ACTIVIDAD ACADÉMICAMENTE DIRIGIDA: DEEP LEARNING CON TENSORFLOW: SEGMENTACIÓN

INTELIGENCIA ARTIFICIAL APLICADA A ROBOTS

GRADO DE INGENIERÍA INFORMÁTICA

CURSO 2023/24

MANUEL RAMÍREZ BALLESTEROS

Índice:

1- Introducción al problema.

En este documento se desglosarán los conocimientos adquiridos sobre Deep Learning abordando un problema de segmentación de carreteras. Este es un problema multiclase con un dataset compuesto por 31 imágenes de carreteras, que serán las entradas del modelo, y sus 31 respectivas máscaras, que nos indicarán el tipo de imágenes de salida que deberá aprender a generar nuestro modelo.

Para ello, se ha empleado una arquitectura de red neuronal llamado U-Net, empleada comúnmente en tareas de segmentación y con capacidad para producir resultados precisos incluso cuando se dispone de un conjunto pequeño de datos. En nuestro problema, consideraremos que hay 5 colores de salida que indicarán los distintos elementos que aparecen en las imágenes procesadas por el modelo de acuerdo con las máscaras que encontramos en el dataset.

Además, se emplearán técnicas de Transfer Learning para aprovechar el conocimiento adquirido por un modelo previamente entrenado y así agilizar el proceso.

Como entorno de desarrollo se ha escogido Google Colab y se han empleado librerías de aprendizaje automático como PyTorch y Scikit-learn.

2- Preparación de los datos y desarrollo del modelo.

En primer lugar, se ha obtenido el dataset desde Kaggle y se han guardado tanto la carpeta de *images* como la de *masks* en Google Drive. El código también se ha obtenido desde un Notebook de Kaggle.

Ha sido necesario instalar un par de paquetes que nos permitan importar bibliotecas con modelos de segmentación previamente entrenados e información detallada de dichos modelos de PyTorch:

```
%%capture
!pip install segmentation-models-pytorch

!pip install torchinfo
```

A continuación, se han importado las diferentes librerías que serán necesarias para nuestro modelo y la visualización de los datos:

```
import pandas as pd
import numpy as np

# Visualizar datos:
import matplotlib.pyplot as plt
import cv2
from PIL import Image

# train test split
from sklearn.model_selection import train_test_split
# Torch
import torch
from torch.utils.data import Dataset, DataLoader
from torch import nn, optim
from torchinfo import summary
import segmentation_models_pytorch as smp
from torchvision import transforms

# os
import os

# Path
from pathlib import Path

# tqdm
from tqdm.auto import tqdm

# warnings
import warnings
warnings.filterwarnings("ignore")
```

También se ha montado Google Drive en el entorno de de Colab para acceder a las imágenes y máscaras del dataset en las rutas definidas, empleando *glob* para obtener una lista de las imágenes en formato *png:*

```
# Imágenes
from google.colab import drive
drive.mount('/content/drive')

IMAGE_PATH = Path('/content/drive/MyDrive/images')
IMAGE_PATH_LIST = list(IMAGE_PATH.glob("*.png"))
IMAGE_PATH_LIST = sorted(IMAGE_PATH_LIST)

print(f'Imágenes = {len(IMAGE_PATH_LIST)}')
```

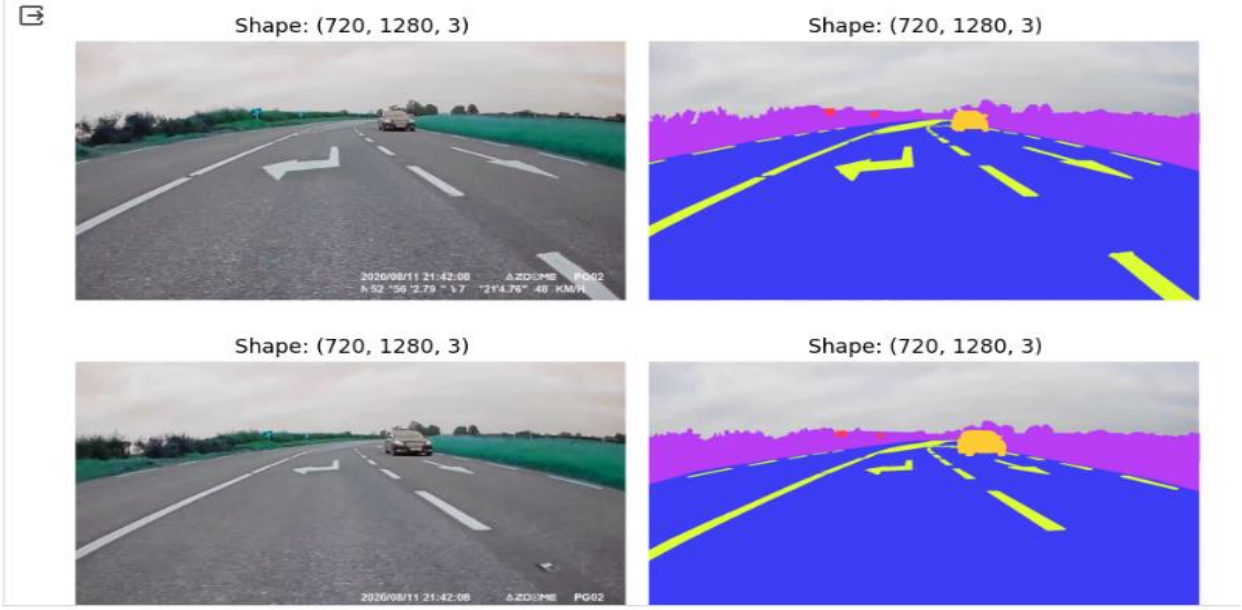
```
Drive already mounted at /content/drive; to attempt
Imágenes = 31
```

```
# Máscaras
MASK_PATH = Path('/content/drive/MyDrive/masks')
MASK_PATH_LIST = list(MASK_PATH.glob("*.png"))
MASK_PATH_LIST = sorted(MASK_PATH_LIST)

print(f'Máscaras = {len(MASK_PATH_LIST)}')
```

```
Máscaras = 31
```

A modo de ejemplo, se empleado *matplotlib* para visualizar algunas de las imágenes y máscaras del dataset iterando sobre los pares de rutas y mostrando las formas de las imágenes RGB:

```
fig,ax = plt.subplots(nrows = 5, ncols = 2, figsize = (9,15))

for i,(img_path, mk_path) in enumerate(zip(IMAGE_PATH_LIST, MASK_PATH_LIST)):
    if i>4:
        break

    img_bgr = cv2.imread(str(img_path))
    img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)
    ax[i,0].imshow(img_bgr)
    ax[i,0].axis('off')
    ax[i,0].set_title(f'Shape: {img_rgb.shape}')

    mask_bgr = cv2.imread(str(mk_path))
    mask_rgb = cv2.cvtColor(mask_bgr, cv2.COLOR_BGR2RGB)
    ax[i,1].imshow(mask_rgb)
    ax[i,1].axis('off')
    ax[i,1].set_title(f'Shape: {mask_rgb.shape}')

fig.tight_layout()
fig.show()
```



