,2))(conv3)

#Capa conv4

conv4 = layers.Conv2D(128, (3,3), activation='relu', padding='same')(pool3)
conv4 = layers.Conv2D(128, (3,3), activation='relu', padding='same')(conv4)
pool4 = layers.MaxPool2D((2,2))(conv4)

#Capa conv5

conv5 = layers.Conv2D(256, (3,3), activation='relu', padding='same')(pool4)
conv5 = layers.Conv2D(256, (3,3), activation='relu', padding='same')(conv5)
```

Y la parte del decodificador y la salida sería esta:

```
#Decodificador

#Capa Dec1

dec1 = layers.Conv2DTranspose(128, (2,2), strides=(2,2), padding='same')(conv5)
dec1 = layers.Concatenate()([dec1,conv4])
dec1 = layers.Conv2DTranspose(128, (3,3), activation = 'relu', padding='same')(dec1)
dec1 = layers.Conv2DTranspose(128, (3,3), activation = 'relu', padding='same')(dec1)

#Capa Dec2

dec2 = layers.Conv2DTranspose(64, (2,2), strides=(2,2), padding='same')(dec1)
dec2 = layers.Concatenate()([dec2,conv3])
dec2 = layers.Conv2DTranspose(64, (3,3), activation = 'relu', padding='same')(dec2)
dec2 = layers.Conv2DTranspose(64, (3,3), activation = 'relu', padding='same')(dec2)


#Capa Dec3

dec3 = layers.Conv2DTranspose(32, (2,2), strides=(2,2), padding='same')(dec2)
dec3 = layers.Concatenate()([dec3,conv2])
dec3 = layers.Conv2DTranspose(32, (3,3), activation = 'relu', padding='same')(dec3)
dec3 = layers.Conv2DTranspose(32, (3,3), activation = 'relu', padding='same')(dec3)


#Capa Dec4

dec4 = layers.Conv2DTranspose(16, (2,2), strides=(2,2), padding='same')(dec3)
dec4 = layers.Concatenate()([dec4,conv1])
dec4 = layers.Conv2DTranspose(16, (3,3), activation = 'relu', padding='same')(dec4)
dec4 = layers.Conv2DTranspose(16, (3,3), activation = 'relu', padding='same')(dec4)

# Salida

salida = layers.Conv2D(filters=2, kernel_size=(1,1), activation='softmax')(dec4)
```

Pasamos ahora a **generar el modelo**. Para ello debemos realizar su **compilación**. En dicha compilación debemos indicar cual es la entrada que toma el modelo y cual es la salida que nos proporciona el modelo. El código es el siguiente:

```
# Generamos el modelo

unet = models.Model(inputs=entrada, outputs=salida)
unet.summary()
```

Una vez que hemos ejecutado la compilación del modelo, se muestra como queda la **estructura de la red neuronal** que hemos construido.

Model: "functional"

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 512, 512, 3)	0	-
conv2d (Conv2D)	(None, 512, 512, 16)	448	input_layer[0][0]
conv2d_1 (Conv2D)	(None, 512, 512, 16)	2,320	conv2d[0][0]
max_pooling2d (MaxPooling2D)	(None, 256, 256, 16)	0	conv2d_1[0][0]
conv2d_2 (Conv2D)	(None, 256, 256, 32)	4,640	max_pooling2d[0][0]
conv2d_3 (Conv2D)	(None, 256, 256, 32)	9,248	conv2d_2[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 128, 128, 32)	0	conv2d_3[0][0]
conv2d_4 (Conv2D)	(None, 128, 128, 64)	18,496	max_pooling2d_1[0][0]
conv2d_5 (Conv2D)	(None, 128, 128, 64)	36,928	conv2d_4[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 64, 64, 64)	0	conv2d_5[0][0]
conv2d_6 (Conv2D)	(None, 64, 64, 128)	73,856	max_pooling2d_2[0][0]
conv2d_7 (Conv2D)	(None, 64, 64, 128)	147,584	conv2d_6[0][0]
max_pooling2d_3 (MaxPooling2D)	(None, 32, 32, 128)	0	conv2d_7[0][0]
conv2d_8 (Conv2D)	(None, 32, 32, 256)	295,168	max_pooling2d_3[0][0]
conv2d_9 (Conv2D)	(None, 32, 32, 256)	590,080	conv2d_8[0][0]
conv2d_transpose (Conv2DTranspose)	(None, 64, 64, 128)	131,200	conv2d_9[0][0]
concatenate (Concatenate)	(None, 64, 64, 256)	0	conv2d_transpose[0][0… conv2d_7[0][0]
conv2d_transpose_1 (Conv2DTranspose)	(None, 64, 64, 128)	295,040	concatenate[0][0]
conv2d_transpose_2 (Conv2DTranspose)	(None, 64, 64, 128)	147,584	conv2d_transpose_1[0]…
conv2d_transpose_3 (Conv2DTranspose)	(None, 128, 128, 64)	32,832	conv2d_transpose_2[0]…
concatenate_1 (Concatenate)	(None, 128, 128, 128)	0	conv2d_transpose_3[0]… conv2d_5[0][0]
conv2d_transpose_4 (Conv2DTranspose)	(None, 128, 128, 64)	73,792	concatenate_1[0][0]
conv2d_transpose_5 (Conv2DTranspose)	(None, 128, 128, 64)	36,928	conv2d_transpose_4[0]…
concatenate_2 (Concatenate)	(None, 256, 256, 64)	0	conv2d_transpose_6[0]… conv2d_3[0][0]
conv2d_transpose_7 (Conv2DTranspose)	(None, 256, 256, 32)	18,464	concatenate_2[0][0]
conv2d_transpose_8 (Conv2DTranspose)	(None, 256, 256, 32)	9,248	conv2d_transpose_7[0]…
conv2d_transpose_9 (Conv2DTranspose)	(None, 512, 512, 16)	2,064	conv2d_transpose_8[0]…
concatenate_3 (Concatenate)	(None, 512, 512, 32)	0	conv2d_transpose_9[0]… conv2d_1[0][0]
conv2d_transpose_10 (Conv2DTranspose)	(None, 512, 512, 16)	4,624	concatenate_3[0][0]
conv2d_transpose_11 (Conv2DTranspose)	(None, 512, 512, 16)	2,320	conv2d_transpose_10[0…
conv2d_10 (Conv2D)	(None, 512, 512, 2)	34	conv2d_transpose_11[0…

Total params: 1,941,122 (7.40 MB)
Trainable params: 1,941,122 (7.40 MB)
Non-trainable params: 0 (0.00 B)

Transcripción automática de símbolos escritos a mano en una imagen.

Compilamos el modelo con el optimizador 'adam' y la función de pérdida 'sparse_categorical_crossentropy'

```
unet.compile(optimizer = 'adam',
             loss = 'sparse_categorical_crossentropy')
```

Una vez que tenemos el modelo compilado pasaremos a su **entrenamiento**, pero antes vamos a dividir los datasets en lotes o batches para no sobrecargar el sistema en exceso y tengamos problemas en dicho entrenamiento.

```
# Definimos el batch size

BATCH_SIZE_TRAIN = 16
BATCH_SIZE_VAL = 16

# Creamos los lotes, aun no leemos las imagenes

train_batch = dataset_train.map(preprocesar_imagen).batch(BATCH_SIZE_TRAIN)
val_batch = dataset_val.map(preprocesar_imagen).batch(BATCH_SIZE_VAL)
```

Ahora ya estamos en disposición para realizar el entrenamiento de nuestro modelo. Dicho **entrenamiento lo realizaremos durante 50 ciclos o epochs , con lotes de tamaño 16.**

```
# Pasamos a entrenar el modelo
unet.fit(train_batch,
         validation_data=val_batch,
         epochs=50,
         verbose=2)
```

El resultado del entrenamiento ha sido el siguiente.

