Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ingeniería Eléctrica

Tarea 6

Segmentación semántica en dataset Kitti

Integrantes: Joaquín Zepeda
Profesor: Javier Ruiz del Solar
Auxiliar: Patricio Loncomilla
Santiago de Chile

Índice de Contenidos

Índice de Figuras

Índice de Tablas

Índice de Códigos

1. Introducción

La segmentación semánticas de imágenes no es algo simple en el ámbito del procesamiento de imágenes, de hecho hace pocos años se comenzó a utilizar en la industria con los autos tesla, los cuales realizan reconocimiento de objetos y segmentaciones semánticas para el piloto automático, reconocimiento de amenazas, etc. El objetivo de esta tarea es es entrenar y probar un sistema de segmentación semántica basado en U-net, esto con el fin de poner en práctica los conceptos vistos en clases, poner en práctica las habilidades de programación para resolver un problema real y poder apreciar en diferentes situaciones los resultados de estos métodos y los resultados modificando las redes U-net. Para esto se utilizará el dataset Kitti. Se espera que la red pueda diferenciar las distintas zonas de la imagen. Esta tarea se desarrolla utilizando el lenguaje de programación Python y el entorno de Google Colab. A continuación se describe el procedimiento de la tarea y se muestran y analizan los resultados en la sección de Desarrollo para luego finalizar con las conclusiones.

2. Desarrollo

Para el desarrollo de esta tarea se utilizaron funciones dadas en el enunciado y otras del repositorio de Github de Pytorch-UNet [1], estas fueron levemente modificadas y fueron reunidas en el Notebook de Google Colab para poder realizar implementar y probar la segmentación semantica.

1. Estructura dataset Kitti

Para el desarrollo de está tarea se utiliza el Kitti Dataset, el cual contiene 200 imágenes de entrenamiento etiquetadas píxel a píxel y 200 imágenes de test sin etiquetar. Las mascaras (etiquetas) de Kitti tienen 31 valores posibles los cuales se reducen a 11+1 clases posibles (11 clases de objetos y una clase extra para píxeles no válidos). El dataset contiene imágenes capturadas por un vehículo en movimiento. A continuación, se muestran ejemplos de las imágenes y máscaras (tanto en escala de grises como en RGB) que tiene este dataset:

(a) Original (b) Gray Mask (c) RGB Mask

Figura 1: Ejemplo de imágenes de entrenamiento.

En estas imágenes de ejemplo se puede observar como se segmentan de color azul los automóviles, de color verde los arboles, de color morado la calle, entre otros.

2. Implementar la clase KittiDataset.

La clase KittiDataset se implementó en base al código entregado *data_loading.py*, esta clase está encargada de generar el dataset, está clase recibe las direcciones tanto de las imágenes y las máscaras. Además de esto, la función *kitti_inverse_map*() [4] la cual reduce las 31 clases a 12. Por último, el método *__getitem__* retorna la imagen y label del índice entregado como parámetro.

Código 1: Extracto clase KittiDataset

```
class KittiDataset(Dataset):
    def __init__(self, imgs_dir, masks_dir, read_mask, scale=1, mask_suffix=''):

        self.imgs_dir = imgs_dir
        self.masks_dir = masks_dir
        self.read_mask = read_mask
        self.scale = scale
        self.mask_suffix = mask_suffix
                        ...
        self.ids = [splitext(file)[0] for file in listdir(imgs_dir)
                    if not file.startswith('.')]
        logging.info(f'Creating dataset with {len(self.ids)} examples')

```

```
    def __len__(self):
        return len(self.ids)
                        ...
    def __getitem__(self, i):
        name = self.ids[i]  #idx = self.ids[i]
        if self.read_mask:
            mask_file = list(self.masks_dir.glob(name + self.mask_suffix + '.*'))
        img_file = list(self.imgs_dir.glob(name + '.*'))
                        ...
        img = Image.open(img_file[0])
                        ...
        if self.read_mask == 'rgb': #lo mismo si es gray
          #print('Calling inverse map rgb...')
          mask = kitti_inverse_map(np.array(mask, dtype=np.int32))
          #print('Ok inverse map rgb...')
                        ...
        img = self.preprocess(img, self.scale)
                        ...
        return {
            'image': torch.from_numpy(img).type(torch.FloatTensor),'mask': mask_torch
        }
```

3. Modelo de la red

El modelo de la red consiste en 4 capas Down, luego 4 capas Up, estás se describen a continuación:

- Capa Down: consiste en bajar la dimensión de los datos con un Maxpooling de 2x2 para luego realizar una doble convolución. [1]
- Capa Up: consiste en aumentar la dimensión realizando un upsample con un factor de 2, para luego realizar una doble convolución. [1]

Estás se copiaron desde el archivo evaluate.py entregado en el material del enunciado.

Código 2: Modelo de la red UNet

```

from unet.unet_parts import *
class UNet(nn.Module):
    def __init__(self, n_channels, n_classes, bilinear=False):
        super(UNet, self).__init__()
        self.n_channels = n_channels
        self.n_classes = n_classes
        self.bilinear = bilinear

        self.inc = DoubleConv(n_channels, 64)
        self.down1 = Down(64, 128)
        self.down2 = Down(128, 256)
        self.down3 = Down(256, 512)
        factor = 2 if bilinear else 1
        self.down4 = Down(512, 1024 // factor)
```

```
        self.up1 = Up(1024, 512 // factor, bilinear)
        self.up2 = Up(512, 256 // factor, bilinear)
        self.up3 = Up(256, 128 // factor, bilinear)
        self.up4 = Up(128, 64, bilinear)
        self.outc = OutConv(64, n_classes)

    def forward(self, x):
        x1 = self.inc(x)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x5 = self.down4(x4)
        x = self.up1(x5, x4)
        x = self.up2(x, x3)
        x = self.up3(x, x2)
        x = self.up4(x, x1)
        logits = self.outc(x)
        return logits
```

Además, para el item 9 se modificó la red tal que quedaran 3 capas UP y 3 capas Down, para esto se eliminaron 2 capas. Para el item 10 se eliminaron 4 capas con el fin de que quedaran 2 capas UP y 2 capas Down. Esto se muestra en la siguiente tabla:

Tabla 1: Número de capas UP y Down de las redes utilizadas.

Configuraciones redes UNet		
4 capas Down	3 capas Down	2 capas Down
4 capas Up	3 capas Up	2 capas Up

5. Entrenar la red considerando 2 épocas

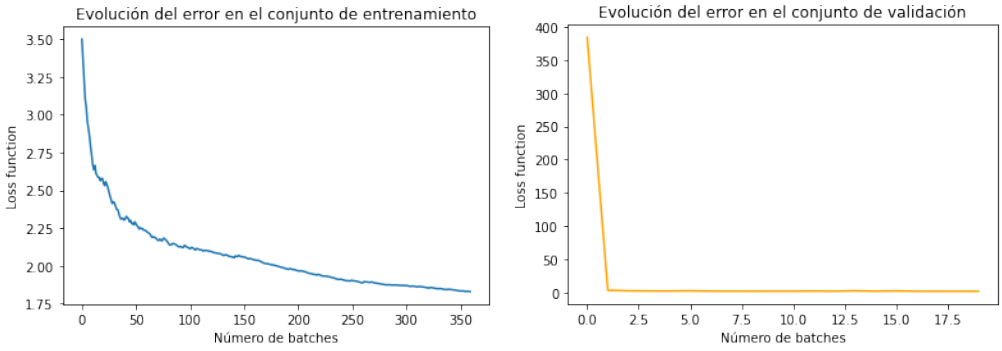
Para entrenar la red se copia el archivo train.py que se entrega junto con el enunciado al notebook. Se actualizan los directorios de las carpetas con las imágenes. La red, al entrenarse, recibe dos imágenes: una imagen RGB de tamaño 1x3xHxW, donde W y H es el ancho y alto de la imagen de entrada, y una máscara de tamaño 1xHxW conteniendo las etiquetas por cada píxel (al usar batch de tamaño 1). El objeto KittiDataset debe entregar imágenes de tamaño 3xHxW y el DataLoader se encarga de redimensionar la imagen a 1x3xHxW. Se usan 2 épocas para el entrenamiento, con un batch de tamaño 1. Se guardan los valores de la función de pérdida de entrenamiento y validación en cada entrenamiento.
La función train_net que se extrae del archivo train.py es la función encargada de realizar el entrenamiento de la red UNet, pero para el entrenamiento no se utilizan los parámetros por defecto de esta función, sino que se utilizan los siguientes parámetros generados por la función get_args_train():

- 2 épocas.
- Batch de tamaño 1.
- Tasa de aprendizaje de 0.0001

- Escalamiento de la imagen de 1.0.
- Porcentaje de validación del 10 %.
- Bilinear Upscaling.
- 12 clases.

6. Evolución de la función de pérdida de entrenamiento y validación

A continuación se pueden observar las curvas de loss tanto del conjunto de entrenamiento como del conjunto de validación utilizando la red UNet con 4 capas UP y 4 capas Down.



(a) (b)

Figura 2: Curvas de Loss para el conjunto de Entrenamiento y Prueba.

7. Análisis y resultados en los conjuntos de entrenamiento.

En la figura 6, se muestran la segmentación semántica en el conjunto de entrenamiento para 5 imágenes, se puede observar que las mascaras predichas por la red segmenta correctamente las calles pero tiene errores en los bordes de los vehículos y en algunos falsos reconocimientos de estos. A pesar de esto, en general el resultado es bueno, se logran detectar las calles, los vehículos, arboles y los contornos en la mayoría de las ocasiones, a pesar de esto, en la primera, tercera y ultima imagen se segmentan de forma incorrecta algunos sectores (se detecta de forma incorrecta autos en color amarillo).