A continuación, se crea un *dataframe* de pandas que almacena las rutas de las imágenes y sus correspondientes máscaras y se muestra una vista previa de sus primeras filas:

```
images_paths = [None] * len(IMAGE_PATH_LIST)
masks_paths = [None] * len(MASK_PATH_LIST)

for i,(img_path, mask_path) in enumerate(zip(IMAGE_PATH_LIST, MASK_PATH_LIST)):
    images_paths[i] = img_path
    masks_paths[i] = mask_path
# Dataframe que almacena las imagenes y sus máscaras
data = pd.DataFrame({'Image':images_paths, 'Mask':masks_paths})
data.head()
```

	Image	Mask
0	/content/drive/MyDrive/images/0.png	/content/drive/MyDrive/masks/0.png
1	/content/drive/MyDrive/images/1.png	/content/drive/MyDrive/masks/1.png
2	/content/drive/MyDrive/images/10.png	/content/drive/MyDrive/masks/10.png
3	/content/drive/MyDrive/images/11.png	/content/drive/MyDrive/masks/11.png
4	/content/drive/MyDrive/images/12.png	/content/drive/MyDrive/masks/12.png

El dataset se ha dividido en tres conjuntos: Entrenamiento, test y validación. Para ello se emplea *train_test_split*, dejando el 70% de los datos para entrenamiento, el 15% para el conjunto de test y el otro 15% para la validación:

```
SEED = 42

data_train, data_rest = train_test_split(data,
                                         test_size = 0.3, # 70% para entrenamiento
                                         random_state = SEED)

data_val, data_test = train_test_split(data_rest,
                                       test_size = 0.5, # 15% para test y otro 15% para validación
                                       random_state = SEED)
```

Se ha generado un diccionario color-id para almacenar los valores de las componentes de los distintos colores de las máscaras y asignarlas a un valor que sirva como identificador:

```
# Diccionario color - id
color2id = {(184, 61, 245): 0, # #b83df5: backgroud
            (255, 53, 94):1, # #ff355e: road_sign
            (255, 204, 51):2, # #ffcc33: car
            (221, 255, 51):3, # #ddff33: marking
            (61,61, 245):4} # #3d3df5: road_surface
```

Se ha definido una función que toma como entrada una imagen y un diccionario de mapeo de valores de colores a ids para que recorra todos los píxeles de la imagen y verifique si el color del píxel se encuentra en el diccionario, con el fin de asignarle el id correspondiente al píxel en la matriz de salida:

```
# Array que ayudará a calcular las métricas
def mapping_color(img:Image, color2id:dict):

    image = np.array(img)

    height,width,_ = image.shape
    output_matrix = np.full(shape = (height, width), fill_value = -1, dtype = np.int32)

    for h in range(height):
        for w in range(width):
            color_pixel = tuple(image[h,w,:])

            if color_pixel in color2id:
                output_matrix[h,w] = color2id[color_pixel]


    return output_matrix
```

A continuación, se define una clase que hereda de Dataset de PyTorch para cargar un conjunto de datos personalizados (imágenes y máscaras) y que nos permita obtener una muestra específica:

```python
# Dataset
class CustomDataset(Dataset):
    def __init__(self, data:pd.DataFrame, color2id:dict, image_transforms, mask_transforms):
        self.data = data
        self.color2id = color2id
        self.image_transforms = image_transforms
        self.mask_transforms = mask_transforms

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):

        image_path = data.iloc[idx, 0]
        image = Image.open(image_path).convert("RGB")
        image = self.image_transforms(image)

        mask_path = data.iloc[idx, 1]
        mask = Image.open(mask_path).convert("RGB")
        mask = self.mask_transforms(mask)
        mask = mapping_color(mask, self.color2id)

        return image, mask
```

Se han empleado algunas transformaciones para procesar las imágenes antes de pasárselas al modelo como redimensionar las imágenes y convertirlas en tensores:

```python
RESIZE = (512, 512)

image_transforms = transforms.Compose([transforms.Resize(RESIZE),
                                       transforms.ToTensor()])

mask_transforms = transforms.Compose([transforms.Resize(RESIZE)])
```

Para la creación de los conjuntos de entrenamiento y validación personalizados se ha hecho lo siguiente:

```python
train_dataset = CustomDataset(data_train,
                              color2id,
                              image_transforms,
                              mask_transforms)

val_dataset = CustomDataset(data_val,
                            color2id,
                            image_transforms,
                            mask_transforms)
```

También se han creado los dataloaders de ambos conjuntos para cargar los conjuntos de datos personalizados en lotes que reciba directamente el modelo:

```
# DataLoader
BATCH_SIZE = 1
NUM_WORKERS = os.cpu_count()

train_dataloader = DataLoader(dataset = train_dataset,
                              batch_size = BATCH_SIZE,
                              shuffle = True,
                              num_workers = NUM_WORKERS)

val_dataloader = DataLoader(dataset = val_dataset,
                            batch_size = 1,
                            shuffle = True,
                            num_workers = NUM_WORKERS)
```

Para comprobar el redimensionamiento, se ha extraído un lote del conjunto de datos de entrenamiento que muestra una imagen con tres canales (RGB) y dimensiones 512x512 píxeles, al que la máscara:

```
batch_images, batch_masks = next(iter(train_dataloader))

batch_images.shape, batch_masks.shape
```

```
(torch.Size([1, 3, 512, 512]), torch.Size([1, 512, 512]))
```

A continuación, se especifica el tipo de dispositivo de procesamiento para ejecutar el modelo, dando prioridad a la GPU, aunque no ha sido mi caso:

```
# GPU
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
DEVICE
```

```
'cpu'
```

Se ha creado una instancia de la arquitectura U-Net con 5 clases de salida empleando la biblioteca *segmentation_models_pytorch*:

```
# Definición del Modelo
model = smp.Unet(classes = 5)
```

```
Downloading: "https://download.pytorch.org/models/resnet34-333f7ec4.pth"
100%|██████████| 83.3M/83.3M [00:00<00:00, 201MB/s]
```