```
Epoch 1/50
163/163 - 174s - 1s/step - loss: 0.1225 - val_loss: 0.0602
Epoch 2/50
163/163 - 39s - 239ms/step - loss: 0.0456 - val_loss: 0.0477
Epoch 3/50
163/163 - 10s - 63ms/step - loss: 0.0407 - val_loss: 0.0598
Epoch 4/50
163/163 - 5s - 33ms/step - loss: 0.0359 - val_loss: 0.0778
Epoch 5/50
163/163 - 10s - 63ms/step - loss: 0.0364 - val_loss: 0.0387
Epoch 6/50
163/163 - 5s - 33ms/step - loss: 0.0315 - val_loss: 0.1170
Epoch 7/50
163/163 - 5s - 33ms/step - loss: 0.0286 - val_loss: 0.0317
Epoch 8/50
163/163 - 5s - 33ms/step - loss: 0.0261 - val_loss: 0.0283
Epoch 9/50
163/163 - 10s - 63ms/step - loss: 0.0224 - val_loss: 0.0226
Epoch 10/50
163/163 - 6s - 34ms/step - loss: 0.0213 - val_loss: 0.0374
Epoch 11/50
163/163 - 10s - 62ms/step - loss: 0.0510 - val_loss: 0.0392
Epoch 12/50
163/163 - 5s - 33ms/step - loss: 0.0271 - val_loss: 0.0235
Epoch 13/50
163/163 - 5s - 32ms/step - loss: 0.0246 - val_loss: 0.0234
Epoch 14/50
163/163 - 10s - 64ms/step - loss: 0.0167 - val_loss: 0.0229
Epoch 15/50
163/163 - 10s - 63ms/step - loss: 0.0229 - val_loss: 0.0221
Epoch 16/50
163/163 - 5s - 32ms/step - loss: 0.0174 - val_loss: 0.0208
Epoch 17/50
163/163 - 5s - 33ms/step - loss: 0.0161 - val_loss: 0.0198
Epoch 18/50
163/163 - 10s - 62ms/step - loss: 0.0150 - val_loss: 0.0202
Epoch 19/50
163/163 - 5s - 33ms/step - loss: 0.0191 - val_loss: 0.0179
Epoch 20/50
163/163 - 10s - 63ms/step - loss: 0.0268 - val_loss: 0.0376
Epoch 21/50
163/163 - 10s - 62ms/step - loss: 0.0180 - val_loss: 0.0207
Epoch 22/50
163/163 - 10s - 62ms/step - loss: 0.0148 - val_loss: 0.0205
Epoch 23/50
163/163 - 5s - 33ms/step - loss: 0.0159 - val_loss: 0.0196
Epoch 24/50
163/163 - 10s - 63ms/step - loss: 0.0153 - val_loss: 0.0180
Epoch 25/50
163/163 - 5s - 33ms/step - loss: 0.0375 - val_loss: 0.0273
Epoch 26/50
163/163 - 5s - 33ms/step - loss: 0.0195 - val_loss: 0.0221
Epoch 27/50
163/163 - 10s - 63ms/step - loss: 0.0129 - val_loss: 0.0212
Epoch 28/50
163/163 - 5s - 33ms/step - loss: 0.0122 - val_loss: 0.0213
Epoch 29/50
163/163 - 10s - 63ms/step - loss: 0.0112 - val_loss: 0.0210
Epoch 30/50
163/163 - 5s - 33ms/step - loss: 0.0120 - val_loss: 0.0157
Epoch 31/50
163/163 - 10s - 62ms/step - loss: 0.0432 - val_loss: 0.0336
Epoch 32/50
163/163 - 10s - 62ms/step - loss: 0.0247 - val_loss: 0.0238
Epoch 33/50
163/163 - 10s - 63ms/step - loss: 0.0188 - val_loss: 0.0198
Epoch 34/50
163/163 - 5s - 33ms/step - loss: 0.0144 - val_loss: 0.0197
Epoch 35/50
163/163 - 10s - 63ms/step - loss: 0.0150 - val_loss: 0.0211
Epoch 36/50
163/163 - 10s - 62ms/step - loss: 0.0179 - val_loss: 0.0217
Epoch 37/50
163/163 - 10s - 63ms/step - loss: 0.0147 - val_loss: 0.0204
Epoch 38/50
163/163 - 5s - 33ms/step - loss: 0.0126 - val_loss: 0.0198
Epoch 39/50
163/163 - 10s - 63ms/step - loss: 0.0105 - val_loss: 0.0189
Epoch 40/50
163/163 - 10s - 62ms/step - loss: 0.0098 - val_loss: 0.0179
Epoch 41/50
163/163 - 10s - 64ms/step - loss: 0.0117 - val_loss: 0.0190
Epoch 42/50
163/163 - 5s - 33ms/step - loss: 0.0108 - val_loss: 0.0201
Epoch 43/50
163/163 - 10s - 63ms/step - loss: 0.0289 - val_loss: 0.0274
Epoch 44/50
163/163 - 6s - 34ms/step - loss: 0.0203 - val_loss: 0.0202
Epoch 45/50
163/163 - 10s - 62ms/step - loss: 0.0134 - val_loss: 0.0193
Epoch 46/50
163/163 - 10s - 62ms/step - loss: 0.0166 - val_loss: 0.0195
Epoch 47/50
163/163 - 10s - 64ms/step - loss: 0.0118 - val_loss: 0.0181
Epoch 48/50
163/163 - 10s - 63ms/step - loss: 0.0119 - val_loss: 0.0244
Epoch 49/50
163/163 - 10s - 63ms/step - loss: 0.0119 - val_loss: 0.0175
Epoch 50/50
163/163 - 5s - 33ms/step - loss: 0.0112 - val_loss: 0.0168
<keras.src.callbacks.history.History at 0x78c6901a42e0>
```

En el resultado que se nos proporciona en cada ciclo o época, **la función de pérdida se hace más pequeña** y cada vez se aproxima más 0 y **diferencia en la validación también decrece** de manera muy pareja con la función de pérdida. Esto son síntomas de que **no estamos tendiendo overfiting o sobreentrenamiento.**

Una vez entrenado el modelo pasaremos a **probarlo con el dataset de prueba**. Para ello construiremos un solo lote de prueba, ya que su tamaño no es muy grande.

```
# Generamos el batch de test con todas las imagenes que contiene el dataset de test
test_batch = dataset_test.map(preprocesar_imagen).batch(len(dataset_test))
```

Realicemos ahora las **predicciones**.

```
print(prediciones.shape)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 42s 42s/step
(21, 512, 512, 2)
```

La salida que nos da el modelo son 21 arrays de Numpy de 512x512, que es el tamaño de cada imagen de entrada, y 2 capas en profundidad. Cada capa corresponde a la clase a la que puede pertenecer cada píxel de la imagen de entrada, en nuestro caso puede pertenecer al fondo o al precinto. En el array correspondiente a cada clase, cada posición contiene la probabilidad de que ese píxel pertenezca a esa clase. Es decir valores cercanos a 1 indican que el píxel pertenece a esa clase y valores cercanos a 0 indica que el píxel no pertenece a esa clase.

Para agrupar todas las clases en una capa usaremos el método *argmax* de Numpy.

```
# Máscaras de segmentación
# axis = 3: buscar el máximo de cada plano
mascaras = np.argmax(prediciones,axis=3)
print(mascaras.shape)
```

```
(21, 512, 512)
```

De esta forma obtenemos 21 arrays de Numpy de 512x512 y 1 sola capa que aglutina todas las clases a las que puede pertenecer un píxel.

Para poder visualizar la salida que proporciona el modelo, lo primero que debemos hacer es deshacer el lote del dataset de prueba, mediante el método *unbatch*. Obteniendo así una lista con las imágenes y otra con las máscaras del dataset.

```
test_list = list(test_batch.unbatch().as_numpy_iterator())

# Almacenar imágenes y máscaras de prueba como arreglos de NumPy
imgs = np.asarray([item[0] for item in test_list])
msks = np.asarray([item[1] for item in test_list])

print(imgs.shape)
print(msks.shape)
```

```
(21, 512, 512, 3)
(21, 512, 512, 1)
```

Y ahora visualizamos la imagen, la máscara original y la máscara inferida.