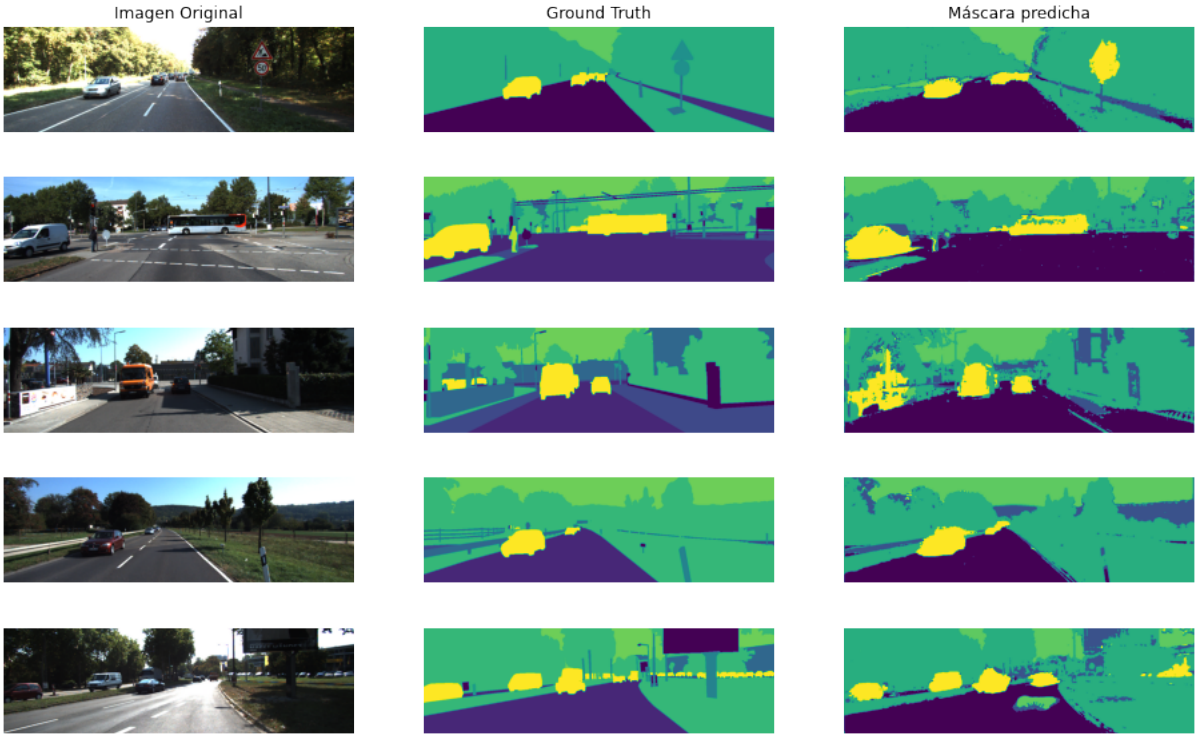


Figura 3: Imagen RGB, máscara real y máscara predicha para 5 imágenes del conjunto de entrenamiento.

Para estas visualizaciones se realizó una función auxiliar que transforma los tensores en imágenes para poder visualizarlas, está función se muestra continuación:

```
def transformar(img):
#depende del tipo de tensor la transformación que se le realiza
if img.shape[1]==3:
    return img.squeeze().permute(1, 2, 0).numpy()
elif img.shape[1]==12:
    img = np.transpose(img) #(1242, 375, 12, 1)
    img = img.squeeze() #(1242, 375, 12)
    img = np.argmax(img,axis=2).astype(np.ubyte)
    return img
else:
    img = img.permute(1, 2, 0).numpy().squeeze().astype(np.ubyte)
    return img
```

Utilizando esto como base, se modifica la función provista en predict.py para poder visualizar y generar las imágenes que se muestran en esta tarea.

8. Análisis y resultados en los conjuntos de Prueba.

En la figura 10, se muestran la segmentación semántica en el conjunto de prueba para 5 imágenes, se puede observar que las mascaras predichas por la red segmenta correctamente las calles, pero no tiene tan buen rendimiento con la segmentación de los autos, en las primeras 3 imágenes se identifican

los autos pero no se definen sus zonas de forma correcta (no se puede apreciar bien la forma), pero en las ultimas 2 imágenes de prueba se segmentan correctamente.

A pesar de esto, se logran detectar las calles, los autos, arboles y los contornos en la mayoría de las ocasiones, pero con bastantes errores en comparación con las imágenes de entrenamiento, incluso en algunas de las imágenes detecta señaléticas como autos.

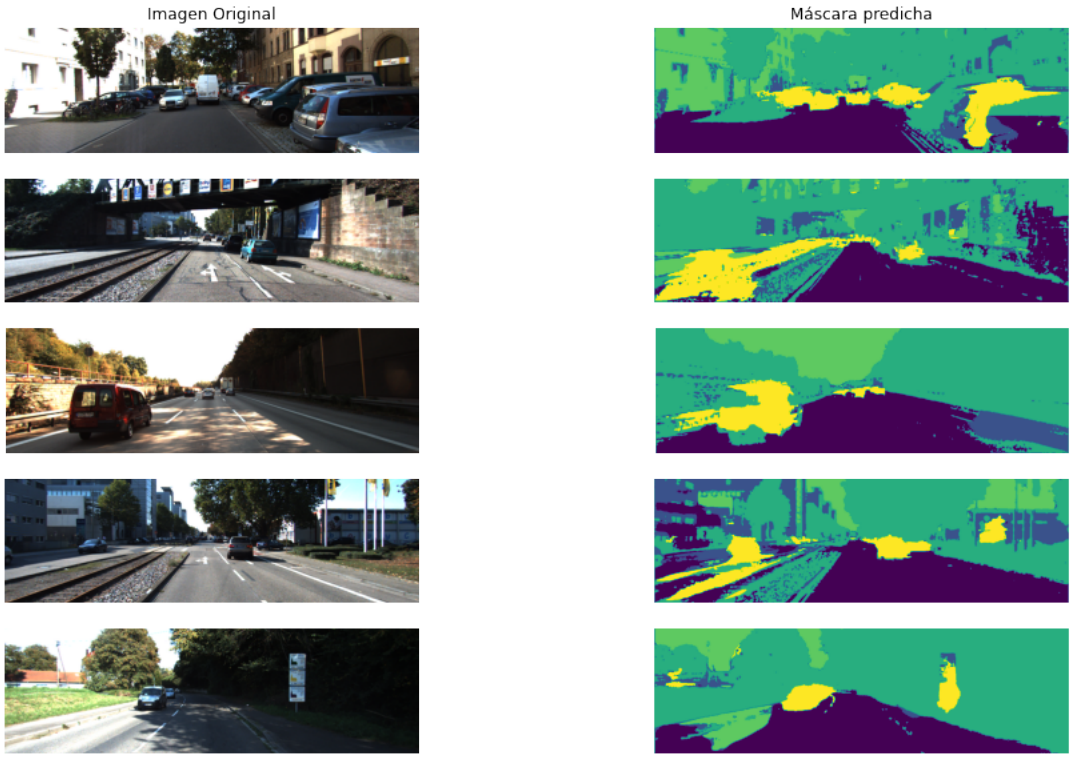
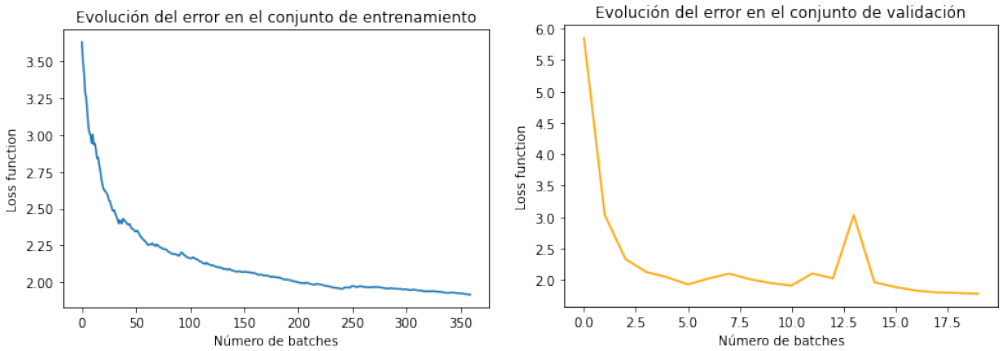


Figura 4: Imagen RGB y máscara predicha para 5 imágenes del conjunto de test.

9. Repetir los pasos 5 – 8, usando una red u-net con sólo 3 capas Up y 3 Down

A continuación, en la figura 5 se pueden observar las curvas de loss tanto del conjunto de entrenamiento como del conjunto de validación utilizando la red UNet con 3 capas UP y 3 capas Down. En la evolución del error en el conjunto de validación se puede observar que desciende de forma más errática teniendo un peak.



(a) (b)

Figura 5: Curvas de Loss para el conjunto de Entrenamiento y Prueba.

Al disminuir el número de capas, los resultados en las imágenes de entrenamiento cambian, por un lado en algunas de las imágenes como las primeras 2 (contando de arriba hacia abajo) una segmentación muy similar a la de la red anterior, pero en la imagen 3 empeora la segmentación reconociendo un árbol como un vehículo, además hay errores en la segmentación de las calles, por ejemplo en la imagen 4 (de arriba a abajo) se segmentan cerros como calles de forma errónea. En las ultimas dos se observa una buena segmentación, distinguiendo de forma correcta la calle (en la mayoría de las ocasiones) y los autos con un mejor contorno en la segmentación de los vehículos.