Se ha empleado la función *summary* para mostrar la arquitectura del modelo:

```
# Arquitectura del modelo
summary(model = model,
        col_width = 17,
        input_size = [1,3,512,512],
        col_names = ['input_size','output_size','num_params','trainable'],
        row_settings = ['var_names'])
```

```
==========================================================================================================================
Layer (type (var_name))                                 Input Shape       Output Shape      Param #           Trainable
==========================================================================================================================
Unet (Unet)                                             [1, 3, 512, 512]  [1, 5, 512, 512]  --                True
├─ResNetEncoder (encoder)                               [1, 3, 512, 512]  [1, 3, 512, 512]  --                True
│    └─Conv2d (conv1)                                   [1, 3, 512, 512]  [1, 64, 256, 256] 9,408             True
│    └─BatchNorm2d (bn1)                                [1, 64, 256, 256] [1, 64, 256, 256] 128               True
│    └─ReLU (relu)                                      [1, 64, 256, 256] [1, 64, 256, 256] --                --
│    └─MaxPool2d (maxpool)                              [1, 64, 256, 256] [1, 64, 128, 128] --                --
│    └─Sequential (layer1)                              [1, 64, 128, 128] [1, 64, 128, 128] --                True
│    │    └─BasicBlock (0)                              [1, 64, 128, 128] [1, 64, 128, 128] 73,984            True
│    │    └─BasicBlock (1)                              [1, 64, 128, 128] [1, 64, 128, 128] 73,984            True
│    │    └─BasicBlock (2)                              [1, 64, 128, 128] [1, 64, 128, 128] 73,984            True
│    └─Sequential (layer2)                              [1, 64, 128, 128] [1, 128, 64, 64]  --                True
│    │    └─BasicBlock (0)                              [1, 64, 128, 128] [1, 128, 64, 64]  230,144           True
│    │    └─BasicBlock (1)                              [1, 128, 64, 64]  [1, 128, 64, 64]  295,424           True
│    │    └─BasicBlock (2)                              [1, 128, 64, 64]  [1, 128, 64, 64]  295,424           True
│    │    └─BasicBlock (3)                              [1, 128, 64, 64]  [1, 128, 64, 64]  295,424           True
│    └─Sequential (layer3)                              [1, 128, 64, 64]  [1, 256, 32, 32]  --                True
│    │    └─BasicBlock (0)                              [1, 128, 64, 64]  [1, 256, 32, 32]  919,040           True
│    │    └─BasicBlock (1)                              [1, 256, 32, 32]  [1, 256, 32, 32]  1,180,672         True
│    │    └─BasicBlock (2)                              [1, 256, 32, 32]  [1, 256, 32, 32]  1,180,672         True
│    │    └─BasicBlock (3)                              [1, 256, 32, 32]  [1, 256, 32, 32]  1,180,672         True
│    │    └─BasicBlock (4)                              [1, 256, 32, 32]  [1, 256, 32, 32]  1,180,672         True
│    │    └─BasicBlock (5)                              [1, 256, 32, 32]  [1, 256, 32, 32]  1,180,672         True
│    └─Sequential (layer4)                              [1, 256, 32, 32]  [1, 512, 16, 16]  --                True
│    │    └─BasicBlock (0)                              [1, 256, 32, 32]  [1, 512, 16, 16]  3,673,088         True
│    │    └─BasicBlock (1)                              [1, 512, 16, 16]  [1, 512, 16, 16]  4,720,640         True
│    │    └─BasicBlock (2)                              [1, 512, 16, 16]  [1, 512, 16, 16]  4,720,640         True
├─UnetDecoder (decoder)                                 [1, 3, 512, 512]  [1, 16, 512, 512] --                True
│    └─Identity (center)                                [1, 512, 16, 16]  [1, 512, 16, 16]  --                --
│    └─ModuleList (blocks)                              --                --                --                True
│    │    └─DecoderBlock (0)                            [1, 512, 16, 16]  [1, 256, 32, 32]  2,360,320         True
│    │    └─DecoderBlock (1)                            [1, 256, 32, 32]  [1, 128, 64, 64]  590,336           True
│    │    └─DecoderBlock (2)                            [1, 128, 64, 64]  [1, 64, 128, 128] 147,712           True
│    │    └─DecoderBlock (3)                            [1, 64, 128, 128] [1, 32, 256, 256] 46,208            True
│    │    └─DecoderBlock (4)                            [1, 32, 256, 256] [1, 16, 512, 512] 6,976             True
├─SegmentationHead (segmentation_head)                  [1, 16, 512, 512] [1, 5, 512, 512]  --                True
│    └─Conv2d (0)                                       [1, 16, 512, 512] [1, 5, 512, 512]  725               True
│    └─Identity (1)                                     [1, 5, 512, 512]  [1, 5, 512, 512]  --                --
│    └─Activation (2)                                   [1, 5, 512, 512]  [1, 5, 512, 512]  --                --
│    │    └─Identity (activation)                       [1, 5, 512, 512]  [1, 5, 512, 512]  --                --
==========================================================================================================================
Total params: 24,436,949
Trainable params: 24,436,949
Non-trainable params: 0
Total mult-adds (G): 31.41
==========================================================================================================================
Input size (MB): 3.15
Forward/backward pass size (MB): 583.01
Params size (MB): 97.75
Estimated Total Size (MB): 683.90
==========================================================================================================================
```