```
# Mostrar imagen, máscara y predicción

# Escoger una imagen aleatoriamente
id = np.random.randint(0,len(dataset_test))

# Generar figura
fig, (ax1, ax2, ax3) = plt.subplots(1,3, figsize=(10,6))

# Imagen original
ax1.imshow(imgs[id])
ax1.set_title('Imagen original')
ax1.axis('off')

# Máscara original
ax2.imshow(msks[id])
ax2.set_title('Máscara original')
ax2.axis('off')

# Máscara predicha
ax3.imshow(mascaras[id])
ax3.set_title('Máscara predicha')
ax3.axis('off');
```

Fig. 1.14. Comparación de la imagen original, la máscara de segmentación y la máscara predicha por el modelo.

Se puede observar que la **precisión a la hora de inferir la máscara es muy buena**. La red es muy precisa aun teniendo un dataset para entrenamiento y validación bastante pequeño.

2.- Detección de objetos en una imagen

2.1.- ¿Qué es la detección de objetos?

La detección de objetos **es una tecnología de visión artificial dedicada a detectar instancias de objetos semánticos** de una o más clases en vídeos o imágenes digitales. Los sistemas de detección de objetos son capaces de reconocer caras, personas, edificios, coches, etc. La detección de objetos **tiene aplicaciones en muchas áreas de visión artificial, como pueden ser la conducción autónoma, reconocimiento de imágenes satelitales, diagnósticos sobre imágenes médicas, etc**.

La detección de objetos constituye uno de los componentes clave para algoritmos de Machine Learning dedicados a la visión artificial.

En la siguiente imagen podemos observar como el sistema de visión artificial de detección de objetos, recuadra cada objeto que aparece en la imagen asignándole la etiqueta de la clase a la que pertenece.

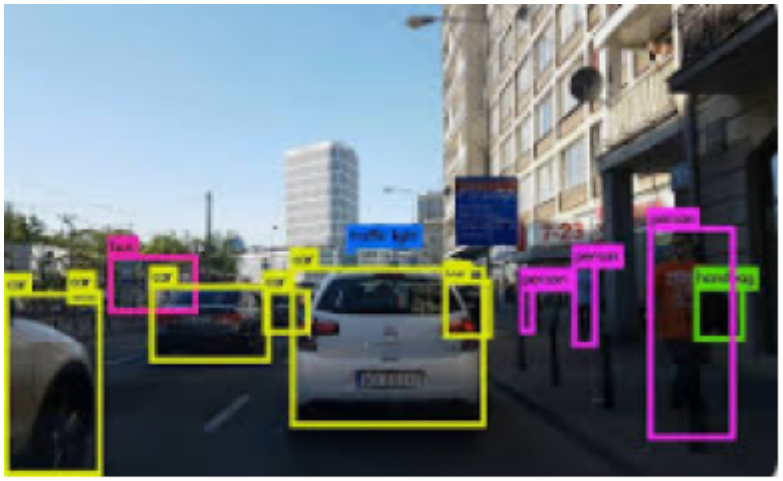


Fig. 2.1. Ejemplo del resultado de la detección de objetos en una imagen.

2.2.- Estrategia para la detección de objetos.

En cuanto a nuestro problema, para realizar la detección de los distintos dígitos que aparecen dentro del área segmentada, obtenida mediante la aplicación del modelo descrito en el punto anterior, me planteé una estrategia de detección y clasificación en un solo paso. Es decir, implementar un modelo que fuese capaz de detectar la localización de cada dígito y a la vez le asignase la clase a la que pertenece. En otras palabras que si hubiese un 1 en la imagen el modelo fuese capaz de enmarcarlo y asignarle la clase '1'.

Para ello construí un dataset personalizado con más de 100 imágenes, en las cuales etiqueté cada dígito incluido en el área designada usando una herramienta de etiquetado.

Una vez construido el dataset, entrené varios modelos, basados en redes convolucionales y redes transformer, entre otros un modelo Faster RCNN o un modelo RT-DETR, obteniendo unos resultados no esperados. Los modelos no realizaban una detección ni clasificación lo suficientemente buena y precisa como para dar esta estrategia por válida.

Llegado a este punto decidí cambiar de estrategia, pasando a una estrategia en dos pasos. El primero sería realizar la detección de los dígitos y una vez detectados y localizados en un segundo paso clasificarlos para conocer la clase a la que pertenecen. Los resultados obtenidos de las pruebas realizadas a partir de esta estrategia fueron positivos, pasando a realizar su implementación.

2.2.1.- Estrategia en dos pasos

Para llevar a cabo la estrategia en dos pasos debemos realizar las siguientes fases:

Dentro de la detección estarían:

- **Construcción de un dataset personalizado,** donde se etiqueten todos los dígitos que aparecen en el área segmentada como perteneciente a una única clase, 'number'.

- **Elección del modelo** para la detección de dichos dígitos. Decidí utilizar un **modelo pre-entrenado,** basándome en el concepto del **aprendizaje por transferencia**. La elección fue YOLO V8.

- Con las coordenadas de los bounding boxes proporcionadas por el modelo YOLO V8, se pasaría a **recortar de la imagen todos los objetos detectados**, es decir los dígitos, para realizar su posterior clasificación.

Y en la fase de clasificación tendríamos que realizar las siguientes tareas:

- **Preprocesar** los recortes de los dígitos **para prepararlos para su clasificación** mediante un modelo de clasificación.

- C**onstruir un modelo de clasificación,** mediante redes convolucionales para la clasificación de cada dígito detectado.

- **Composición de la salida**. En nuestro caso el código del contenedor y la fecha de precintado, a partir de los dígitos detectados.

2.3.- Implementación del detector de objetos

2.3.1.- Dataset personalizado

Para poder utilizar un modelo pre-entrenado para nuestros fines de detección de objetos, debemos entrenarlo sobre un **dataset personalizado, en el que se le indique los objetos que se deben detectar.** Esta es la base del concepto de aprendizaje por transferencia.

Para dicha tarea nos centraremos en nuestra **selección de imágenes,** a las cuales les aplicaremos su **máscara c**orrespondiente. De esta forma, conseguiremos que el modelo trabaje sobre unas imágenes **similares a las que les llegará desde el modelo de segmentación anteriormente expuesto.**

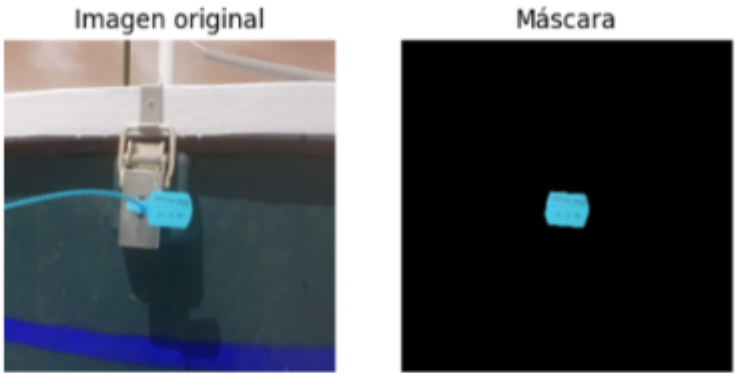


Fig 2.2. Imagen original con su máscara de segmentación.

Para aplicar las máscaras necesitaremos ejecutar el siguiente script.

1.- Hacemos la importación para poder montar la unidad de

Google Drive y poder haceder a los datos para su tratamiento

```
[ ]  from google.colab import drive
     drive.mount('/content/gdrive/')
```

```
Mounted at /content/gdrive/
```

2.- Establecemos las rutas para acceder a las imagenes y las máscaras.

También establecemos la ruta donde se almacenarán las máscaras generadas.

```
[ ]  ruta_img ='/content/gdrive/MyDrive/dataset_PFM/proyecto_unet/images/'
     ruta_msk = '/content/gdrive/MyDrive/dataset_PFM/proyecto_unet/masks/'
     ruta_out = '/content/gdrive/MyDrive/dataset_PFM/proyecto_unet/dts_msks/'
```

4.- Creamos una función auxiliar para obtenemos la lista con

los archivos de imagenes que hay en la ruta establecida anteriormente

```
#Pasamos una ruta y nos devuelve los ficheros de imagen contenidos en esa ruta.
def CargarImagenes(path):
    imagenes = []
    contenido = os.listdir(path)
    for fichero in contenido:
        if os.path.isfile(os.path.join(path, fichero)) and
         (fichero.endswith('.jpg') or fichero.endswith('.png')):
            imagenes.append(fichero)

    return imagenes
```

```
[ ]  imagenes = CargarImagenes(ruta_img)
     mascaras = CargarImagenes(ruta_msk)
```

5.- Cargamos el archivo .csv que contiene la relacción entre

los archivos de imagenes y sus máscaras

```
[ ]  rel_archivos = pd.read_csv(ruta_msk + 'rel_archivos.csv')
```

6.- Comprobamos que la lectura se ha realizado correctamente

```
[ ]  print(rel_archivos.loc[1,'imgs'])
     print(rel_archivos.loc[1,'msks'])
```

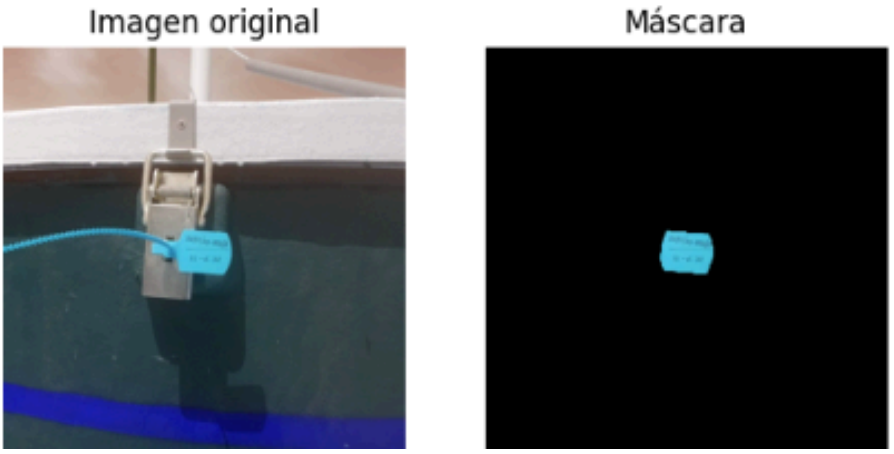
```
imagen1.jpg
mascara1.jpg
```

7.- Realizamos una muestra de la funcionalidad que perseguimos sobre una imagen y visualizamos la salida obtenida.

```
img = cv2.imread(ruta_img + rel_archivos.loc[0,'imgs'])
msk = cv2.imread(ruta_msk + rel_archivos.loc[0,'msks'],cv2.IMREAD_GRAYSCALE )
dts_mask = cv2.bitwise_and(img,img,mask=msk)
```

```
[ ] fig, ax = plt.subplots(1,2)
    ax[0].set_title('Imagen original')
    ax[0].axis('off')
    ax[0].imshow(img)
    ax[1].set_title('Máscara')
    ax[1].axis('off')
    ax[1].imshow(dts_mask)
```

```
<matplotlib.image.AxesImage at 0x7bcc9127cf10>
```



8.- Visto que la prueba se ha realizado con exito, procederemos a realizarlo sobre todas las imagenes