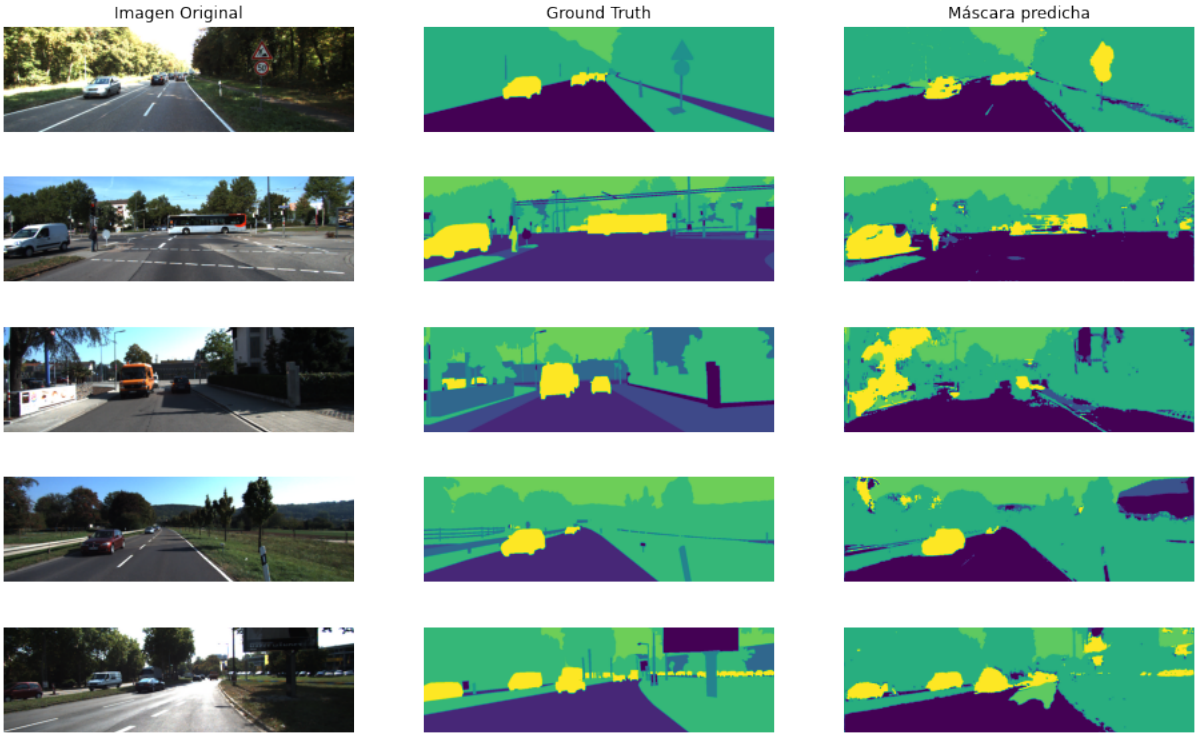


Figura 6: Imagen RGB, máscara real y máscara predicha para 5 imágenes del conjunto de entrenamiento.

Al disminuir el número de capas, los resultados en las imágenes de Prueba empeoran de forma drástica, sin ser posible distinguir las formas en algunas de estas (como por ejemplo la segunda y la cuarta de arriba a abajo), a pesar de esto en las que si segmento de forma correcta se ve una mejora con respecto a que existe un contorno de segmentación más suave.

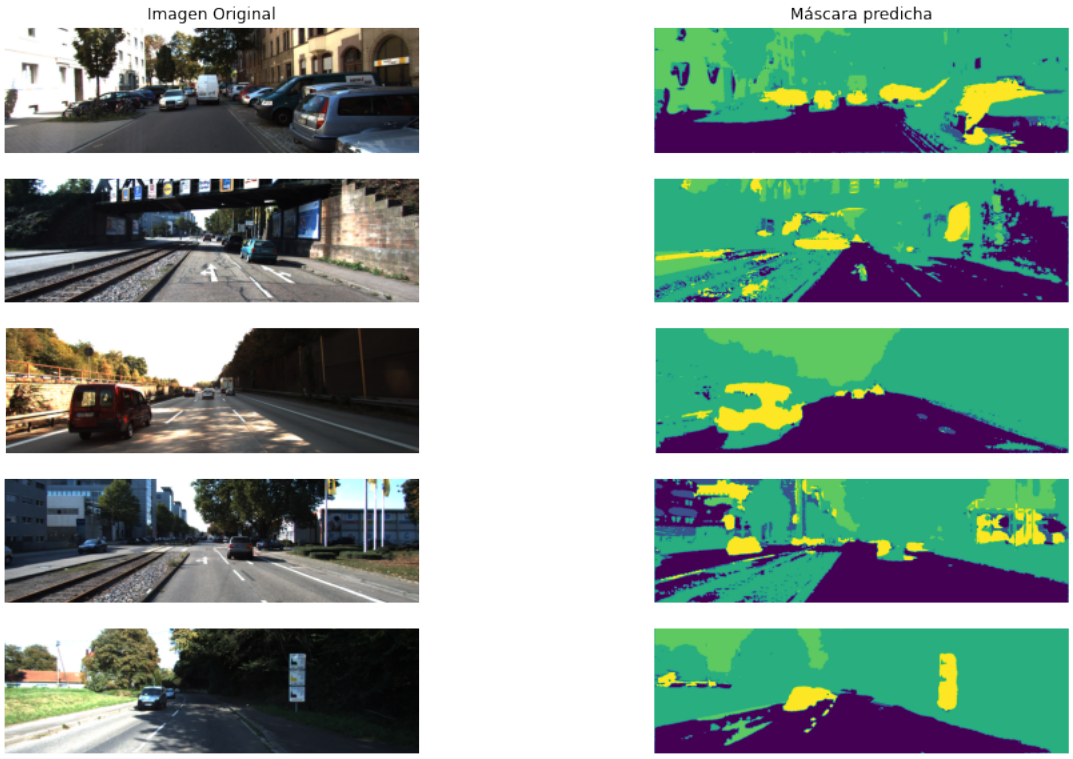
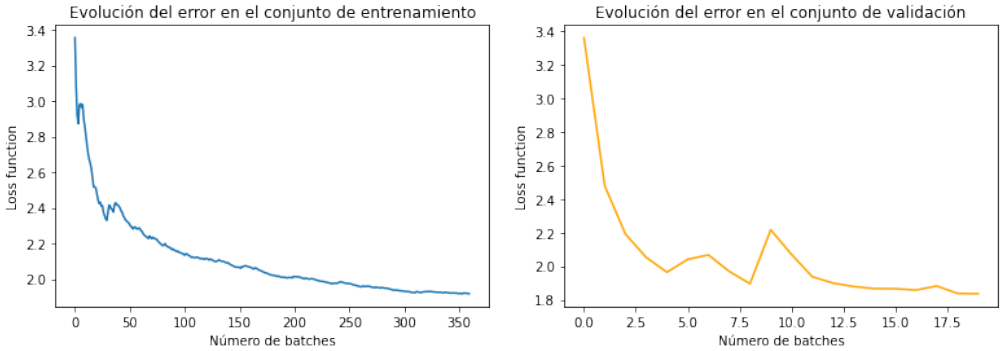


Figura 7: Imagen RGB y máscara predicha para 5 imágenes del conjunto de test.

10. Repetir los pasos 5 – 8, usando una red u-net con sólo 2 capas Up y 2 Down.

A continuación, en la figura 8 se pueden observar las curvas de loss tanto del conjunto de entrenamiento como del conjunto de validación utilizando la red UNet con 2 capas UP y 2 capas Down. En la evolución del error en el conjunto de entrenamiento y validación se puede observar que desciende de forma más errática.



(a) (b)

Figura 8: Curvas de Loss para el conjunto de Entrenamiento y Prueba.

Los resultados de la red en el conjunto de entrenamiento empeoraron, este segmenta parte de los vehículos no reconociéndolos de forma completa (como se espera), por otro lado hay errores en la segmentación de las calles, por ejemplo en la imagen 4 (de arriba a abajo) se segmentan cerros como calles de forma errónea. Nuevamente se detectan arboles como vehículos y no se detectan las veredas correspondientes.

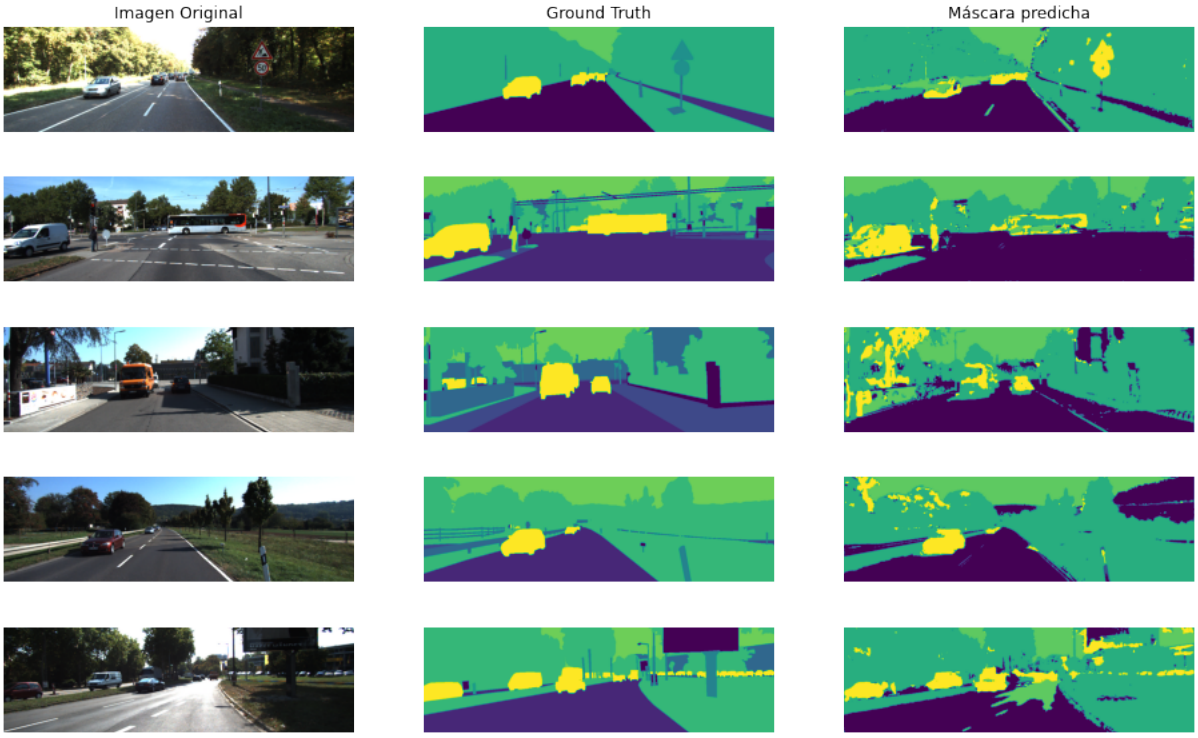


Figura 9: Imagen RGB, máscara real y máscara predicha para 5 imágenes del conjunto de entrenamiento.