Luego se ha iterado sobre los parámetros del *encoder* del modelo para establecer que éstos no se actualizarán durante el entrenamiento y se ha generado un resumen del modelo para mostrar sus características:

```
# Capa encoder
for param in model.encoder.parameters():
    param.requires_grad = False

summary(model = model,
        col_width = 17,
        input_size = [1,3,512,512],
        col_names = ['input_size','output_size','num_params','trainable'],
        row_settings = ['var_names'])
```

```
==========================================================================================================================================
Layer (type (var_name))                                 Input Shape       Output Shape      Param #           Trainable
==========================================================================================================================================
Unet (Unet)                                             [1, 3, 512, 512]  [1, 5, 512, 512]  --                Partial
├─ResNetEncoder (encoder)                               [1, 3, 512, 512]  [1, 3, 512, 512]  --                False
│    └─Conv2d (conv1)                                   [1, 3, 512, 512]  [1, 64, 256, 256] (9,408)           False
│    └─BatchNorm2d (bn1)                                [1, 64, 256, 256] [1, 64, 256, 256] (128)             False
│    └─ReLU (relu)                                      [1, 64, 256, 256] [1, 64, 256, 256] --                --
│    └─MaxPool2d (maxpool)                              [1, 64, 256, 256] [1, 64, 128, 128] --                --
│    └─Sequential (layer1)                              [1, 64, 128, 128] [1, 64, 128, 128] --                False
│    │    └─BasicBlock (0)                              [1, 64, 128, 128] [1, 64, 128, 128] (73,984)          False
│    │    └─BasicBlock (1)                              [1, 64, 128, 128] [1, 64, 128, 128] (73,984)          False
│    │    └─BasicBlock (2)                              [1, 64, 128, 128] [1, 64, 128, 128] (73,984)          False
│    └─Sequential (layer2)                              [1, 64, 128, 128] [1, 128, 64, 64]  --                False
│    │    └─BasicBlock (0)                              [1, 64, 128, 128] [1, 128, 64, 64]  (230,144)         False
│    │    └─BasicBlock (1)                              [1, 128, 64, 64]  [1, 128, 64, 64]  (295,424)         False
│    │    └─BasicBlock (2)                              [1, 128, 64, 64]  [1, 128, 64, 64]  (295,424)         False
│    │    └─BasicBlock (3)                              [1, 128, 64, 64]  [1, 128, 64, 64]  (295,424)         False
│    └─Sequential (layer3)                              [1, 128, 64, 64]  [1, 256, 32, 32]  --                False
│    │    └─BasicBlock (0)                              [1, 128, 64, 64]  [1, 256, 32, 32]  (919,040)         False
│    │    └─BasicBlock (1)                              [1, 256, 32, 32]  [1, 256, 32, 32]  (1,180,672)       False
│    │    └─BasicBlock (2)                              [1, 256, 32, 32]  [1, 256, 32, 32]  (1,180,672)       False
│    │    └─BasicBlock (3)                              [1, 256, 32, 32]  [1, 256, 32, 32]  (1,180,672)       False
│    │    └─BasicBlock (4)                              [1, 256, 32, 32]  [1, 256, 32, 32]  (1,180,672)       False
│    │    └─BasicBlock (5)                              [1, 256, 32, 32]  [1, 256, 32, 32]  (1,180,672)       False
│    └─Sequential (layer4)                              [1, 256, 32, 32]  [1, 512, 16, 16]  --                False
│    │    └─BasicBlock (0)                              [1, 256, 32, 32]  [1, 512, 16, 16]  (3,673,088)       False
│    │    └─BasicBlock (1)                              [1, 512, 16, 16]  [1, 512, 16, 16]  (4,720,640)       False
│    │    └─BasicBlock (2)                              [1, 512, 16, 16]  [1, 512, 16, 16]  (4,720,640)       False
├─UnetDecoder (decoder)                                 [1, 3, 512, 512]  [1, 16, 512, 512] --                True
│    └─Identity (center)                                [1, 512, 16, 16]  [1, 512, 16, 16]  --                --
│    └─ModuleList (blocks)                              --                --                --                True
│    │    └─DecoderBlock (0)                            [1, 512, 16, 16]  [1, 256, 32, 32]  2,360,320         True
│    │    └─DecoderBlock (1)                            [1, 256, 32, 32]  [1, 128, 64, 64]  590,336           True
│    │    └─DecoderBlock (2)                            [1, 128, 64, 64]  [1, 64, 128, 128] 147,712           True
│    │    └─DecoderBlock (3)                            [1, 64, 128, 128] [1, 32, 256, 256] 46,208            True
│    │    └─DecoderBlock (4)                            [1, 32, 256, 256] [1, 16, 512, 512] 6,976             True
├─SegmentationHead (segmentation_head)                  [1, 16, 512, 512] [1, 5, 512, 512]  --                True
│    └─Conv2d (0)                                       [1, 16, 512, 512] [1, 5, 512, 512]  725               True
│    └─Identity (1)                                     [1, 5, 512, 512]  [1, 5, 512, 512]  --                --
│    └─Activation (2)                                   [1, 5, 512, 512]  [1, 5, 512, 512]  --                --
│    │    └─Identity (activation)                       [1, 5, 512, 512]  [1, 5, 512, 512]  --                --
==========================================================================================================================================
Total params: 24,436,949
Trainable params: 3,152,277
Non-trainable params: 21,284,672
Total mult-adds (G): 31.41
==========================================================================================================================================
Input size (MB): 3.15
Forward/backward pass size (MB): 583.01
Params size (MB): 97.75
Estimated Total Size (MB): 683.90
==========================================================================================================================================
```

Se establecen la función de pérdida y un optimizador Adam para el modelo:

```
loss_fn = smp.losses.DiceLoss(mode = "multiclass", classes = 5, ignore_index = -1)
optimizer = optim.Adam(model.parameters(), lr = 0.01, weight_decay = 0.0001)
```

También se crea una clase que permita detener el entrenamiento antes de tiempo si no se observa una mejora significativa durante cierto número de épocas:

```
class EarlyStopping:
    def __init__(self, patience:int = 5, delta:float = 0.0001, path = "best_model.pth"):
        self.patience = patience
        self.delta = delta
        self.path = path
        self.best_score = None
        self.counter = 0
        self.early_stop = False

    def __call__(self, val_loss, model):
        if self.best_score is None:
            self.best_score = val_loss
            self.save_checkpoint(model)

        elif val_loss > self.best_score + self.delta:
            self.counter += 1
            if self.counter >= self.patience:
                self.early_stop = True

        else:
            self.best_score = val_loss
            self.save_checkpoint(model)
            self.counter = 0

    def save_checkpoint(self, model):
        torch.save(model.state_dict(), self.path)
```

```
# Definir EarlyStopping
early_stopping = EarlyStopping(patience = 20, delta = 0.)
```

Se define una función que realice un paso de entrenamiento para el modelo que tome como parámetros el modelo de segmentación de imágenes, el dataloader con los lotes, la función de pérdida y el optimizador y devuelva la pérdida y el IOU (métrica de evaluación típica en problemas de segmentación) promedio durante el entrenamiento:

```
def train_step(model:torch.nn.Module, dataloader:torch.utils.data.DataLoader,
               loss_fn:smp.losses, optimizer:torch.optim.Optimizer):

    model.train()

    train_loss = 0.
    train_iou = 0.

    for batch,(X,y) in enumerate(dataloader):
        X = X.to(device = DEVICE, dtype = torch.float32)
        y = y.to(device = DEVICE, dtype = torch.long)

        optimizer.zero_grad()

        pred_logit = model(X)
        loss = loss_fn(pred_logit, y)
        train_loss = loss.item()

        loss.backward()
        optimizer.step()

        pred_prob = pred_logit.softmax(dim = 1)
        pred_class = pred_prob.argmax(dim = 1)

        tp,fp,fn,tn = smp.metrics.get_stats(output = pred_class.detach().cpu().long(),
                                            target = y.cpu(),
                                            mode = "multiclass",
                                            ignore_index = -1,
                                            num_classes = 5)

        train_iou += smp.metrics.iou_score(tp, fp, fn, tn, reduction = "micro")

    train_loss = train_loss / len(dataloader)
    train_iou = train_iou / len(dataloader)

    return train_loss, train_iou
```