```
[ ] for i in range(len(rel_archivos)):
      img = cv2.imread(ruta_img + rel_archivos.loc[i,'imgs'])
      msk = cv2.imread(ruta_msk + rel_archivos.loc[i,'msks'],cv2.IMREAD_GRAYSCALE )
      dts_mask = cv2.bitwise_and(img,img,mask=msk)
      cv2.imwrite(ruta_out + f'dts_mask{i}.jpg', dts_mask)
```

Etiquetado de imágenes del dataset.

Una vez que tenemos la base de imágenes de máscara pasaremos a realizar el etiquetado, para ello utilizaremos Roboflow.

Roboflow es una **herramienta online** que entre otras cosas permite el **etiquetado de imágenes para detección de objetos y segmentación semántica**. Nosotros lo utilizaremos para la primera tarea.

Para realizar el etiquetado en Roboflow, primero debemos registrarnos de manera gratuita. Una vez registrados, accedemos a la herramienta y creamos un espacio de trabajo que se denomina workspace. Dentro de este workspace es donde crearemos nuestro proyecto de etiquetado. Aquí es donde guardaremos las imágenes sobre las que realizaremos el etiquetado y que formarán el dataset.

Aquí tenemos una vista general del escritorio.

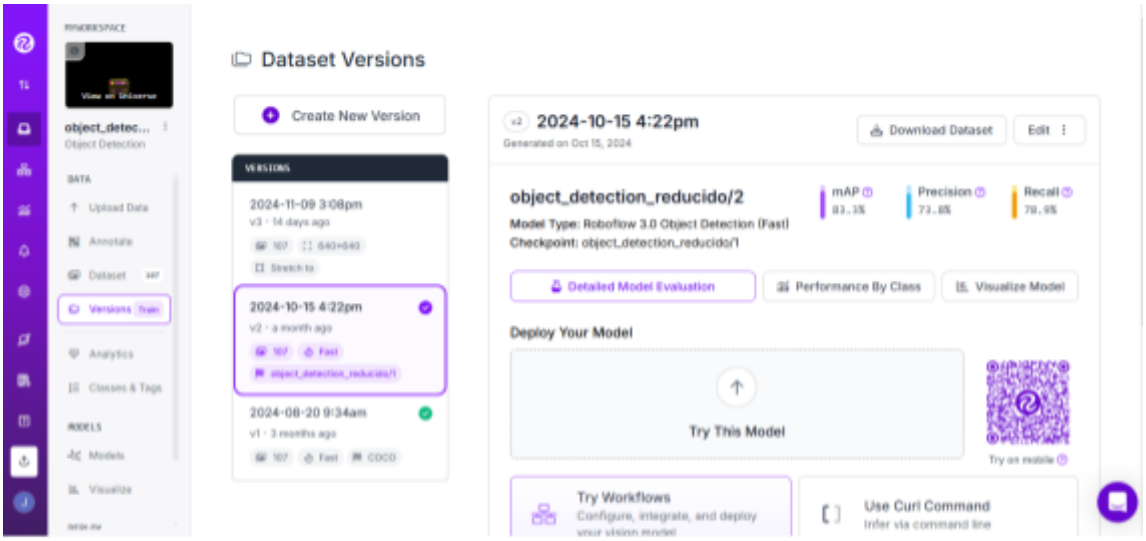


Fig 2.3. Escritorio de Roboflow

Roboflow, **permite tener varias versiones de un mismo proyecto,** funcionalidad que nos ha venido bien para poder obtener dos datasets distintos sobre un mismo conjunto de imágenes. En una versión realizamos el etiquetado para la estrategia de detección y clasificación en un solo paso y en otra versión realizamos el etiquetado para la estrategia en dos pasos.

Accederemos a la pestaña Dataset para realizar el etiquetado y la división del mismo en la parte de train, test y validación.

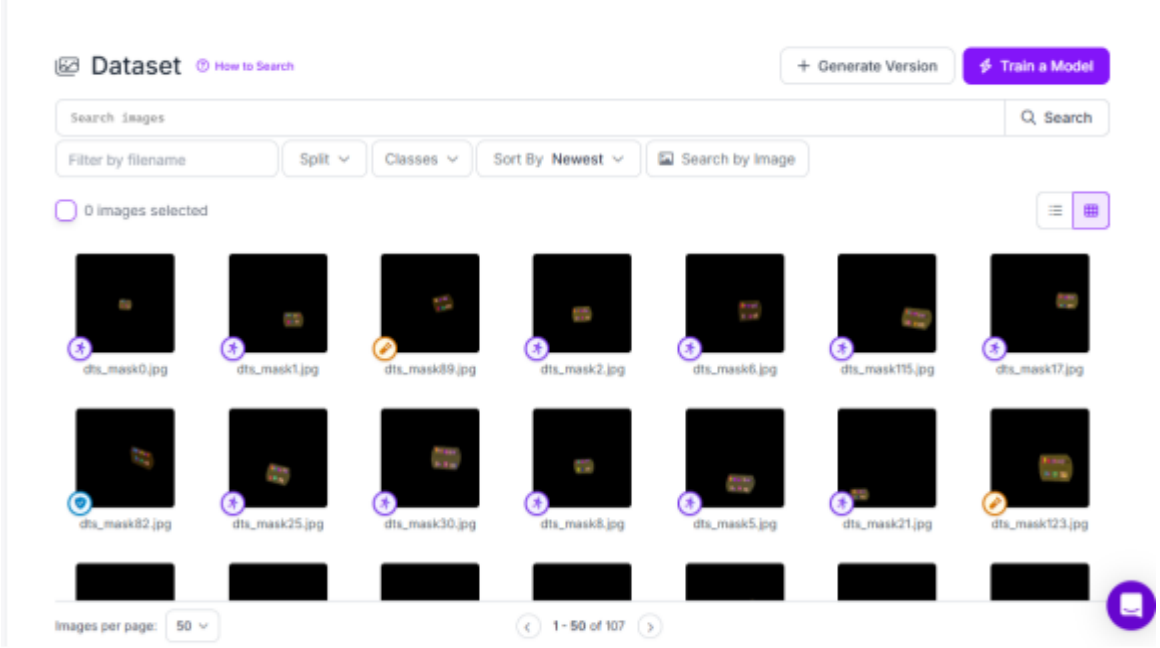


Fig 2.4. Dataset en Roboflow.

Aquí seleccionamos una imagen y comenzamos a trabajar sobre ella el etiquetado.

En la zona de edición, seleccionamos cada zona de la imagen y le asignamos la clase a la que pertenece.

El siguiente ejemplo, pertenece al etiquetado de las imágenes para la estrategia de detección y clasificación en un paso. En este caso se ve mucho más claro cómo cada zona pertenece a una clase diferente.

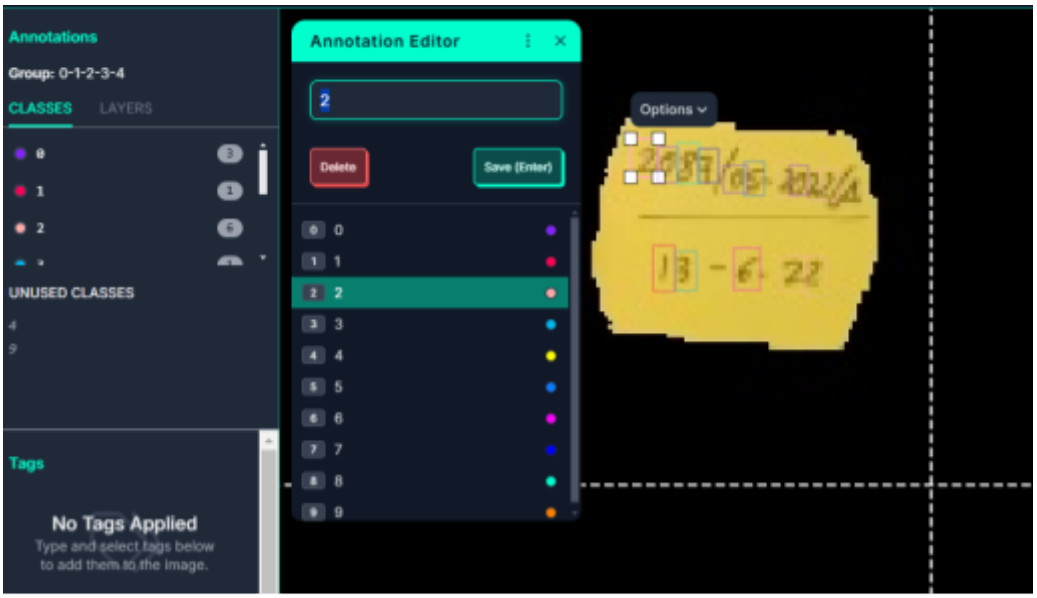


Fig 2.5. Etiquetado multiclase

Un ejemplo de etiquetado para la estrategia en dos pasos sería este.

Fig 2.6. Etiquetado clase única

En él, vemos como cada dígito pertenece a una única clase, ‘number’, de este modo podemos detectar cada dígito sin preocuparnos de qué dígito se trata, dejando este trabajo para la etapa siguiente de clasificación.

2.3.2.- Modelo de detección de objetos.

Para implementar el modelo de detección de objetos nos basaremos en el concepto de aprendizaje por transferencia.

El aprendizaje por transferencia es una t**écnica en la que un modelo desarrollado para una tarea concreta se reutiliza como punto de partida para un modelo sobre una segunda tarea.** Es decir, se vuelven a aplicar los componentes de un modelo de aprendizaje automático preentrenado a nuevos modelos destinados a algo diferente pero relacionado.

El concepto es similar a la forma en que los seres humanos aprenden nuevas habilidades. Un ejemplo podría ser el siguiente, un guitarrista decide aprender a tocar el ukelele. Su experiencia previa con la guitarra acelerará su proceso de aprendizaje. Esto se debe a que muchas de las habilidades y conocimientos

necesarios para tocar la guitarra, como las posiciones de los dedos, los patrones de rasgueo, la teoría musical y el ritmo, también son aplicables para tocar el ukelele.

En IA, el aprendizaje por transferencia nos **permite aprovechar el entrenamiento previo para resolver problemas nuevos relacionados de forma más eficiente**, reduciendo así el tiempo y los recursos computacionales.

En nuestro caso aplicaremos el aprendizaje por transferencia aplicado a el modelo preentrenado YOLO V8. Aprovecharemos todas sus bondades para aplicarlo en la detección de dígitos.

El algoritmo You Only Look Once (YOLO), es un **sistema de código abierto para detección de objetos en tiempo real,** el cual hace uso de una única red neuronal convolucional para detectar objetos en imágenes. Para su funcionamiento, la red neuronal divide la imagen en regiones, prediciendo cuadros de identificación y probabilidades por cada región; las cajas son ponderadas a partir de las probabilidades predichas. El algoritmo aprende representaciones generalizables de los objetos, permitiendo un bajo error de detección para entradas nuevas, diferentes al conjunto de datos de entrenamiento.

El modelo se implementó como una red neuronal convolucional y fue evaluado en el set de datos para detección de PASCAL VOC. Las capas convolucionales iniciales de la red se encargan de la extracción de características de la imagen, mientras que las capas de conexión completa predicen la probabilidad de salida y las coordenadas del objeto.

La red tiene 24 capas convolucionales seguidas por 2 capas de conexión completa; esta hace uso de capas de reducción de 1x1 seguidas de capas convolucionales de 3x3. El modelo Fast YOLO hace uso de una red neuronal de 9 capas. La salida final del modelo de predicción es de 7x7x30.

Para el pre entrenamiento, se hace uso de las primeras 20 capas convolucionales seguidas de una capa promediadora de grupos y una capa de conexión completa; posteriormente se convierte el modelo resultante para la detección de objetos.

Para la implementación de la detección de objetos se agregan 4 capas convolucionales y 2 capas de conexión completa con ponderaciones aleatoriamente inicializadas. La última capa de la red predice probabilidades de clases y coordenadas para las cajas de identificación; para este paso se normaliza la altura y ancho de la caja de identificación con respecto a los parámetros de la imagen, de tal manera que sus valores se mantengan entre 0 y 1. En la última capa se usa una función de activación, utilizando un error de suma cuadrada para la optimización de la salida.

El algoritmo YOLO **predice múltiples cuadros de identificación por cuadrícula de celdas.** En tiempo de entrenamiento se busca tener un solo cuadro de identificación por objeto, lo cual se consigue a partir de las probabilidades predichas para cada cuadro, manteniendo el de mayor alta probabilidad.

El modelo YOLO V8 está desarrollado por Ultralytics, por ello debemos seguir los pasos que han propuestos para el uso de su modelo.

Como primer paso en la implementación es la instalación del paquete de Ultralytics y la importación de las librerías necesarias