Los resultados de esta red no son buenos, segmenta múltiples autos en zonas donde no hay, además los contornos de detección son poco prolijos, solo se destaca la detección de la imagen 3 en donde

se detecta bien un vehículo. Las calles tampoco se segmentan bien, detectando zonas de arboles en ellas.

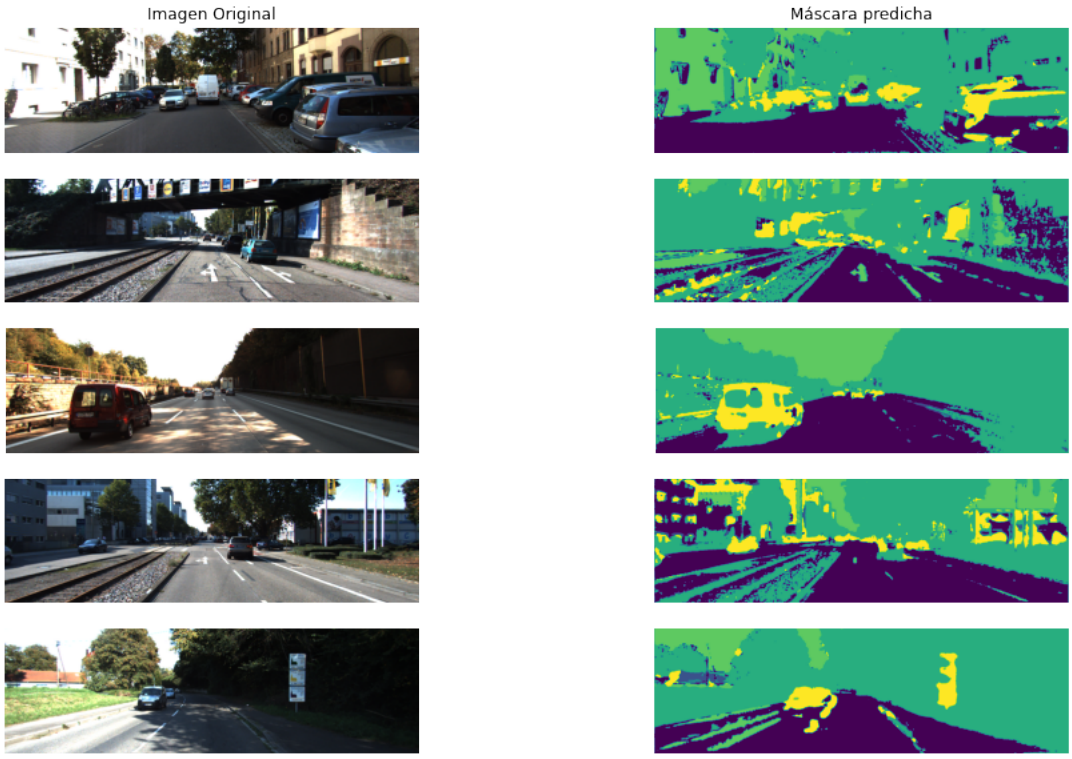


Figura 10: Imagen RGB y máscara predicha para 5 imágenes del conjunto de test.

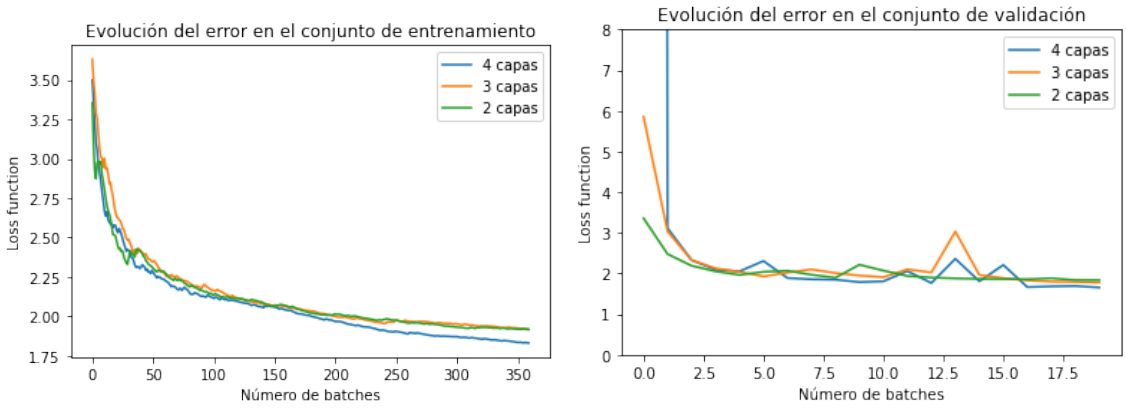
11. Comparar resultados

Luego de analizar los resultados de todos los modelos de forma individual en las secciones anteriores, los desempeños no son como se esperaba en las imágenes de training, pero a pesar de esto la red logra segmentar en la mayoría de los casos 3 zonas básicas: vehículo, calle y vegetación. La mejor de las 3 redes según los resultados tanto en las imágenes de entrenamiento como en las imágenes de prueba corresponde a la red inicial, la cual tiene 4 capas Down y 4 capas UP. Además, esta red es la que tiene un menor error en entrenamiento y validación (el cual es el más importante). A pesar de esto, los resultados en entrenamiento son mucho mejores que los resultados en prueba, lo que evidencia el gran sobreajuste de todos estos modelos. Los resultados en prueba no son los esperados, observando imágenes donde no se puede distinguir de forma clara a que corresponden.

12. Análisis del nivel de sobreajuste

Como se puede observar en la figura 11, la evolución de la función de error en los conjuntos de entrenamiento son similares, a medida que se eliminan capas UP y Down se llega a un error luego del entrenamiento, a pesar de esto, las redes presentan un overfitting pues si bien llegan a un error bajo en entrenamiento, en validación no ocurre lo mismo. De hecho al disminuir el número de capas el error en validación se vuelve errático, aumentando en ocasiones, a pesar de esto la red con 4 capas es la que llega a un menor error en validación, seguida por la red de 3 capas y por último la red de 2

capas. Cabe destacar también que el error inicial en validación de la primera red es más de 60 veces mayor que los errores iniciales de las otras redes, a pesar de esto este disminuye muy rápido, llegando a resultados similares a los de sus pares.



(a) (b)

Figura 11: Curvas de Loss para el conjunto de Entrenamiento y Prueba para cada red.

3. Conclusión

Luego del desarrollo de esta tarea, fue posible entrenar, probar y visualizar distintos sistemas de segmentación semántica usando redes U-net, cumpliendo el objetivo de esta, además las redes presentaron resultados que segmentaban bien en gran parte de los casos calles, vehículos y vegetación, estos resultados podrían mejorar modificando las redes U-net, quizás agregando capas extras, etc. En esta tarea fue posible utilizar un modelo ya creado para probar y entrenar de forma personal con un dataset definido usando la librería Pytorch. Estas herramientas son bastante útiles para nuestra formación como ingenieros pues se puede aplicar a un sin fin de proyectos.
Se recomienda el uso de la red U-net con 4 capas UP y 4 capas Down, pues fue el que presentó mejores resultados tanto visuales como en el conjunto de validación.
Está actividad permitió entender la segmentación semántica, utilizarlas y programarlas en Python. La mayor dificultad de esta tarea fue el de modificar las redes para disminuir el número de capas UP y Down, pues como estás realizan Maxpooling y Upsampling (disminuyendo y aumentando el tamaño de las imágenes respectivamente)las dimensiones no calzaban siempre, por lo que se tuvo que analizar a detalle para que estas calzaran.

Referencias

[1] Repositorio U-Net: Semantic segmentation with PyTorch. Disponible en: https://github.com/milesial/Pytorch-UNet