De manera similar, se define una función para realizar una iteración en el conjunto de datos de validación:

```python
def val_step(model:torch.nn.Module, dataloader:torch.utils.data.DataLoader,
             loss_fn:smp.losses):

    model.eval()

    val_loss = 0.
    val_iou = 0.

    with torch.inference_mode():

        for batch,(X,y) in enumerate(dataloader):
            X = X.to(device = DEVICE, dtype = torch.float32)
            y = y.to(device = DEVICE, dtype = torch.long)

            pred_logit = model(X)
            loss = loss_fn(pred_logit, y)
            val_loss = loss.item()

            pred_prob = pred_logit.softmax(dim = 1)
            pred_class = pred_prob.argmax(dim = 1)

            tp,fp,fn,tn = smp.metrics.get_stats(output = pred_class.detach().cpu().long(),
                                                target = y.cpu(),
                                                mode = "multiclass",
                                                ignore_index = -1,
                                                num_classes = 5)

            val_iou += smp.metrics.iou_score(tp, fp, fn, tn, reduction = "micro")


    val_loss = val_loss / len(dataloader)
    val_iou = val_iou / len(dataloader)

    return val_loss, val_iou
```

También se define la función de entrenamiento del modelo durante un número específico de épocas. Esta función emplea un diccionario para almacenar las métricas de entrenamiento y validación, iterar sobre el rango de épocas para calcular la pérdida y el IOU e imprimirlas y comprobar el criterio de parada:

```python
def train(model:torch.nn.Module, train_dataloader:torch.utils.data.DataLoader,
          val_dataloader:torch.utils.data.DataLoader, loss_fn:torch.nn.Module, optimizer:torch.optim.Optimizer,
          early_stopping, epochs:int = 10):

    results = {'train_loss':[], 'train_iou':[], 'val_loss':[], 'val_iou':[]}

    for epoch in tqdm(range(epochs)):
        train_loss, train_iou = train_step(model = model,
                                           dataloader = train_dataloader,
                                           loss_fn = loss_fn,
                                           optimizer = optimizer)

        val_loss, val_iou = val_step(model = model,
                                     dataloader = val_dataloader,
                                     loss_fn = loss_fn)

        print(f'Epoch: {epoch + 1} | ',
              f'Train Loss: {train_loss:.4f} | ',
              f'Train IOU: {train_iou:.4f} | ',
              f'Val Loss: {val_loss:.4f} | ',
              f'Val IOU: {val_iou:.4f}')


        early_stopping(val_loss, model)

        if early_stopping.early_stop == True:
            print("Early Stopping!!")
            break

        results['train_loss'].append(train_loss)
        results['train_iou'].append(train_iou)
        results['val_loss'].append(val_loss)
        results['val_iou'].append(val_iou)

    return results
```

A continuación, se inicia el proceso de entrenamiento estableciendo inicialmente 100 épocas y almacenando los resultados. Como se puede apreciar, el entrenamiento finaliza al 57% (época 58), con una duración de 21 minutos:

```
# Training!!!
EPOCHS = 100

torch.cuda.manual_seed(SEED)
torch.manual_seed(SEED)

RESULTS = train(model.to(device = DEVICE),
                train_dataloader,
                val_dataloader,
                loss_fn,
                optimizer,
                early_stopping,
                EPOCHS)
```