```
# Pip install method

!pip install ultralytics==8.2.103 -q

from IPython import display
display.clear_output()

import ultralytics
ultralytics.checks()
```

```
Ultralytics YOLOv8.2.103 🚀 Python-3.10.12 torch-2.5.0+cu121 CUDA:0 (NVIDIA L4, 22700MiB)
Setup complete ✅ (12 CPUs, 53.0 GB RAM, 32.3/112.6 GB disk)
```

```
from ultralytics import YOLO

from IPython.display import display, Image
```

Transcripción automática de símbolos escritos a mano en una imagen.

Para **acceder al dataset almacenado en Roboflow desde Colab, debemos generar una key** en dicha plataforma.

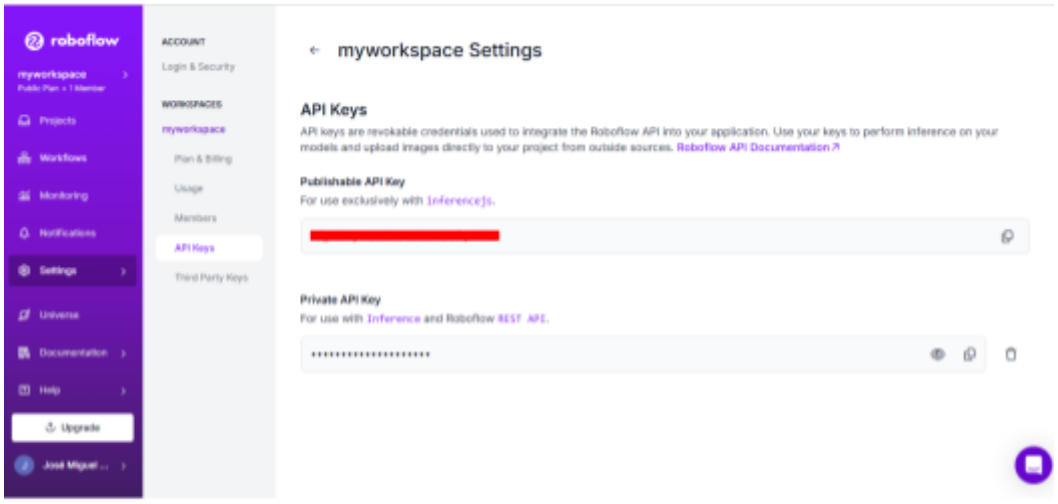


Fig. 2.7. Generando una key en Roboflow.

Una vez que tenemos la key generada en Roboflow **debemos añadirla en Colab**.

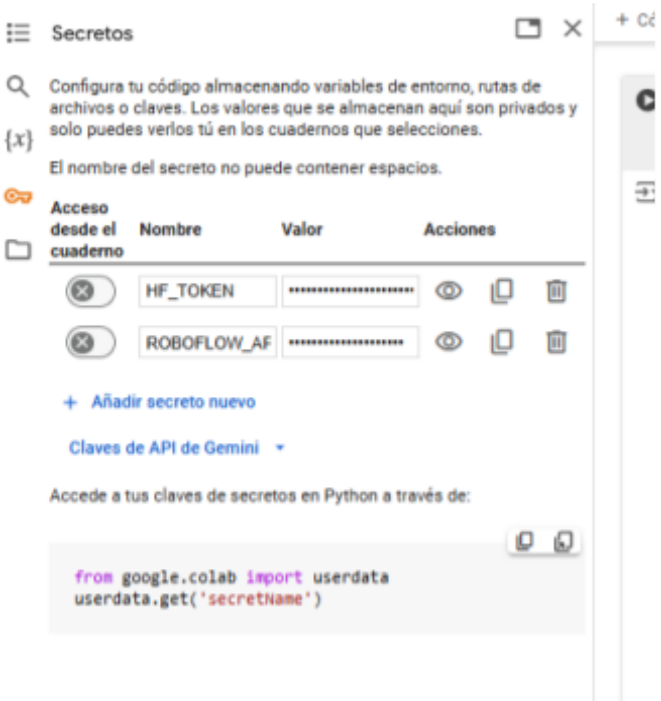


Fig. 2.8. Añadimos la key de Roboflow en Colab

De esta manera nos aseguramos de que solo nosotros **accederemos de una forma privada a los datos del dataset personalizado**.

Transcripción automática de símbolos escritos a mano en una imagen.

Pasaremos ahora a instalar el paquete de Roboflow y a importar las dependencias necesarias para la utilización del dataset en el código.

```
!pip install roboflow==1.1.48 --quiet

import roboflow

roboflow.login()

rf = roboflow.Roboflow()

project = rf.workspace("myworkspace-pgngk").project("object_detection_reducido")
dataset = project.version(3).download("yolov8")
```

```
/content/datasets
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 80.3/80.3 kB 10.0 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 66.8/66.8 kB 6.5 MB/s eta 0:00:00
visit https://app.roboflow.com/auth-cli to get your authentication token.
Paste the authentication token here: ··········
loading Roboflow workspace...
loading Roboflow project...
Downloading Dataset Version Zip in object_detection_reducido-3 to yolov8:: 100%|██████████| 1121/1121 [00:01<00:00, 867.16it/s]

Extracting Dataset Version Zip to object_detection_reducido-3 in yolov8:: 100%|██████████| 226/226 [00:00<00:00, 9253.07it/s]
```

Para trabajar con el modelo YOLO V8 usaremos la interfaz de línea de comandos YOLO (CLI). Esta línea de comandos nos permite ejecutar comandos sencillos de una sola línea sin necesidad de un entorno Python, aunque nosotros trabajamos sobre el entorno Google Colab . Basta con ejecutar todas las tareas desde el terminal con el comando *yolo*.

Para entrenar el modelo pre-entrenado, haremos una llamada proporcionándole nuestro dataset personalizado y haremos que realice el entrenamiento a lo largo de 100 epochs. Usaremos la versión de YOLO V8 más potente para la detección de objetos, ***yolov8x.pt***

```
!yolo task=detect mode=train model=yolov8x.pt data={dataset.location}/data.yaml epochs=100 imgsz=800 plots=True
```

Una vez entrenado el modelo, debemos **comprobar cómo de bueno es en las predicciones.** Para ello debemos observar las diferentes **métricas de precisión** que nos proporciona el modelo.

```
Validating runs/detect/train2/weights/best.pt...
Ultralytics YOLOv8.2.103 🚀 Python-3.10.12 torch-2.5.0+cu121 CUDA:0 (NVIDIA L4, 22700MiB)
Model summary (fused): 268 layers, 68,124,531 parameters, 0 gradients, 257.4 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% 1/1 [00:00<00:00,  1.80it/s]
                   all         21        309      0.985      0.997      0.995      0.537
Speed: 0.3ms preprocess, 20.3ms inference, 0.0ms loss, 0.9ms postprocess per image
```

En este caso **la precisión del modelo es buena para realizar la detección de objetos** de forma precisa.

Para comprobar que el modelo no sufre overfitting, realizamos la evaluación del modelo con el dataset de test.

```
!yolo task=detect mode=val model={HOME}/runs/detect/train/weights/best.pt data={dataset.location}/data.yaml
```

```
/content
Ultralytics YOLOv8.2.103 🚀 Python-3.10.12 torch-2.5.0+cu121 CUDA:0 (NVIDIA L4, 22700MiB)
Model summary (fused): 168 layers, 11,125,971 parameters, 0 gradients, 28.4 GFLOPs
val: Scanning /content/datasets/object_detection_reducido-3/valid/labels.cache... 21 images, 0 backgrounds, 0 corrupt: 100% 21/21 [00:00<?, ?it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% 2/2 [00:01<00:00,  1.00it/s]
                   all         21        309      0.986       0.99      0.994      0.541
Speed: 5.2ms preprocess, 44.5ms inference, 0.0ms loss, 31.1ms postprocess per image
Results saved to runs/detect/val3
💡 Learn more at https://docs.ultralytics.com/modes/val
```

Los resultados de **las métricas en evaluación son similares a las obtenidas en el entrenamiento, por lo que podemos decir que el modelo no sufre overfitting** y debe tener buen desempeño.

Pasemos a realizar predicciones con el modelo. Con el siguiente código las realizamos.