4. Anexos

```
# -*- coding: utf-8 -*-
"""Tarea6_imagenes.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1PkvdVqd3AHCg7n3lMWi88e0l7gwRoFfm

# Tarea 6 EL7008 - Segmentación semántica en dataset Kitti.

## Desarrollo por Joaquin Zepeda V.

El objetivo de esta tarea es entrenar y probar un sistema de segmentación semántica basado en U-net . El dataset a usar es Kitti, el cual contiene 200 imágenes de entrenamiento etiquetadas píxel a píxel y 200
imágenes de test sin etiquetar. Las etiquetas corresponden a 11+1 clases posibles (11 clases de objetos y
una clase extra para píxeles no válidos). El dataset contiene imágenes capturadas por un vehículo en movimiento.

# Reducir las etiquetas

Las máscaras de Kitti contienen 31 valores posibles, los cuales deben ser reducidos a 11+1 etiquetas dentro
de la clase KittiDataset. La función kitty_inverse_map_1channel() que realiza la conversión
será entregada junto con el enunciado de la tarea.
"""

!git clone https://github.com/milesial/Pytorch-UNet

!pip install wandb

"""# Descargamos el Dataset"""

!wget https://s3.eu-central-1.amazonaws.com/avg-kitti/data_semantics.zip

# Commented out IPython magic to ensure Python compatibility.
# %cd Pytorch-UNet/

!unzip ../data_semantics.zip

"""# Cargar los datos, clase KittiDataset"""

# Esta funcion debe ser copiada a una celda de colaboratory para poder usarla

from numba import jit
@jit(nopython=True)
def kitti_inverse_map_1channel(img):
```

```
  cmap = [
    (0, 0), #void (ignorable)
    (4, 0),
    (5, 0),
    (6, 0),
    (7, 1), #road
    (8, 2), #sidewalk
    (9, 2),
    (10, 0), #rail truck (ignorable)
    (11, 3), #construction
    (12, 3),
    (13, 3),
    (14, 3),
    (15, 3),
    (16, 3),
    (17, 4), #pole(s)
    (18, 4),
    (19, 5), #traffic sign
    (20, 5),
    (21, 6), #vegetation
    (22, 6),
    (23, 7), #sky
    (24, 8), #human
    (25, 8),
    (26, 9), #vehicle
    (27, 9),
    (28, 9),
    (29, 9),
    (30, 9),
    (31, 10), #train
    (32, 11), #cycle
    (33, 11)
  ]

  arrmap = np.zeros( (34), dtype=np.int32 )

  for el in cmap:
    arrmap[el[0]] = el[1]

  val = np.ones((img.shape[0],img.shape[1]), dtype=np.int32) * -1

  for i in range(img.shape[0]):
    for j in range(img.shape[1]):
      val[i,j] = arrmap[img[i,j]]
  return val

from os.path import splitext
from os import listdir
import numpy as np
from glob import glob
import torch
```

```
from torch.utils.data import Dataset
import logging
from PIL import Image

class KittiDataset(Dataset):
    def ___init___(self, imgs_dir, masks_dir, read_mask, scale=1, mask_suffix=''):

        self.imgs_dir = imgs_dir
        self.masks_dir = masks_dir
        self.read_mask = read_mask
        self.scale = scale
        self.mask_suffix = mask_suffix
        assert 0 < scale <= 1, 'Scale must be between 0 and 1'

        self.ids = [splitext(file)[0] for file in listdir(imgs_dir)
                    if not file.startswith('.')]
        logging.info(f'Creating dataset with {len(self.ids)} examples')

    def ___len___(self):
        return len(self.ids)

    @classmethod
    def preprocess(cls, pil_img, scale):
        w, h = pil_img.size
        newW, newH = int(scale * w), int(scale * h)
        assert newW > 0 and newH > 0, 'Scale is too small'
        pil_img = pil_img.resize((newW, newH))

        img_nd = np.array(pil_img)

        if len(img_nd.shape) == 2:
            img_nd = np.expand_dims(img_nd, axis=2)

        # HWC to CHW
        img_trans = img_nd.transpose((2, 0, 1))
        if img_trans.max() > 1:
            img_trans = img_trans / 255

        return img_trans

    @classmethod
    def do_transpose(cls, img_nd):
        img_trans = img_nd.transpose((2, 0, 1))
        return img_trans


    def ___getitem___(self, i):
        name = self.ids[i]
        idx = self.ids[i]
        if self.read_mask:
```

```
146         mask_file = list(self.masks_dir.glob(name + self.mask_suffix + '.*'))   #mask_file = glob(
    ↪ self.masks_dir + idx + self.mask_suffix + '.*')
147       img_file = list(self.imgs_dir.glob(name + '.*'))    #img_file = glob(self.imgs_dir + idx + '.*')
148
149       #print(mask_file[0])
150       #print(img_file[0])
151
152       if self.read_mask != None:
153         assert len(mask_file) == 1, \
154             f'Either no mask or multiple masks found for the ID {idx}: {mask_file}'
155         mask = Image.open(mask_file[0])
156
157       assert len(img_file) == 1, \
158           f'Either no image or multiple images found for the ID {idx}: {img_file}'
159
160       img = Image.open(img_file[0])
161
162       if self.read_mask != None:
163         assert img.size == mask.size, \
164           f'Image and mask {idx} should be the same size, but are {img.size} and {mask.size}'
165
166       if self.read_mask == 'rgb':
167         #print('Calling inverse map rgb...')
168         mask = kitti_inverse_map(np.array(mask, dtype=np.int32))
169         #print('Ok inverse map rgb...')
170       if self.read_mask == 'gray':
171         #print('Calling inverse map gray...')
172         mask = kitti_inverse_map_1channel(np.array(mask, dtype=np.int32))
173         #print('Ok inverse map gray...')
174
175       #print(np.array(mask))
176
177       img = self.preprocess(img, self.scale)
178
179       if self.read_mask != None:
180         #mask = mask[0:370, 0:1224]   #(370, 1224, 3)
181         mask_torch = torch.from_numpy(mask).type(torch.IntTensor)
182       else:
183         mask_torch = None
184
185       return {
186           'image': torch.from_numpy(img).type(torch.FloatTensor),
187           'mask': mask_torch
188       }
189
190 """#Modelo de la red
191
192 Se debe copiar el archivo unet_model.py al notebook. Se debe reemplazar el import por el siguiente:
193 """
194
195 """ Full assembly of the parts to form the complete network """
```

```
196
197 from unet.unet_parts import *
198
199 # red original para la primera parte
200 class UNet(nn.Module):
201     def ___init___(self, n_channels, n_classes, bilinear=False):
202         super(UNet, self).___init___()
203         self.n_channels = n_channels
204         self.n_classes = n_classes
205         self.bilinear = bilinear
206
207         self.inc = DoubleConv(n_channels, 64)
208         self.down1 = Down(64, 128)
209         self.down2 = Down(128, 256)
210         self.down3 = Down(256, 512)
211         factor = 2 if bilinear else 1
212         self.down4 = Down(512, 1024 // factor)
213         self.up1 = Up(1024, 512 // factor, bilinear)
214         self.up2 = Up(512, 256 // factor, bilinear)
215         self.up3 = Up(256, 128 // factor, bilinear)
216         self.up4 = Up(128, 64, bilinear)
217         self.outc = OutConv(64, n_classes)
218
219     def forward(self, x):
220         x1 = self.inc(x)
221         x2 = self.down1(x1)
222         x3 = self.down2(x2)
223         x4 = self.down3(x3)
224         x5 = self.down4(x4)
225         x = self.up1(x5, x4)
226         x = self.up2(x, x3)
227         x = self.up3(x, x2)
228         x = self.up4(x, x1)
229         logits = self.outc(x)
230         return logits
231
232
233 # Red utilizando solo 3 capas UP y 3 capas Down
234 class UNet2(nn.Module):
235     def ___init___(self, n_channels, n_classes, bilinear=False):
236         super(UNet2, self).___init___()
237         self.n_channels = n_channels
238         self.n_classes = n_classes
239         self.bilinear = bilinear
240
241         self.inc = DoubleConv(n_channels, 64)
242         self.down1 = Down(64, 128)
243         self.down2 = Down(128, 256)
244         factor = 2 if bilinear else 1
245
246         self.down3 = Down(256, 512// factor)
```

```
247
248         #self.down4 = Down(512, 1024 // factor) #eliminamos la ultima capa down
249
250         #self.up1 = Up(1024, 512 // factor, bilinear) #eliminamos la primera capa up
251         self.up2 = Up(512, 256 // factor, bilinear)
252         self.up3 = Up(256, 128 // factor, bilinear)
253         self.up4 = Up(128, 64, bilinear)
254
255         self.outc = OutConv(64, n_classes)
256
257     def forward(self, x):
258         x1 = self.inc(x)
259         x2 = self.down1(x1)
260         x3 = self.down2(x2)
261         x4 = self.down3(x3)
262         #x5 = self.down4(x4)
263         #x = self.up1(x5, x4)
264         x = self.up2(x4, x3)
265         x = self.up3(x, x2)
266         x = self.up4(x, x1)
267         logits = self.outc(x)
268         return logits
269
270
271 # Red utilizando solo 2 capas UP y 2 capas Down
272 class UNet3(nn.Module):
273     def ___init___(self, n_channels, n_classes, bilinear=False):
274         super(UNet3, self).___init___()
275         self.n_channels = n_channels
276         self.n_classes = n_classes
277         self.bilinear = bilinear
278
279         self.inc = DoubleConv(n_channels, 64)
280         self.down1 = Down(64, 128)
281         factor = 2 if bilinear else 1
282         self.down2 = Down(128, 256// factor)
283
284         #self.down3 = Down(256, 512)
285         #self.down4 = Down(512, 1024 // factor) #eliminamos la ultima capa down
286
287         #self.up1 = Up(1024, 512 // factor, bilinear) #eliminamos la primera capa up
288         #self.up2 = Up(512, 256 // factor, bilinear)
289         self.up3 = Up(256, 128 // factor, bilinear)
290         self.up4 = Up(128, 64, bilinear)
291
292         self.outc = OutConv(64, n_classes)
293
294     def forward(self, x):
295         x1 = self.inc(x)
296         x2 = self.down1(x1)
297         x3 = self.down2(x2)
```

```
        #x4 = self.down3(x3)
        #x5 = self.down4(x4)
        #x = self.up1(x5, x4)
        #x = self.up2(x4, x3)
        x = self.up3(x3, x2)
        x = self.up4(x, x1)
        logits = self.outc(x)
        return logits

"""#Funciones de pérdida
Se debe almacenar el valor de la función de pérdida, tanto de entrenamiento como de validación, cada vez
que se llame a la función evaluate(). Para lograr esto, se entregará un archivo modificado evaluate.py,
el cual se debe copiar a una celda del notebook. Además de esto, se debe implementar el cálculo del valor
promedio de la función de pérdida de entrenamiento promedio, sobre todos los batches de entrenamiento
procesados hasta el momento en cada época.
"""

!ls

#  evaluate.py
import torch
import torch.nn.functional as F
from tqdm import tqdm

from utils.dice_score import multiclass_dice_coeff, dice_coeff


def evaluate(net, dataloader, device, criterion):
    net.eval()
    num_val_batches = len(dataloader)
    dice_score = 0

    val_loss = 0

    # iterate over the validation set
    for batch in tqdm(dataloader, total=num_val_batches, desc='Validation round', unit='batch', leave=False):
        image, mask_true = batch['image'], batch['mask']
        # move images and labels to correct device and type
        image = image.to(device=device, dtype=torch.float32)
        mask_true = mask_true.to(device=device, dtype=torch.long)
        true_masks = mask_true   # PL 2022  agregado;  mask_pred en vez de masks_pred
        mask_true = F.one_hot(mask_true, net.n_classes).permute(0, 3, 1, 2).float()

        with torch.no_grad():
            # predict the mask
            mask_pred = net(image)