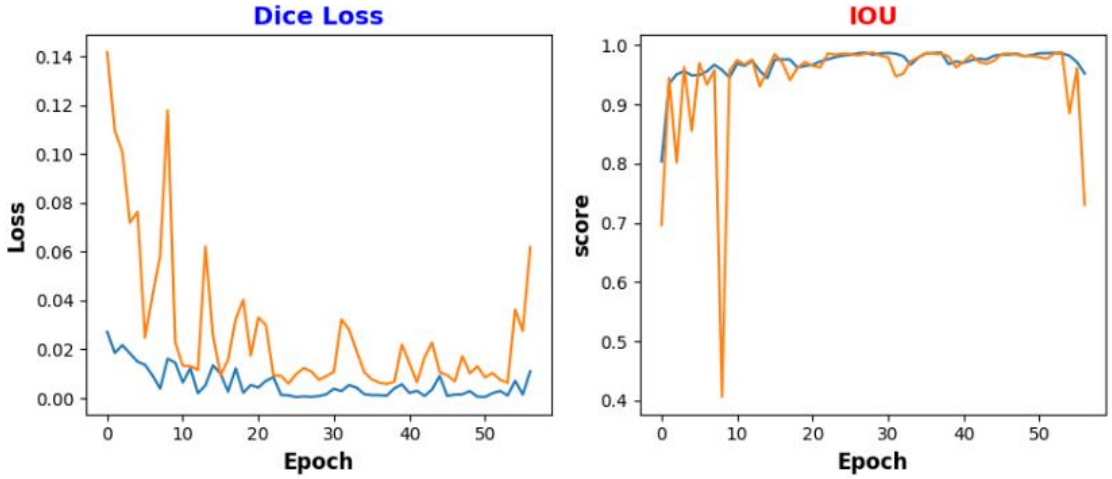
```
57%|█████▋    | 57/100 [21:22<16:26, 22.94s/it]
Epoch: 1 | Train Loss: 0.0272 | Train IOU: 0.8036 | Val Loss: 0.1418 | Val IOU: 0.6964
Epoch: 2 | Train Loss: 0.0186 | Train IOU: 0.9344 | Val Loss: 0.1097 | Val IOU: 0.9443
Epoch: 3 | Train Loss: 0.0217 | Train IOU: 0.9505 | Val Loss: 0.1009 | Val IOU: 0.8020
Epoch: 4 | Train Loss: 0.0184 | Train IOU: 0.9562 | Val Loss: 0.0719 | Val IOU: 0.9629
Epoch: 5 | Train Loss: 0.0150 | Train IOU: 0.9489 | Val Loss: 0.0764 | Val IOU: 0.8555
Epoch: 6 | Train Loss: 0.0137 | Train IOU: 0.9494 | Val Loss: 0.0248 | Val IOU: 0.9697
Epoch: 7 | Train Loss: 0.0093 | Train IOU: 0.9555 | Val Loss: 0.0422 | Val IOU: 0.9331
Epoch: 8 | Train Loss: 0.0039 | Train IOU: 0.9666 | Val Loss: 0.0585 | Val IOU: 0.9573
Epoch: 9 | Train Loss: 0.0162 | Train IOU: 0.9579 | Val Loss: 0.1180 | Val IOU: 0.4056
Epoch: 10 | Train Loss: 0.0145 | Train IOU: 0.9453 | Val Loss: 0.0228 | Val IOU: 0.9558
Epoch: 11 | Train Loss: 0.0064 | Train IOU: 0.9686 | Val Loss: 0.0133 | Val IOU: 0.9748
Epoch: 12 | Train Loss: 0.0123 | Train IOU: 0.9649 | Val Loss: 0.0132 | Val IOU: 0.9672
Epoch: 13 | Train Loss: 0.0020 | Train IOU: 0.9736 | Val Loss: 0.0116 | Val IOU: 0.9756
Epoch: 14 | Train Loss: 0.0053 | Train IOU: 0.9565 | Val Loss: 0.0621 | Val IOU: 0.9299
Epoch: 15 | Train Loss: 0.0134 | Train IOU: 0.9439 | Val Loss: 0.0256 | Val IOU: 0.9579
Epoch: 16 | Train Loss: 0.0100 | Train IOU: 0.9754 | Val Loss: 0.0095 | Val IOU: 0.9844
Epoch: 17 | Train Loss: 0.0027 | Train IOU: 0.9758 | Val Loss: 0.0158 | Val IOU: 0.9698
Epoch: 18 | Train Loss: 0.0122 | Train IOU: 0.9756 | Val Loss: 0.0324 | Val IOU: 0.9409
Epoch: 19 | Train Loss: 0.0022 | Train IOU: 0.9620 | Val Loss: 0.0403 | Val IOU: 0.9615
Epoch: 20 | Train Loss: 0.0054 | Train IOU: 0.9647 | Val Loss: 0.0176 | Val IOU: 0.9712
Epoch: 21 | Train Loss: 0.0044 | Train IOU: 0.9667 | Val Loss: 0.0331 | Val IOU: 0.9650
Epoch: 22 | Train Loss: 0.0071 | Train IOU: 0.9723 | Val Loss: 0.0300 | Val IOU: 0.9621
Epoch: 23 | Train Loss: 0.0086 | Train IOU: 0.9759 | Val Loss: 0.0093 | Val IOU: 0.9856
Epoch: 24 | Train Loss: 0.0014 | Train IOU: 0.9796 | Val Loss: 0.0092 | Val IOU: 0.9841
Epoch: 25 | Train Loss: 0.0012 | Train IOU: 0.9822 | Val Loss: 0.0060 | Val IOU: 0.9855
Epoch: 26 | Train Loss: 0.0005 | Train IOU: 0.9836 | Val Loss: 0.0099 | Val IOU: 0.9847
Epoch: 27 | Train Loss: 0.0007 | Train IOU: 0.9863 | Val Loss: 0.0123 | Val IOU: 0.9825
Epoch: 28 | Train Loss: 0.0006 | Train IOU: 0.9870 | Val Loss: 0.0110 | Val IOU: 0.9854
Epoch: 29 | Train Loss: 0.0009 | Train IOU: 0.9836 | Val Loss: 0.0076 | Val IOU: 0.9876
Epoch: 30 | Train Loss: 0.0016 | Train IOU: 0.9860 | Val Loss: 0.0090 | Val IOU: 0.9820
Epoch: 31 | Train Loss: 0.0039 | Train IOU: 0.9867 | Val Loss: 0.0109 | Val IOU: 0.9794
Epoch: 32 | Train Loss: 0.0029 | Train IOU: 0.9855 | Val Loss: 0.0323 | Val IOU: 0.9472
Epoch: 33 | Train Loss: 0.0054 | Train IOU: 0.9813 | Val Loss: 0.0283 | Val IOU: 0.9517
Epoch: 34 | Train Loss: 0.0043 | Train IOU: 0.9668 | Val Loss: 0.0193 | Val IOU: 0.9725
Epoch: 35 | Train Loss: 0.0017 | Train IOU: 0.9791 | Val Loss: 0.0108 | Val IOU: 0.9792
Epoch: 36 | Train Loss: 0.0013 | Train IOU: 0.9858 | Val Loss: 0.0078 | Val IOU: 0.9866
Epoch: 37 | Train Loss: 0.0013 | Train IOU: 0.9865 | Val Loss: 0.0064 | Val IOU: 0.9853
Epoch: 38 | Train Loss: 0.0010 | Train IOU: 0.9873 | Val Loss: 0.0058 | Val IOU: 0.9850
Epoch: 39 | Train Loss: 0.0040 | Train IOU: 0.9677 | Val Loss: 0.0067 | Val IOU: 0.9810
Epoch: 40 | Train Loss: 0.0057 | Train IOU: 0.9724 | Val Loss: 0.0220 | Val IOU: 0.9623
Epoch: 41 | Train Loss: 0.0022 | Train IOU: 0.9700 | Val Loss: 0.0144 | Val IOU: 0.9717
Epoch: 42 | Train Loss: 0.0030 | Train IOU: 0.9744 | Val Loss: 0.0065 | Val IOU: 0.9832
Epoch: 43 | Train Loss: 0.0009 | Train IOU: 0.9769 | Val Loss: 0.0167 | Val IOU: 0.9715
Epoch: 44 | Train Loss: 0.0036 | Train IOU: 0.9756 | Val Loss: 0.0228 | Val IOU: 0.9683
Epoch: 45 | Train Loss: 0.0091 | Train IOU: 0.9820 | Val Loss: 0.0107 | Val IOU: 0.9736
Epoch: 46 | Train Loss: 0.0010 | Train IOU: 0.9837 | Val Loss: 0.0094 | Val IOU: 0.9845
Epoch: 47 | Train Loss: 0.0015 | Train IOU: 0.9838 | Val Loss: 0.0069 | Val IOU: 0.9857
Epoch: 48 | Train Loss: 0.0016 | Train IOU: 0.9855 | Val Loss: 0.0172 | Val IOU: 0.9841
Epoch: 49 | Train Loss: 0.0029 | Train IOU: 0.9812 | Val Loss: 0.0102 | Val IOU: 0.9808
Epoch: 50 | Train Loss: 0.0006 | Train IOU: 0.9830 | Val Loss: 0.0131 | Val IOU: 0.9815
Epoch: 51 | Train Loss: 0.0005 | Train IOU: 0.9862 | Val Loss: 0.0085 | Val IOU: 0.9797
Epoch: 52 | Train Loss: 0.0021 | Train IOU: 0.9863 | Val Loss: 0.0103 | Val IOU: 0.9770
Epoch: 53 | Train Loss: 0.0030 | Train IOU: 0.9869 | Val Loss: 0.0076 | Val IOU: 0.9857
Epoch: 54 | Train Loss: 0.0011 | Train IOU: 0.9858 | Val Loss: 0.0063 | Val IOU: 0.9878
Epoch: 55 | Train Loss: 0.0071 | Train IOU: 0.9822 | Val Loss: 0.0363 | Val IOU: 0.8849
```