```
%cd {HOME}
!yolo task=detect mode=predict model={HOME}/runs/detect/train/weights/best.pt conf=0.25 source={dataset.location}/test/images save=True
```

Visualizemos una como ejemplo para comprobar que se está realizando correctamente la detección de los dígitos de la imagen.

El código que implementa esta visualización es el que se muestra a continuación.

```
import glob
from IPython.display import Image, display

# Define the base path where the folders are located
base_path = '/content/runs/detect/'

# List all directories that start with 'predict' in the base path
subfolders = [os.path.join(base_path, d) for d in os.listdir(base_path)
              if os.path.isdir(os.path.join(base_path, d)) and d.startswith('predict')]

# Find the latest folder by modification time
latest_folder = max(subfolders, key=os.path.getmtime)

image_paths = glob.glob(f'{latest_folder}/*.jpg')[:3]

# Display each image
for image_path in image_paths:
    display(Image(filename=image_path, width=600))
    print("\n")
```

Esta es la imágen donde se incluyen las detecciones realizadas por el modelo de detección.

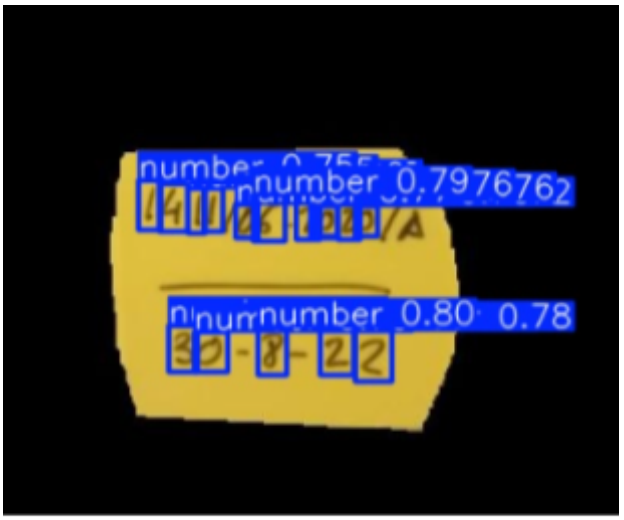


Fig. 2.9. Objetos detectados por el modelo.

El modelo realiza la **detección de todos los dígitos** que se encuentran dentro de la zona segmentada en la máscara. El modelo muestra también el nombre de la clase a la que pertenece cada detección y la probabilidad con la que el modelo predice su pertenencia a dicha clase.

Una vez verificado que **el modelo realiza correctamente la detección**, pasaremos a guardar el modelo para poderlo utilizar en el código de detección que debemos implementar posteriormente.

Cuando entrenamos un modelo YOLO se almacena de manera automática un archivo best.pt correspondiente al modelo ya entrenado, dentro de una carpeta en la ruta ./ultralytics/runs/detect . Por cada entrenamiento se genera un archivo best.pt.

Nosotros lo guardaremos en una carpeta de Google Drive para su posterior utilización.

2.4.- Aplicación del modelo de detección de objetos.

Comenzamos escribiendo las rutas donde se encuentran el modelo entrenado con nuestro dataset personalizado y la imagen de máscara sobre la cual trabajaremos a modo de prueba la detección y la posterior clasificación.

```
path = '/content/drive/MyDrive/dataset_PFM/modelo_custom_yolov8/'
file_model = path + 'best.pt'
file_img1 = path + 'im1.jpg'
```

Ahora debemos instalar el paquete de Ultralytics e importar las dependencias oportunas,

```
!pip install ultralytics==8.2.103 -q
```

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

Cargamos la imagen para empezar con el procesado.

```
# Load the image using cv2.imread()
image = cv2.imread(image_path)

# Convert the image from BGR (used by OpenCV) to RGB (used by matplotlib)
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
```

Visualizamos la imagen que hemos cargado.

```
# Display the image using plt.imshow()
plt.imshow(image_rgb)
plt.title('Image')  # Add a title to the plot
plt.axis('off')  # Hide the axis ticks and labels
plt.show()
```

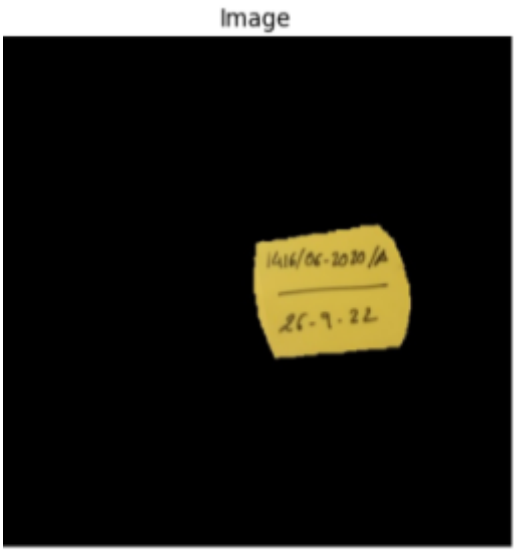


Fig. 2.10. Máscara de imagen

Importamos el modelo de detección y **realizamos las predicciones.**

```
import cv2
from ultralytics import YOLO
import matplotlib.pyplot as plt

# Load your YOLO model
model = YOLO(file_model)
```

```
results = model.predict(source=[image])
```

Una vez realizadas las predicciones, nos dispondremos a **capturar los bounding boxes correspondientes a los dígitos detectados por el modelo.**

Con el siguiente código conseguimos capturar los bounding boxes y los visualizamos sobre la imagen de partida.

```
# Draw bounding boxes on im1
for r in results:
    for *xyxy, conf, cls in r.boxes.data:  # Access bounding box data
        x1, y1, x2, y2 = map(int, xyxy)  # Convert coordinates to integers
        cv2.rectangle(image, (x1, y1), (x2, y2), (0, 255, 0), 2)  # Draw rectangle

# Display the image with bounding boxes using matplotlib
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))  # Convert BGR to RGB for matplotlib
plt.show()
```

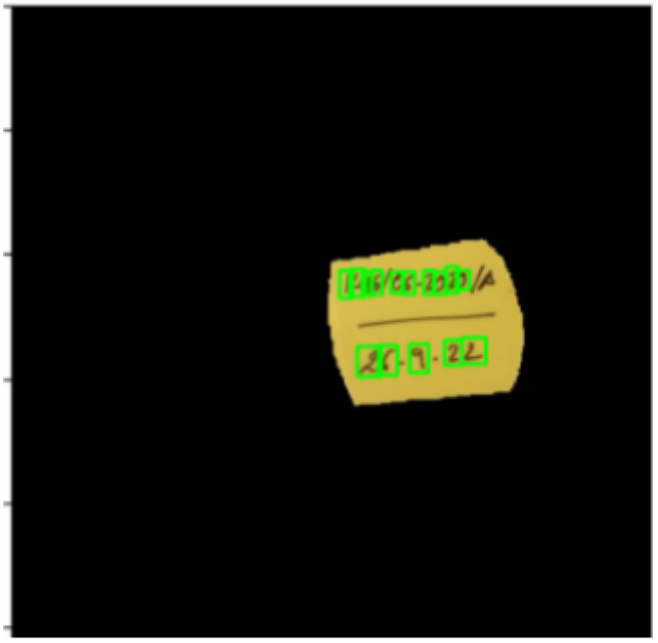


Fig. 2.11. Predicción de detección de objetos

A partir de estos bounding boxes **calculamos las coordenadas** de cada uno.

```
list_coordenadas = []
media = 0

for r in results:
    for box in r.boxes:
        coordinates = (box.xyxy).tolist()[0]
        list_coordenadas.append(coordinates)
```

A partir de estas coordenadas **ordenaremos los bounding tanto en el eje x como en el eje y**. De esta forma podemos separar los bounding boxes pertenecientes al código del contenedor y los que pertenecen a la fecha.

```
sorted_by_x = sorted(list_coordenadas, key=lambda coord: coord[0])
```

```
for coordenada in list_coordenadas:
  media = media + coordenada[1]
media = media / len(list_coordenadas)
```

```
codigo_contenedor = []
fecha = []

for coord in sorted_by_x:
  if coord[1]< media:
    codigo_contenedor.append(coord)
  else:
    fecha.append(coord)
```

Una vez que tenemos las coordenadas de todos los dígitos, tanto del código del contenedor como de la fecha, nos dispondremos a obtener cada dígito de la imagen original.

```
lista_img_nums_codigo = []

for codigo in codigo_contenedor:
  # Define the coordinates of the region to crop
  x1 = round(codigo[0])
  y1 = round(codigo[1])  # Top-left corner coordinates
  x2 = round(codigo[2])
  y2 = round(codigo[3]) # Bottom-right corner coordinates
  # Crop the image using array slicing
  cropped_image_codigo = imagen[y1:y2, x1:x2]
  resized_image_codigo = cv2.resize(cropped_image_codigo, (28, 28), interpolation=cv2.INTER_AREA)
  lista_img_nums_codigo.append(resized_image_codigo)
```

```
fig, ax = plt.subplots(figsize=(15,15),ncols=len(lista_img_nums_codigo),nrows=1)
for i, img in enumerate(lista_img_nums_codigo):

    ax[i].set_xticks([])
    ax[i].set_yticks([])
    ax[i].imshow(cv2.cvtColor(img,cv2.COLOR_BGR2RGB), cmap='gray', vmin=0, vmax=255)
plt.show()
```

Aquí podemos ver los **recortes obtenidos** correspondientes al código del contenedor.

Fig. 2.12. Dígitos detectados en la imagen del código del contenedor.

Para **recortar los de la fecha** ejecutamos lo siguiente.