```

```
345             # PL 2022  agregado;  mask_pred en vez de masks_pred
346             loss = criterion(mask_pred, true_masks) \
347                    + dice_loss(F.softmax(mask_pred, dim=1).float(),
348                                F.one_hot(true_masks, net.n_classes).permute(0, 3, 1, 2).float(),
349                                multiclass=True)
350 
351             val_loss = val_loss + loss.item() # PL 2022
352 
353             # convert to one-hot format
354             if net.n_classes == 1:
355                 mask_pred = (F.sigmoid(mask_pred) > 0.5).float()
356                 # compute the Dice score
357                 dice_score += dice_coeff(mask_pred, mask_true, reduce_batch_first=False)
358             else:
359                 mask_pred = F.one_hot(mask_pred.argmax(dim=1), net.n_classes).permute(0, 3, 1, 2).
    ↪ float()
360                 # compute the Dice score, ignoring background
361                 dice_score += multiclass_dice_coeff(mask_pred[:, 1:, ...], mask_true[:, 1:, ...],
    ↪ reduce_batch_first=False)
362 
363 
364     net.train()
365 
366     # Fixes a potential division by zero error
367     if num_val_batches == 0:
368         return dice_score
369     return dice_score / num_val_batches, val_loss / num_val_batches
370 
371 """#Entrenamiento de la red
372 Se recomienda copiar el archivo train.py al notebook y modificarlo para poder usar el dataset Kitti,
373 considerando 12 clases posibles. El código en train.py ya contiene todo el código necesario para
        ↪ efectuar el
374 entrenamiento. Se deben reemplazar las líneas que crean el objeto dataset por la siguiente:
375 dataset = KittiDataset(dir_img, dir_mask, 'gray', img_scale)
376 Se debe comentar (deshabilitar) las siguientes dos líneas:
377 from evaluate import evaluate
378 from unet import UNet
379 Además se debe reemplazar la función get_args () por la función get_args_train() entregada en
380 el archivo get_args_train.py
381 La red, al entrenarse, recibe dos imágenes: una imagen RGB de tamaño 1x3xHxW, donde W y H es
        ↪ el
382 ancho y alto de la imagen de entrada, y una máscara de tamaño 1xHxW conteniendo las etiquetas
        ↪ por cada
383 píxel (al usar batch de tamaño 1). El objeto KittiDataset debe entregar imágenes de tamaño 3xHxW
        ↪ y
384 el DataLoader se encarga de redimensionar la imagen a 1x3xHxW. Se debe usar 2 épocas para el
385 entrenamiento, con un batch de tamaño 1. Se debe guardar los valores de la función de pérdida de
386 entrenamiento y validación para poder graficarlos posteriormente. El código base guarda un
        ↪ checkpoint
387 después de completar cada época.
388 """
```

```
389
390 import matplotlib.pyplot as plt7
391 import wandb
392 #!wandb login --relogin
393
394 import argparse
395 import logging
396 import sys
397 from pathlib import Path
398
399 import torch
400 import torch.nn as nn
401 import torch.nn.functional as F
402 import wandb
403 from torch import optim
404 from torch.utils.data import DataLoader, random_split
405 from tqdm import tqdm
406
407 from utils.dice_score import dice_loss
408 #from evaluate import evaluate
409 #from unet import UNet
410
411 dir_img = Path('./training/image_2/')
412 dir_mask = Path('./training/semantic/')
413 dir_checkpoint = Path('./checkpoints/')
414
415
416 def train_net(net,
417               device,
418               epochs: int = 5,
419               batch_size: int = 1,
420               learning_rate: float = 1e-5,
421               val_percent: float = 0.1,
422               save_checkpoint: bool = True,
423               img_scale: float = 0.5,
424               amp: bool = False):
425     # 1. Create dataset
426     try:
427         dataset = KittiDataset(dir_img, dir_mask, 'gray', img_scale)
428         print('Largo dataset:',len(dataset))
429     except (AssertionError, RuntimeError):
430         dataset = KittiDataset(dir_img, dir_mask, img_scale)
431
432     # 2. Split into train / validation partitions
433     n_val = int(len(dataset) * val_percent)
434     n_train = len(dataset) - n_val
435     train_set, val_set = random_split(dataset, [n_train, n_val], generator=torch.Generator().
      ↪ manual_seed(0))
436
437     # 3. Create data loaders
438     loader_args = dict(batch_size=batch_size, num_workers=1, pin_memory=True)
```

```
439     train_loader = DataLoader(train_set, shuffle=True, **loader_args)
440     val_loader = DataLoader(val_set, shuffle=False, drop_last=True, **loader_args)
441 
442     # (Initialize logging)
443     experiment = wandb.init(project='U-Net', resume='allow', anonymous='must')
444     experiment.config.update(dict(epochs=epochs, batch_size=batch_size, learning_rate=
        ↪ learning_rate,
445                                   val_percent=val_percent, save_checkpoint=save_checkpoint, img_scale=
        ↪ img_scale,
446                                   amp=amp))
447 
448     logging.info(f'''Starting training:
449         Epochs:          {epochs}
450         Batch size:      {batch_size}
451         Learning rate:   {learning_rate}
452         Training size:   {n_train}
453         Validation size: {n_val}
454         Checkpoints:     {save_checkpoint}
455         Device:          {device.type}
456         Images scaling:  {img_scale}
457         Mixed Precision: {amp}
458     ''')
459 
460     # 4. Set up the optimizer, the loss, the learning rate scheduler and the loss scaling for AMP
461     optimizer = optim.RMSprop(net.parameters(), lr=learning_rate, weight_decay=1e-8, momentum
        ↪ =0.9)
462     scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'max', patience=2)  # goal:
        ↪ maximize Dice score
463     grad_scaler = torch.cuda.amp.GradScaler(enabled=amp)
464     criterion = nn.CrossEntropyLoss()
465     global_step = 0
466 
467     #la idea es ir agregando el error total dividido en el número de batches recorridos
468     # hasta ese momento
469     t_loss = []
470     v_loss = []
471 
472     train_loss = 0
473 
474     # 5. Begin training
475     for epoch in range(1, epochs+1):
476         net.train()
477         epoch_loss = 0
478         with tqdm(total=n_train, desc=f'Epoch {epoch}/{epochs}', unit='img') as pbar:
479             for batch in train_loader:
480                 images = batch['image']
481                 true_masks = batch['mask']
482 
483                 assert images.shape[1] == net.n_channels, \
484                     f'Network has been defined with {net.n_channels} input channels, ' \
485                     f'but loaded images have {images.shape[1]} channels. Please check that ' \
```

```
            'the images are loaded correctly.'

        images = images.to(device=device, dtype=torch.float32)
        true_masks = true_masks.to(device=device, dtype=torch.long)

        with torch.cuda.amp.autocast(enabled=amp):
            masks_pred = net(images)
            loss = criterion(masks_pred, true_masks) \
                + dice_loss(F.softmax(masks_pred, dim=1).float(),
                            F.one_hot(true_masks, net.n_classes).permute(0, 3, 1, 2).float(),
                            multiclass=True)

        optimizer.zero_grad(set_to_none=True)
        grad_scaler.scale(loss).backward()
        grad_scaler.step(optimizer)
        grad_scaler.update()

        pbar.update(images.shape[0])
        global_step += 1
        epoch_loss += loss.item()
        train_loss += loss.item() #guarda el error global para todas las epocas
        #global step nos dice cuantos batches hemos recorrido
        t_loss.append(train_loss/global_step)
        experiment.log({
            'train loss': loss.item(),
            'step': global_step,
            'epoch': epoch
        })
        pbar.set_postfix(**{'loss (batch)': loss.item()})

        # Evaluation round
        division_step = (n_train // (10 * batch_size))
        if division_step > 0:
            if global_step % division_step == 0:
                histograms = {}
                for tag, value in net.named_parameters():
                    tag = tag.replace('/', '.')
                    if not torch.isinf(value).any():
                        histograms['Weights/' + tag] = wandb.Histogram(value.data.cpu())
                    if not torch.isinf(value.grad).any():
                        histograms['Gradients/' + tag] = wandb.Histogram(value.grad.data.cpu())

                val_score,val_loss = evaluate(net, val_loader, device,criterion)
                v_loss.append(val_loss)
                scheduler.step(val_score)

                logging.info('Validation Dice score: {}'.format(val_score))
                experiment.log({
                    'learning rate': optimizer.param_groups[0]['lr'],
                    'validation Dice': val_score,
                    'images': wandb.Image(images[0].cpu()),
```

```
                        'masks': {
                            'true': wandb.Image(true_masks[0].float().cpu()),
                            'pred': wandb.Image(masks_pred.argmax(dim=1)[0].float().cpu()),
                        },
                        'step': global_step,
                        'epoch': epoch,
                        **histograms
                    })

        if save_checkpoint:
            Path(dir_checkpoint).mkdir(parents=True, exist_ok=True)
            torch.save(net.state_dict(), str(dir_checkpoint / 'checkpoint_epoch{}.pth'.format(epoch)))
            logging.info(f'Checkpoint {epoch} saved!')

    return t_loss,v_loss


def get_args_train():
    #get_args_train()
    parser = argparse.ArgumentParser(description='Train the UNet on images and target masks',
                                     formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    parser.epochs = 2
    parser.batch_size = 1
    parser.lr = 0.0001
    parser.load = False
    parser.scale = 1.0
    parser.val = 10.0
    parser.amp = False
    parser.bilinear = True
    parser.classes = 12

    return parser

if __name__ == '__main__':
    args = get_args_train()

    logging.basicConfig(level=logging.INFO, format='%(levelname)s: %(message)s')
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    logging.info(f'Using device {device}')

    # Change here to adapt to your data
    # n_channels=3 for RGB images
    # n_classes is the number of probabilities you want to get per pixel
    net = UNet(n_channels=3, n_classes=args.classes, bilinear=args.bilinear)

    logging.info(f'Network:\n'
                 f'\t{net.n_channels} input channels\n'
                 f'\t{net.n_classes} output channels (classes)\n'
                 f'\t{"Bilinear" if net.bilinear else "Transposed conv"} upscaling')

    if args.load:
```

```
        net.load_state_dict(torch.load(args.load, map_location=device))
        logging.info(f'Model loaded from {args.load}')

    net.to(device=device)
    try:
        t_loss,v_loss = train_net(net=net,
                  epochs=args.epochs,
                  batch_size=args.batch_size,
                  learning_rate=args.lr,
                  device=device,
                  img_scale=args.scale,
                  val_percent=args.val / 100,
                  amp=args.amp)
    except KeyboardInterrupt:
        torch.save(net.state_dict(), 'INTERRUPTED.pth')
        logging.info('Saved interrupt')
        raise

import matplotlib.pyplot as plt

plt.figure()
plt.plot(t_loss)
plt.ylabel('Loss function')
plt.xlabel('Número de batches')
plt.title('Evolución del error en el conjunto de entrenamiento')

plt.figure()
plt.plot(v_loss,c='orange')
plt.ylabel('Loss function')
plt.xlabel('Número de batches')
plt.title('Evolución del error en el conjunto de validación')

"""# Función auxiliar para transformar las imagenes"""

def transformar(img):
    if img.shape[1]==3:
        return img.squeeze().permute(1, 2, 0).numpy()
    elif img.shape[1]==12:
        img = np.transpose(img) #(1242, 375, 12, 1)
        img = img.squeeze() #(1242, 375, 12)
        img = np.argmax(img,axis=2).astype(np.ubyte)
        return img
    else:
        img = img.permute(1, 2, 0).numpy().squeeze().astype(np.ubyte)
        return img

"""# Cargamos los checkpoints"""

import matplotlib.pyplot as plt

dir_img = Path('/content/Pytorch-UNet/training/image_2/')
```

```
dir_mask = Path('/content/Pytorch-UNet/training/semantic/')
checkpoint_file = '/content/Pytorch-UNet/checkpoints/checkpoint_epoch2.pth'

dataset_global = KittiDataset(dir_img, dir_mask, 'gray', scale = 1.0)

net = UNet(n_channels=3, n_classes=12, bilinear = True)

device = torch.device('cpu')
logging.info(f'Using device {device}')
net.to(device=device)
net.load_state_dict(torch.load(checkpoint_file, map_location=device))

net.cpu()

fig, ((ax1, ax2,ax3), (ax4, ax5,ax6), (ax7, ax8,ax9),(ax10, ax11,ax12),(ax13, ax14,ax15)) = plt.
    ↪ subplots(5,3, figsize=(16,10))

axes = [(ax1, ax2,ax3), (ax4, ax5,ax6), (ax7, ax8,ax9),(ax10, ax11,ax12),(ax13, ax14,ax15)]
n=10
j=0
for i in range(n,n+5):

  axa = axes[j][0]
  axb = axes[j][1]
  axc = axes[j][2]
  mysize = dataset_global[i]['image'].size()

  if j==0:
    axa.set_title('Imagen Original')
    axb.set_title('Ground Truth')
    axc.set_title('Máscara predicha')

  img_global = dataset_global[i]['image'].reshape((1,mysize[0],mysize[1],mysize[2])) #.cuda()
  img_gmask = dataset_global[i]['mask'].reshape((1,mysize[1],mysize[2])) #.cuda()

  #print("Tamaño del tensor de entrada:", img_global.size())
  img_pred = net.forward(img_global)
  img_np = img_pred.cpu().detach().numpy()
  #print("Tamaño del tensor de salida:", img_pred.size())

  # Por hacer: dibujar img_global, img_gmask y img_np usando plt.imshow()
  # Para esto, es necesario modificar las dimensiones de las imágenes, y transformar
  # las imágenes que contienen labels a formato np.ubyte
  # Además img_global debe ser multiplicada por 255
  axa.imshow(transformar(img_global))
  axb.imshow(transformar(img_gmask))
  axc.imshow(np.transpose(transformar(img_np))) # (W,H)
  axa.axis("off")
  axb.axis("off")
  axc.axis("off")
  j+=1
```

```
689
690 import matplotlib.pyplot as plt
691
692 dir_img = Path('/content/Pytorch-UNet/testing/image_2/')
693 dir_mask = Path('/content/Pytorch-UNet/training/semantic/')
694 checkpoint_file = '/content/Pytorch-UNet/checkpoints/checkpoint_epoch2.pth'
695
696 dataset_global = KittiDataset(dir_img, dir_mask, 'gray', scale = 1.0)
697
698 net = UNet(n_channels=3, n_classes=12, bilinear = True)
699
700 device = torch.device('cpu')
701 logging.info(f'Using device {device}')
702 net.to(device=device)
703 net.load_state_dict(torch.load(checkpoint_file, map_location=device))
704
705 net.cpu()
706
707 fig, ((ax1, ax2), (ax3, ax4), (ax5, ax6),(ax7, ax8),(ax9, ax10)) = plt.subplots(5,2, figsize=(16,10))
708
709 axes = [(ax1, ax2), (ax3, ax4), (ax5, ax6),(ax7, ax8),(ax9, ax10)]
710 n=15
711 j=0
712 for i in range(n,n+5):
713
714
715   axa = axes[j][0]
716   axb = axes[j][1]
717
718   if j==0:
719     axa.set_title('Imagen Original')
720     #axb.set_title('Ground Truth')
721     axb.set_title('Máscara predicha')
722
723   mysize = dataset_global[i]['image'].size()
724
725   img_global = dataset_global[i]['image'].reshape((1,mysize[0],mysize[1],mysize[2])) #.cuda()
726   #img_gmask = dataset_global[i]['mask'].reshape((1,mysize[1],mysize[2])) #.cuda()
727
728   #print("Tamaño del tensor de entrada:", img_global.size())
729   img_pred = net.forward(img_global)
730   img_np = img_pred.cpu().detach().numpy()
731   #print("Tamaño del tensor de salida:", img_pred.size())
732
733   # Por hacer: dibujar img_global, img_gmask y img_np usando plt.imshow()
734   # Para esto, es necesario modificar las dimensiones de las imágenes, y transformar
735   # las imágenes que contienen labels a formato np.ubyte
736   # Además img_global debe ser multiplicada por 255
737   axa.imshow(transformar(img_global))
738   axb.imshow(np.transpose(transformar(img_np))) # (W,H)
739   axa.axis("off")
```

```
 axb.axis("off")
 j+=1

"""# item 9

Usando red u-net con sólo 3 capas Up y 3 capas Down
"""

import argparse
import logging
import sys
from pathlib import Path

import torch
import torch.nn as nn
import torch.nn.functional as F
import wandb
from torch import optim
from torch.utils.data import DataLoader, random_split
from tqdm import tqdm

from utils.dice_score import dice_loss
dir_img = Path('./training/image_2/')
dir_mask = Path('./training/semantic/')
dir_checkpoint = Path('./checkpoints/')


if __name__ == '__main__':
    args = get_args_train()

    logging.basicConfig(level=logging.INFO, format='%(levelname)s: %(message)s')
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    logging.info(f'Using device {device}')

    # Change here to adapt to your data
    # n_channels=3 for RGB images
    # n_classes is the number of probabilities you want to get per pixel
    net2 = UNet2(n_channels=3, n_classes=args.classes, bilinear=args.bilinear)

    logging.info(f'Network:\n'
                 f'\t{net2.n_channels} input channels\n'
                 f'\t{net2.n_classes} output channels (classes)\n'
                 f'\t{"Bilinear" if net2.bilinear else "Transposed conv"} upscaling')

    if args.load:
        net2.load_state_dict(torch.load(args.load, map_location=device))
        logging.info(f'Model loaded from {args.load}')

    net2.to(device=device)
    try:
        t_loss2,v_loss2 = train_net(net=net2,
```

```
                epochs=args.epochs,
                batch_size=args.batch_size,
                learning_rate=args.lr,
                device=device,
                img_scale=args.scale,
                val_percent=args.val / 100,
                amp=args.amp)
    except KeyboardInterrupt:
        torch.save(net2.state_dict(), 'INTERRUPTED.pth')
        logging.info('Saved interrupt')
        raise

plt.figure()
plt.plot(t_loss2)
plt.ylabel('Loss function')
plt.xlabel('Número de batches')
plt.title('Evolución del error en el conjunto de entrenamiento')

plt.figure()
plt.plot(v_loss2,c='orange')
plt.ylabel('Loss function')
plt.xlabel('Número de batches')
plt.title('Evolución del error en el conjunto de validación')

import matplotlib.pyplot as plt

dir_img = Path('/content/Pytorch-UNet/training/image_2/')
dir_mask = Path('/content/Pytorch-UNet/training/semantic/')
checkpoint_file = '/content/Pytorch-UNet/checkpoints/checkpoint_epoch2.pth'

dataset_global = KittiDataset(dir_img, dir_mask, 'gray', scale = 1.0)

net = UNet2(n_channels=3, n_classes=12, bilinear = True)

device = torch.device('cpu')
logging.info(f'Using device {device}')