Para visualizar las curvas de pérdida y puntuación IOU durante el entrenamiento del modelo, se ha empleado la siguiente función que toma como entrada los resultados:

```python
# Define una función para ver la evolucion de la metrica y el loss durante el entrenamiento
def loss_and_metric_plot(results:dict):
    training_loss = results['train_loss']
    valid_loss = results['val_loss']

    training_iou = results['train_iou']
    valid_iou = results['val_iou']

    fig,axes = plt.subplots(nrows = 1, ncols = 2, figsize = (9,4))
    axes = axes.flat

    axes[0].plot(range(len(training_loss)), training_loss)
    axes[0].plot(range(len(valid_loss)), valid_loss)
    axes[0].set_xlabel("Epoch", fontsize = 12, fontweight = "bold", color = "black")
    axes[0].set_ylabel("Loss", fontsize = 12, fontweight = "bold", color = "black")
    axes[0].set_title("Dice Loss", fontsize = 14, fontweight = "bold", color = "blue")

    axes[1].plot(range(len(training_iou)), training_iou)
    axes[1].plot(range(len(valid_iou)), valid_iou)
    axes[1].set_xlabel("Epoch", fontsize = 12, fontweight = "bold", color = "black")
    axes[1].set_ylabel("score", fontsize = 12, fontweight = "bold", color = "black")
    axes[1].set_title("IOU", fontsize = 14, fontweight = "bold", color = "red")

    fig.tight_layout()
    fig.show()
```

Al invocar la función, vemos que el modelo parece caer en el sobreaprendizaje al comparar el conjunto de datos de entrenamiento y de evaluación cuando aumenta demasiado el número de épocas, razón por la cual se detiene antes de las 100 épocas.

```python
loss_and_metric_plot(RESULTS)
```



También se genera una función que toma el dataloader de test y la ruta del mejor modelo entrenado y lo utiliza para hacer predicciones sobre las imágenes de test, devolviendo el tensor de máscaras predichas.

```python
# Para la evaluación:

def predictions(test_dataloader:torch.utils.data.DataLoader, best_model:str):

    checkpoint = torch.load(best_model)

    loaded_model = smp.Unet(encoder_name = "resnet34", encoder_weights = None, classes = 5)

    loaded_model.load_state_dict(checkpoint)

    loaded_model.to(device = DEVICE)

    loaded_model.eval()

    pred_mask_test = []

    with torch.inference_mode():
        for X,_ in tqdm(test_dataloader):
            X = X.to(device = DEVICE, dtype = torch.float32)
            logit_mask = loaded_model(X)
            prob_mask = logit_mask.softmax(dim = 1)
            pred_mask = prob_mask.argmax(dim = 1)
            pred_mask_test.append(pred_mask.detach().cpu())

    pred_mask_test = torch.cat(pred_mask_test)

    return pred_mask_test
```

Se ha creado un conjunto de datos de test mediante la clase CustomDataset y un dataloader de test a partir del conjunto de datos anterior para cargar los datos en lotes:

```python
test_dataset = CustomDataset(data_test, color2id, image_transforms, mask_transforms)

test_dataloader = DataLoader(dataset = test_dataset,
                             batch_size = BATCH_SIZE,
                             shuffle = False)
```

También se han creado los tensores que contienen las imágenes de test y sus máscaras correspondientes para visualizar las imágenes después de realizar las predicciones:

```python
# Crear los tensores para plotear luego
IMAGE_TEST = []
MASK_TEST = []

for img,mask in test_dataloader:
    IMAGE_TEST.append(img)
    MASK_TEST.append(mask)

IMAGE_TEST = torch.cat(IMAGE_TEST)

MASK_TEST = torch.cat(MASK_TEST)
```

3- Resultados test.

Se han realizado las predicciones indicando la ruta para cargar el mejor modelo y se han calculado las métricas TP, FP, FN y TN correspondientes a la matriz de confusión del modelo que se utilizarán para evaluar el rendimiento del modelo en las imágenes de test:

```
pred_mask_test = predictions(test_dataloader, "/content/best_model.pth")

TP, FP, FN, TN = smp.metrics.get_stats(output = pred_mask_test.long(),
                                       target = MASK_TEST.long(),
                                       mode = "multiclass",
                                       ignore_index = -1,
                                       num_classes = 5)
```

```
100% 5/5 [00:10<00:00,  2.00s/it]
```

A continuación, se ha calculado el IOU para las predicciones del modelo en las imágenes de test utilizando las métricas anteriores y arrojando un resultado de 0.9850:

```
iou_test = smp.metrics.iou_score(TP, FP, FN, TN, reduction = "micro")
print(f'IOU Test = {iou_test:.4f}')
```

```
IOU Test = 0.9850
```

Para obtener los distintos colores en función de los identificadores que se definieron previamente, se ha generado un diccionario id-color:

```
#Diccionario para recuperar color segun el id:

id2color = {0: (184, 61, 245), # background
            1: (255, 53, 94), # road_sign
            2: (255, 204, 51), # car
            3: (221, 255, 51), # marking
            4: (61,61, 245)} # road_surface
```

Se han transformado las matrices que contenían los identificadores a sus máscaras predichas con sus respectivos colores y filtrando los valores que no se corresponden a ninguna clase:

```
# Transformar las matrices que contenian los ids a sus mascaras predichas con sus respectivos colores :
total_mask_output = []

for i,mask_pred in enumerate(pred_mask_test):

    # Extraer ancho y alto:
    height,width = mask_pred.shape

    # Array booleano array para filtrar los valores -1 que no pertenecen a ninguna clase
    mask_original = MASK_TEST[i]
    ignore = mask_original == -1 # id: -1

    mask_predicted = torch.where(ignore, -1, mask_pred)
    mask_zeros = torch.zeros(size = (height, width, 3), dtype = torch.uint8)

    for h in range(height):
        for w in range(width):
            idcolor = int(mask_predicted[h,w])

            if idcolor in id2color:
                mask_zeros[h,w,:] = torch.tensor(id2color[idcolor])

    total_mask_output.append(mask_zeros)
```

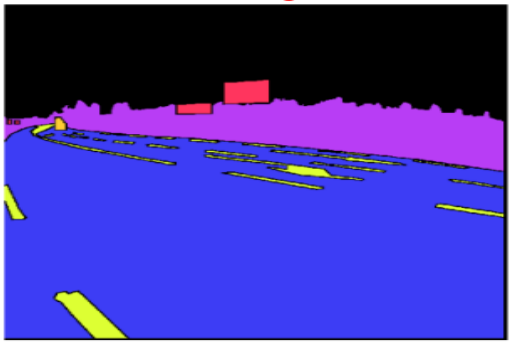
De manera similar, se generan las máscaras a partir de las reales de la base de datos de test:

```
# Igual pero con las máscaras reales
total_mask_test = []

for mask_tst in MASK_TEST:

    # Extraemos el height y width de la máscara.
    height,width = mask_tst.shape

    mask_zeros = torch.zeros(size = (height, width, 3), dtype = torch.uint8)

    for h in range(height):
        for w in range(width):
            idcolor = int(mask_tst[h,w])

            if idcolor in id2color:
                mask_zeros[h,w,:] = torch.tensor(id2color[idcolor])

    total_mask_test.append(mask_zeros)
```