```
lista_img_nums_fecha = []

for f in fecha:
  # Define the coordinates of the region to crop
  x1 = round(f[0])
  y1 = round(f[1])  # Top-left corner coordinates
  x2 = round(f[2])
  y2 = round(f[3]) # Bottom-right corner coordinates

  # Crop the image using array slicing
  cropped_image_fecha = imagen[y1:y2, x1:x2]
  resized_image_fecha = cv2.resize(cropped_image_fecha, (28, 28), interpolation=cv2.INTER_AREA)
  lista_img_nums_fecha.append(resized_image_fecha)

fig, ax = plt.subplots(figsize=(15,15),ncols=len(lista_img_nums_fecha),nrows=1)
for i, img in enumerate(lista_img_nums_fecha):

    ax[i].set_xticks([])
    ax[i].set_yticks([])
    ax[i].imshow(cv2.cvtColor(img,cv2.COLOR_BGR2RGB), cmap='gray', vmin=0, vmax=255)
plt.show()
```

Destacar que **las imágenes se han redimensionado** a 28 x 28 píxeles, que corresponde con el **tamaño de las imágenes con las que entrenaremos el modelo** de clasificación.

Fig. 2.13. Dígitos detectados de la fecha.

Al igual que hemos tenido que redimensionar el tamaño de las imágenes, es necesario realizar un **preprocesado de la imagen** de cada dígito, **para adecuarlo al dataset de entrenamiento**. Dicho dataset está compuesto por imágenes con fondo oscuro y el trazo del dígito en claro.

Para conseguir este efecto necesitamos primero **convertirlos a escala de grises, posteriormente realizar una inversión de color para acabar realizando una umbralización para delimitar con más definición el contorno del dígito.**

También entra dentro del preprocesado la **normalización, para que los valores correspondientes a cada pixel se encuentren entre 0 y 1**.

El código que realiza el preprocesado de los códigos de contenedor es este.

```
lista_codigos =[]
for num in lista_img_nums_codigo:
    img_aux = num
    # Conversión a escala de grises
    img_aux = cv2.cvtColor(img_aux, cv2.COLOR_BGR2GRAY)
    # Inversión de color
    img_aux = cv2.bitwise_not(img_aux)
    # Umbralización
    _,img_aux = cv2.threshold(img_aux,120,255,cv2.THRESH_BINARY)

    lista_codigos.append(img_aux)

# Normalización
x_predict_codigos = np.asarray(lista_codigos, dtype=np.uint8)
x_predict_codigos= x_predict_codigos / 255.

fig, ax = plt.subplots(figsize=(15,15),ncols=len(lista_codigos),nrows=1)
for i, img in enumerate(lista_codigos):

    ax[i].set_xticks([])
    ax[i].set_yticks([])
    ax[i].imshow(img, cmap='gray', vmin=0, vmax=255)
plt.show()
```

Fig. 2.14. Dígitos preprocesados del código del contenedor.

Y para el **preprocesado de las fechas**.

```
lista_fechas =[]
for num in lista_img_nums_fecha:
    img_aux = num
    # Conversion a escala de grises
    img_aux = cv2.cvtColor(img_aux, cv2.COLOR_BGR2GRAY)
    # Inversion de color
    img_aux = cv2.bitwise_not(img_aux)
    # Umbralizacion
    _,img_aux = cv2.threshold(img_aux,120,255,cv2.THRESH_BINARY)
    lista_fechas.append(img_aux)

#Normalizacion
x_predict_fechas = np.asarray(lista_fechas, dtype=np.uint8)
x_predict_fechas= x_predict_fechas / 255.

fig, ax = plt.subplots(figsize=(15,15),ncols=len(lista_fechas),nrows=1)
for i, img in enumerate(lista_fechas):

    ax[i].set_xticks([])
    ax[i].set_yticks([])
    ax[i].imshow(img, cmap='gray', vmin=0, vmax=255)
plt.show()
```

Fig. 2.15. Dígitos preprocesados de la fecha.

3.- Clasificación de objetos

Una vez que hemos segmentado la zona de la imagen donde se encuentran los dígitos, los hemos detectado y los hemos preprocesado para que se asemejen al dataset de entrenamiento al que someteremos al modelo de clasificación, es el momento de comenzar ya con la implementación de dicho modelo.

3.1.- Modelo de clasificación

3.1.1.- Dataset Mnist

El **conjunto de datos MNIST** es una base de datos de dígitos escritos a mano que se utiliza para entrenar sistemas de procesamiento de imágenes y modelos de aprendizaje automático.

Las características principales de este dataset son las siguientes:

- contiene **60.000 imágenes de entrenamiento y 10.000 imágenes de prueba** de dígitos escritos a mano

- incluye **imágenes en escala de grises de 28x28 píxeles.**

- las **imágenes se normalizan** para que adaptarlas al tamaño de 28x28 píxeles

- se suavizan introduciendo **niveles de escala de grises**

- se utiliza para **entrenamiento y pruebas en el campo del aprendizaje automático**, en especial en tareas de clasificación de imágenes.

Fig. 3.1. Ejemplo dataset Mnist.

3.1.2.- Implementación del modelo

Lo primero es cargar las dependencias de las librerías que utilizaremos para la implementación del modelo.

```
import tensorflow as tf
import tensorflow_datasets as tfds
from tensorflow import keras
import matplotlib.pyplot as plt
import cv2
```

Seguidamente cargamos el dataset Mnist y definimos las diferentes clases a los que pueden pertenecer los dígitos, será un array de 0 a 9, que corresponden con todos los números posibles de una cifra.

```
datos_train, datos_test = tf.keras.datasets.mnist.load_data()
imagenes_train, labels_train = datos_train
imagenes_test, labels_test = datos_test

clases = ['0', '1','2', '3', '4', '5', '6', '7', '8', '9']
```

Ahora pasaríamos a implementar el modelo de clasificación. Para dicha tarea utilizaremos una red en la que alternaremos capas convolucionales con capas de maxpooling, seguidas de capas dropout para evitar el overfitting. La arquitectura de la red expuesta es la siguiente:

- **1.- capa convolucional 2d** con 32 neuronas, kernel de tamaño 3x3 y función de activación 'relu
- **2.- capa maxpooling** con filtros de tamaño 2x2
- 3.- capa de dropout
- **4.- capa convolucional 2d** con 64 neuronas, kernel de tamaño 3x3 y función de activación 'relu'
- **5.- capa maxpooling** con filtros de tamaño 2x2
- **6.- capa de dropout**
- **7.- capa convolucional 2d** con 128 neuronas, kernel tamaño 3x3 y función de activación 'relu'

- **8.- capa de maxpooling** con filtros 2x2

- **9.- capa de dropout**

- **10.- capa flatten**

- **11.- capa Dense** de 128 neuronal y función de activación 'relu'

- **12.- capa de dropout**

- **13.- capa de salida Dense** con 10 neuronas, una por cada clase que debemos clasificar, y función de activación 'softmax' para obtener un valor entre 0 y 1 en cada neurona de salida correspondiente con la probabilidad de que la entrada corresponda con dicha clase de salida.

El código correspondiente a la implementación del modelo sería el siguiente:

```
model_conv=tf.keras.models.Sequential()
model_conv.add(
    tf.keras.layers.Conv2D(
      filters=32,
      kernel_size=(3,3),
      activation='relu',
      input_shape = input_shape
      )
)
model_conv.add(
    tf.keras.layers.MaxPooling2D(pool_size=(2,2))
    )
model_conv.add(
    tf.keras.layers.Dropout(rate=0.5)
    )
model_conv.add(
    tf.keras.layers.Conv2D(
      filters=64,
      kernel_size=(3,3),
      activation='relu',
      input_shape = input_shape
      )
)
model_conv.add(
    tf.keras.layers.MaxPooling2D(pool_size=(2,2))
    )
model_conv.add(
    tf.keras.layers.Dropout(rate=0.5)
    )
```

```
model_conv.add(
    tf.keras.layers.Conv2D(
      filters=128,
      kernel_size=(3,3),
      activation='relu',
      input_shape = input_shape
      )
)
model_conv.add(
    tf.keras.layers.MaxPooling2D(pool_size=(2,2))
    )
model_conv.add(
    tf.keras.layers.Dropout(rate=0.4)
    )
model_conv.add(tf.keras.layers.Flatten())
model_conv.add(tf.keras.layers.Dense(128, 'relu'))
model_conv.add(tf.keras.layers.Dropout(rate=0.5

                                       ))
model_conv.add(tf.keras.layers.Dense(10,'softmax'))
```

En la **compilación** del modelo usaremos la función de **optimización Adam**, **la función de pérdica 'sparse_categorical_entropy'** y para medir la precisión la métrica 'accuracy'.

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

3.1.3.- Entrenamiento

Ya tenemos el modelo definido y compilado, pasaremos ahora a entrenarlo.Utilizaremos los datasets *imagenes_train* y *labels_train* como entrenamiento e *imagenes_test* y *labels_test* como dataset de validación. El entrenamiento se realizará durante 300 epochs.

```
history = model_conv.fit(
    imagenes_train_reshape,
    labels_train,
    epochs=300,
    validation_data=(imagenes_test_reshape, labels_test)
)
```

Una vez entrenado el modelo debemos **comprobar cómo de preciso** es. Para ello estudiamos el accuracy que muestra la última etapa de entrenamiento.

```
Epoch 300/300
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9744 - loss: 0.0894 - val_accuracy: 0.9890 - val_loss: 0.0370
```

Para ver si el modelo ha entrenado bien y no presenta overfitting, debemos realizar una evaluación sobre el dataset de validación y ver que accuracy presenta.