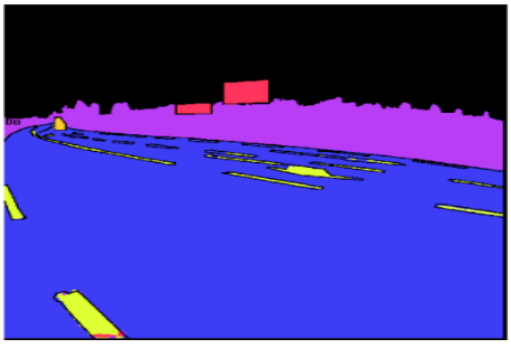
Por último, se han mostrado conjuntamente las máscaras predichas por el modelo y las reales:

```
# Visualizar las máscaras predichas y la máscara actual:
fig, ax = plt.subplots(nrows = 5, ncols = 2, figsize = (8,20))

for i,(mk_out,mk_test) in enumerate(zip(total_mask_output, total_mask_test)):

    mask_test = mk_test.numpy()
    ax[i,0].imshow(mask_test)
    ax[i,0].set_title("Máscara Original", fontsize = 12, fontweight = "bold", color = "red")
    ax[i,0].axis('off')

    mask_out = mk_out.numpy()
    ax[i,1].imshow(mask_out)
    ax[i,1].set_title("Máscara Predicha", fontsize = 12, fontweight = "bold", color = "green")
    ax[i,1].axis('off')

fig.tight_layout()
fig.show()
```

Mask Original

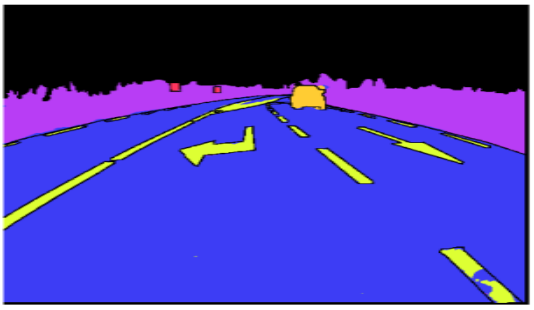


Mask Predicted



Mask Original

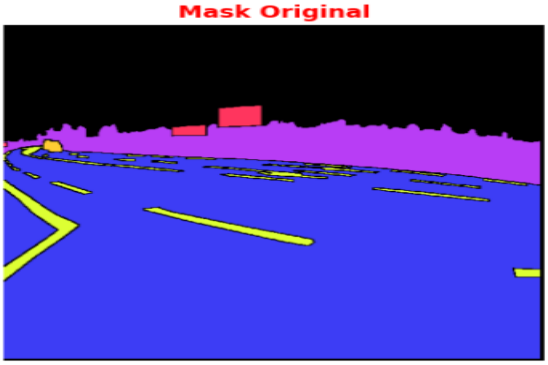
Mask Predicted



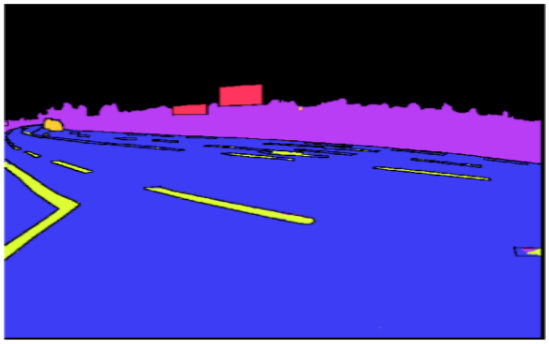
Mask Original

Mask Predicted

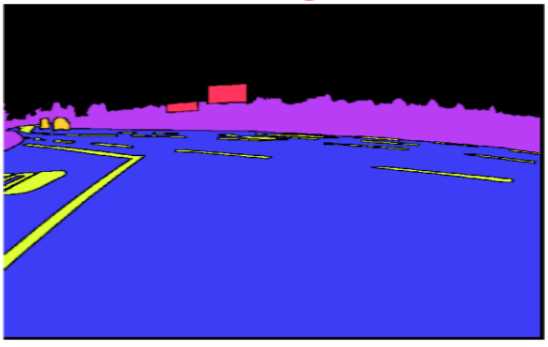
Mask Original



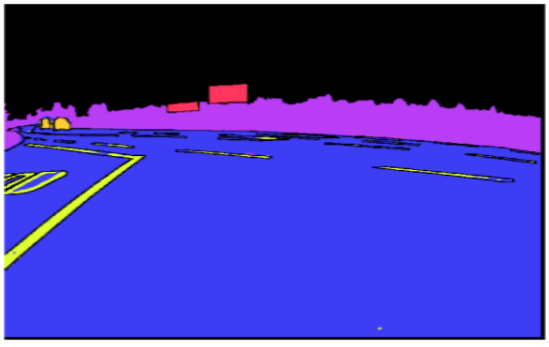
Mask Predicted



Mask Original



Mask Predicted



4- Conclusiones.

En este primer contacto práctico con el Deep Learning, hemos conseguido entrenar una red convolucional con arquitectura U-Net a partir de un modelo previamente entrenado para una tarea similar y que hemos adaptado a nuestro problema multiclase de segmentación de carreteras obteniendo unos resultados muy satisfactorios.

Como vemos, las máscaras predichas son muy similares a las originales procesadas, obviando algunos pequeños errores que casi no se aprecian a la hora de asignar las distintas clases (colores) a los elementos de las imágenes.

He de admitir que la primera red que planteé no la extraje de un Notebook de Kaggle ni pretendía usar técnicas de Transfer Learning, pero no supe abordar algunos problemas de dimensionalidad entre las capas que se me plantearon. Luego, traté de modificar una que encontré en un notebook que me resolvía el problema de la dimensionalidad, pero para un entrenamiento de 75 épocas, el modelo tardaba hasta 2 horas y los resultados que obtuve eran imágenes de prácticamente un solo color con algunas manchas de otros colores que no representaban adecuadamente los diferentes elementos de las imágenes originales.

Por último, me decanté por utilizar este modelo que he explicado, con el que no he tenido complicaciones de ningún tipo y que sí logra unos resultados realmente buenos.

5- Referencias.

- Apuntes de la asignatura IAAR.
- https://www.kaggle.com/datasets/trainingdatapro/roads-segmentation-dataset?select=roads_segmentation.csv
- https://www.kaggle.com/code/bryamblasrimac/segmentation-multiclass-unet-iou-98-30/notebook
- ChatGPT para comprender mejor algunas partes del código y aclarar algunos conceptos empleados.