```
model_conv.evaluate(imagenes_test_reshape, labels_test)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9864 - loss: 0.0466
[0.037029992789030075, 0.9890000224113464]
```

El **accuracy obtenido en entrenamiento y en evaluación son similares**, por lo que podemos decir que el modelo se ha entrenado bien y **no presenta overfitting.** Además la precisión está cercana al 99%, lo que quiere decir que 99 de cada 100 imágenes el modelo las clasifica correctamente.

Una vez entrenado el modelo, analizada su precisión y dándolo por bueno, debemos **guardarlo en un archivo .h5** para que pueda importarse desde cualquier aplicación en la que se necesite clasificar dígitos escritos a mano.

```
model_conv.save('./modelos/reconocedor_numeros_conv_v1.h5')
```

3.2.- Aplicación del modelo de clasificación

El modelo de clasificación ya lo tenemos operativo, para poder aplicarlo a cada dígito, primero debemos cargarlo desde su ubicación. En nuestro caso está almacenado en Google Drive.

```
path_modelo = '/content/drive/MyDrive/dataset_PFM/modelo_custom_yolov8/numeros_a_mano/reconocedor_numeros_conv_v1.h5'
modelo_conv = keras.models.load_model(path_modelo)
```

Una vez que ya lo tenemos disponible, estamos preparados para realizar la **predicción sobre la imagen de cada dígito detectado por el modelo de detección y**

procesado para su clasificación. Trabajaremos por separado la predicción del código del contenedor y la fecha.

```
prediccion_conv_codigos = modelo_conv.predict(x_predict_codigos)
prediccion_conv_fechas = modelo_conv.predict(x_predict_fechas)
```

El modelo nos da como **salida para cada dígito, un array con la probabilidad de pertenecer a cada clase.** A nosotros nos interesa la posición en el array del valor mayor, por tanto usaremos el método Argmax() de Numpy.

```
codigos_salida=[]
for cod in prediccion_conv_codigos:
  codigos_salida.append(np.argmax(cod))
print(codigos_salida)

fechas_salida=[]
for fec in prediccion_conv_fechas:
  fechas_salida.append(np.argmax(fec))
print(fechas_salida)
```

Mediante la librería Matplotlib podremos visualizar las predicciones. Este sería el código para visualizar el código del contenedor.

```
[52] fig, ax = plt.subplots(figsize=(15,15),ncols=len(x_predict_codigos),nrows=1)
     for i in range(len(x_predict_codigos)):

         ax[i].set_xticks([])
         ax[i].set_yticks([])
         ax[i].imshow(lista_codigos[i], cmap='gray', vmin=0, vmax=255)
         if (i==0):
             ax[i].set_title('Codigo contenedor')
         ax[i].set_xlabel(f'Pred. Num: {codigos_salida[i]} ')
```

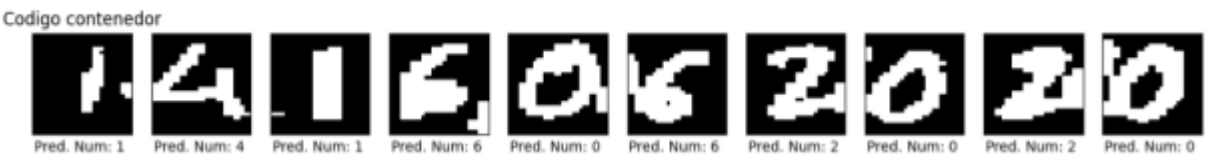


Fig. 3.2. Predicción de los dígitos del código del contenedor.

Y este sería el código para la fecha.

```
fig, ax = plt.subplots(figsize=(15,15),ncols=len(x_predict_fechas),nrows=1)
for i in range(len(x_predict_fechas)):

    ax[i].set_xticks([])
    ax[i].set_yticks([])
    ax[i].imshow(lista_fechas[i], cmap='gray', vmin=0, vmax=255)
    if (i==0):
        ax[i].set_title('Fecha')
    ax[i].set_xlabel(f'Pred. Num: {fechas_salida[i]} ')
```

Fig. 3.3. Predicción de los dígitos de la fecha.

Las **predicciones realizadas por el modelo son totalmente precisas**. La clasificación de todos los dígitos es óptima.

3.3.- Construcción del código y fecha

Una vez realizada la predicción por parte del modelo y como hemos podido comprobar es una muy buena clasificación, solo queda **construir el código del contenedor y la fecha de precintado con base a tal predicción**.

Con este código realizamos la construcción del código del contenedor y de la fecha.

```
print(f'Codigo Contenedor: {codigos_salida[0]}{codigos_salida[1]}{codigos_salida[2]}{codigos_salida[3]}/\
                           {codigos_salida[4]}{codigos_salida[5]}-{codigos_salida[6]}{codigos_salida[7]}\
                           {codigos_salida[8]}{codigos_salida[9]}/A')

year = str(fechas_salida[len(fechas_salida)-1])+str(fechas_salida[len(fechas_salida)-2])

if len(fechas_salida) == 5:
  month = fechas_salida[2]*10+fechas_salida[2]

  if int(month)>12:
    month = str(fechas_salida[2])
    day = str(fechas_salida[0])+str(fechas_salida[1])
elif len(fechas_salida) ==4:
  month = str(fechas_salida[1])
  day = str(fechas_salida[0])
print(f'Fecha de precintado: {day}-{month}-{year}')
```

```
Codigo Contenedor: 1416/06-20\20/A
Fecha de precintado: 26-9-22
```

Mostramos la imagen original de partida para verificar que la correspondencia con la predicción existe.

```
plt.imshow(image_rgb)
plt.title('Image')  # Add a title to the plot
plt.axis('off')  # Hide the axis ticks and labels
plt.show()
```

Fig. 3.4. Máscara de imagen original.

3.4.- Conclusión

Podemos concluir que tanto el código de contenedor y la fecha que hemos podido construir a partir de las predicciones realizadas por el modelo de segmentación, detección y clasificación, coinciden al 100% con lo aparece escrito en la imagen.

Posibles mejoras futuras del proyecto.

Como posibles mejoras del proyecto podríamos pensar en varias opciones.

Una es la posible utilización de redes transformer en la etapa de detección y ver realmente si aportan mayor precisión a la hora de la detección, para que en la etapa posterior sea un poco más precisa la clasificación.

También habría que tener en cuenta para mejoras futuras, sería el desarrollo de las funcionalidades necesarias para aplicar los modelos en imágenes donde aparezcan los precintos girados o con condiciones de luz no idóneas, cosas que no se han contemplado en el desarrollo de este PFM.

Otra posible mejora que podría plantearse es la aplicación de los modelos en tiempo real, aplicando las etapas de segmentación, detección y clasificación sobre video in situ. De esta manera se podría realizar el trabajo de inserción de la información de precintado de cada contenedor en tiempo real, y no sobre lotes de imágenes en un proceso a posteriori, pudiendo detectar así el usuario si la transcripción se realiza correctamente.

Bibliografía

Romero, C.A.]. (2023, 28 de mayo). Instalar Labelme y Anaconda en Windows 11 - Ejemplos Para Crear Etiquetas en un Folder de Imágenes [Video]. YouTube. https://www.youtube.com/watch?v=O1gXz9CL1kE]

Codificando Bits. . (2024, 8 de abril).Tutorial: SEGMENTACIÓN DE IMÁGENES SATELITALES con Redes Convolucionales (U-Net) [Video]. YouTube.https://www.youtube.com/watch?v=MSiVi9I8SFU

U-NET : todo lo que tienes que saber sobre la red neuronal de Computer Vision(2022). https://datascientest.com/es/u-net-lo-que-tienes-que-saber

¿Qué es la segmentación semántica?. https://www.ibm.com/es-es/topics/semantic-segmentation

¿Qué son las redes neuronales convolucionales?. https://www.ibm.com/es-es/topics/convolutional-neural-networks

COCO El formato del conjunto de datos https://docs.aws.amazon.com/es_es/rekognition/latest/customlabels-dg/md-coco-overview.html

Documentación CV2. Open Source Computer Vision. https://docs.opencv.org/

Documentación Tensor Flow. https://www.tensorflow.org/

Matplotlib 3.9.2 documentation https://matplotlib.org/stable/index.html

¿Qué es la detección de objetos?. https://www.ibm.com/es-es/topics/object-detection

Build Vision Models with Roboflow (2024) https://docs.roboflow.com/

¿Qué es el aprendizaje por transferencia? https://aws.amazon.com/es/what-is/transfer-learning/

Roboflow. (2020). tensorflow-object-detection-faster-rcnn [Repositorio de GitHub]. https://github.com/roboflow/tensorflow-object-detection-faster-rcnn

How to Train Detectron2 on Custom Object Detection Data. https://blog.roboflow.com/how-to-train-detectron2/

How to Train RT-DETR on a Custom Dataset with Transformers https://blog.roboflow.com/train-rt-detr-custom-dataset-transformers/

Roboflow. (2024) How to Train YOLOv8 Object Detection on a Custom Dataset [Repositorio de GitHub].
https://github.com/roboflow/notebooks/blob/main/notebooks/train-yolov8-object-detection-on-custom-dataset.ipynb

Ultralytics YOLO Docs. (2024)
https://docs.ultralytics.com/es

Clasificación de objetos. (2021)
https://www.ibm.com/docs/es/wsr-and-r/8.5.6?topic=services-classifying-objects

MNIST Dataset
https://www.kaggle.com/datasets/hojjatk/mnist-dataset

Ormaza, J. (2020, 17 Mayo) Clasificando dígitos escritos a mano con Python. Medium.
https://medium.com/@ormax563jj/aprende-machine-learning-con-python-clasificando-d%C3%ADgitos-2b144d5b61c1

Índice de imágenes

A continuación comentaremos las diferentes imágenes que han aparecido en la memoria del PFM.