net.to(device=device)
net.load_state_dict(torch.load(checkpoint_file, map_location=device))

net.cpu()

fig, ((ax1, ax2,ax3), (ax4, ax5,ax6), (ax7, ax8,ax9),(ax10, ax11,ax12),(ax13, ax14,ax15)) = plt.
    ↪ subplots(5,3, figsize=(16,10))

axes = [(ax1, ax2,ax3), (ax4, ax5,ax6), (ax7, ax8,ax9),(ax10, ax11,ax12),(ax13, ax14,ax15)]
n=10
j=0
for i in range(n,n+5):

  axa = axes[j][0]
  axb = axes[j][1]
```

```
  axc = axes[j][2]
  mysize = dataset_global[i]['image'].size()

  if j==0:
    axa.set_title('Imagen Original')
    axb.set_title('Ground Truth')
    axc.set_title('Máscara predicha')

  img_global = dataset_global[i]['image'].reshape((1,mysize[0],mysize[1],mysize[2])) #.cuda()
  img_gmask = dataset_global[i]['mask'].reshape((1,mysize[1],mysize[2])) #.cuda()

  #print("Tamaño del tensor de entrada:", img_global.size())
  img_pred = net.forward(img_global)
  img_np = img_pred.cpu().detach().numpy()
  #print("Tamaño del tensor de salida:", img_pred.size())

  # Por hacer: dibujar img_global, img_gmask y img_np usando plt.imshow()
  # Para esto, es necesario modificar las dimensiones de las imágenes, y transformar
  # las imágenes que contienen labels a formato np.ubyte
  # Además img_global debe ser multiplicada por 255
  axa.imshow(transformar(img_global))
  axb.imshow(transformar(img_gmask))
  axc.imshow(np.transpose(transformar(img_np))) # (W,H)
  axa.axis("off")
  axb.axis("off")
  axc.axis("off")
  j+=1

import matplotlib.pyplot as plt

dir_img = Path('/content/Pytorch-UNet/testing/image_2/')
dir_mask = Path('/content/Pytorch-UNet/training/semantic/')
checkpoint_file = '/content/Pytorch-UNet/checkpoints/checkpoint_epoch2.pth'

dataset_global = KittiDataset(dir_img, dir_mask, 'gray', scale = 1.0)

net = UNet2(n_channels=3, n_classes=12, bilinear = True)

device = torch.device('cpu')
logging.info(f'Using device {device}')
net.to(device=device)
net.load_state_dict(torch.load(checkpoint_file, map_location=device))

net.cpu()

fig, ((ax1, ax2), (ax3, ax4), (ax5, ax6),(ax7, ax8),(ax9, ax10)) = plt.subplots(5,2, figsize=(16,10))

axes = [(ax1, ax2), (ax3, ax4), (ax5, ax6),(ax7, ax8),(ax9, ax10)]
n=15
j=0
for i in range(n,n+5):
```

```
892
893
894   axa = axes[j][0]
895   axb = axes[j][1]
896
897   if j==0:
898     axa.set_title('Imagen Original')
899     #axb.set_title('Ground Truth')
900     axb.set_title('Máscara predicha')
901
902   mysize = dataset_global[i]['image'].size()
903
904   img_global = dataset_global[i]['image'].reshape((1,mysize[0],mysize[1],mysize[2])) #.cuda()
905   #img_gmask = dataset_global[i]['mask'].reshape((1,mysize[1],mysize[2])) #.cuda()
906
907   #print("Tamaño del tensor de entrada:", img_global.size())
908   img_pred = net.forward(img_global)
909   img_np = img_pred.cpu().detach().numpy()
910   #print("Tamaño del tensor de salida:", img_pred.size())
911
912   # Por hacer: dibujar img_global, img_gmask y img_np usando plt.imshow()
913   # Para esto, es necesario modificar las dimensiones de las imágenes, y transformar
914   # las imágenes que contienen labels a formato np.ubyte
915   # Además img_global debe ser multiplicada por 255
916   axa.imshow(transformar(img_global))
917   axb.imshow(np.transpose(transformar(img_np))) # (W,H)
918   axa.axis("off")
919   axb.axis("off")
920   j+=1
921
922 import argparse
923 import logging
924 import sys
925 from pathlib import Path
926
927 import torch
928 import torch.nn as nn
929 import torch.nn.functional as F
930 import wandb
931 from torch import optim
932 from torch.utils.data import DataLoader, random_split
933 from tqdm import tqdm
934
935 from utils.dice_score import dice_loss
936 dir_img = Path('./training/image_2/')
937 dir_mask = Path('./training/semantic/')
938 dir_checkpoint = Path('./checkpoints/')
939
940
941 if __name__ == '__main__':
942     args = get_args_train()
```

```
943
944     logging.basicConfig(level=logging.INFO, format='%(levelname)s: %(message)s')
945     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
946     logging.info(f'Using device {device}')
947
948     # Change here to adapt to your data
949     # n_channels=3 for RGB images
950     # n_classes is the number of probabilities you want to get per pixel
951     net3 = UNet3(n_channels=3, n_classes=args.classes, bilinear=args.bilinear)
952
953     logging.info(f'Network:\n'
954                  f'\t{net3.n_channels} input channels\n'
955                  f'\t{net3.n_classes} output channels (classes)\n'
956                  f'\t{"Bilinear" if net3.bilinear else "Transposed conv"} upscaling')
957
958     if args.load:
959         net3.load_state_dict(torch.load(args.load, map_location=device))
960         logging.info(f'Model loaded from {args.load}')
961
962     net3.to(device=device)
963     try:
964         t_loss3,v_loss3 = train_net(net=net3,
965                   epochs=args.epochs,
966                   batch_size=args.batch_size,
967                   learning_rate=args.lr,
968                   device=device,
969                   img_scale=args.scale,
970                   val_percent=args.val / 100,
971                   amp=args.amp)
972     except KeyboardInterrupt:
973         torch.save(net3.state_dict(), 'INTERRUPTED.pth')
974         logging.info('Saved interrupt')
975         raise
976
977 plt.figure()
978 plt.plot(t_loss3)
979 plt.ylabel('Loss function')
980 plt.xlabel('Número de batches')
981 plt.title('Evolución del error en el conjunto de entrenamiento')
982
983 plt.figure()
984 plt.plot(v_loss3,c='orange')
985 plt.ylabel('Loss function')
986 plt.xlabel('Número de batches')
987 plt.title('Evolución del error en el conjunto de validación')
988
989 import matplotlib.pyplot as plt
990
991 dir_img = Path('/content/Pytorch-UNet/training/image_2/')
992 dir_mask = Path('/content/Pytorch-UNet/training/semantic/')
993 checkpoint_file = '/content/Pytorch-UNet/checkpoints/checkpoint_epoch2.pth'
```

```
994
995 dataset_global = KittiDataset(dir_img, dir_mask, 'gray', scale = 1.0)
996
997 net = UNet3(n_channels=3, n_classes=12, bilinear = True)
998
999 device = torch.device('cpu')
1000 logging.info(f'Using device {device}')
1001 net.to(device=device)
1002 net.load_state_dict(torch.load(checkpoint_file, map_location=device))
1003
1004 net.cpu()
1005
1006 fig, ((ax1, ax2,ax3), (ax4, ax5,ax6), (ax7, ax8,ax9),(ax10, ax11,ax12),(ax13, ax14,ax15)) = plt.
     ↪ subplots(5,3, figsize=(16,10))
1007
1008 axes = [(ax1, ax2,ax3), (ax4, ax5,ax6), (ax7, ax8,ax9),(ax10, ax11,ax12),(ax13, ax14,ax15)]
1009 n=10
1010 j=0
1011 for i in range(n,n+5):
1012
1013   axa = axes[j][0]
1014   axb = axes[j][1]
1015   axc = axes[j][2]
1016   mysize = dataset_global[i]['image'].size()
1017
1018   if j==0:
1019     axa.set_title('Imagen Original')
1020     axb.set_title('Ground Truth')
1021     axc.set_title('Máscara predicha')
1022
1023   img_global = dataset_global[i]['image'].reshape((1,mysize[0],mysize[1],mysize[2])) #.cuda()
1024   img_gmask = dataset_global[i]['mask'].reshape((1,mysize[1],mysize[2])) #.cuda()
1025
1026   #print("Tamaño del tensor de entrada:", img_global.size())
1027   img_pred = net.forward(img_global)
1028   img_np = img_pred.cpu().detach().numpy()
1029   #print("Tamaño del tensor de salida:", img_pred.size())
1030
1031   # Por hacer: dibujar img_global, img_gmask y img_np usando plt.imshow()
1032   # Para esto, es necesario modificar las dimensiones de las imágenes, y transformar
1033   # las imágenes que contienen labels a formato np.ubyte
1034   # Además img_global debe ser multiplicada por 255
1035   axa.imshow(transformar(img_global))
1036   axb.imshow(transformar(img_gmask))
1037   axc.imshow(np.transpose(transformar(img_np))) # (W,H)
1038   axa.axis("off")
1039   axb.axis("off")
1040   axc.axis("off")
1041   j+=1
1042
1043 import matplotlib.pyplot as plt
```

```
1044
1045 dir_img = Path('/content/Pytorch-UNet/testing/image_2/')
1046 dir_mask = Path('/content/Pytorch-UNet/training/semantic/')
1047 checkpoint_file = '/content/Pytorch-UNet/checkpoints/checkpoint_epoch2.pth'
1048
1049 dataset_global = KittiDataset(dir_img, dir_mask, 'gray', scale = 1.0)
1050
1051 net = UNet3(n_channels=3, n_classes=12, bilinear = True)
1052
1053 device = torch.device('cpu')
1054 logging.info(f'Using device {device}')
1055 net.to(device=device)
1056 net.load_state_dict(torch.load(checkpoint_file, map_location=device))
1057
1058 net.cpu()
1059
1060 fig, ((ax1, ax2), (ax3, ax4), (ax5, ax6),(ax7, ax8),(ax9, ax10)) = plt.subplots(5,2, figsize=(16,10))
1061
1062 axes = [(ax1, ax2), (ax3, ax4), (ax5, ax6),(ax7, ax8),(ax9, ax10)]
1063 n=15
1064 j=0
1065 for i in range(n,n+5):
1066
1067
1068   axa = axes[j][0]
1069   axb = axes[j][1]
1070
1071   if j==0:
1072     axa.set_title('Imagen Original')
1073     #axb.set_title('Ground Truth')
1074     axb.set_title('Máscara predicha')
1075
1076   mysize = dataset_global[i]['image'].size()
1077
1078   img_global = dataset_global[i]['image'].reshape((1,mysize[0],mysize[1],mysize[2])) #.cuda()
1079   #img_gmask = dataset_global[i]['mask'].reshape((1,mysize[1],mysize[2])) #.cuda()
1080
1081   #print("Tamaño del tensor de entrada:", img_global.size())
1082   img_pred = net.forward(img_global)
1083   img_np = img_pred.cpu().detach().numpy()
1084   #print("Tamaño del tensor de salida:", img_pred.size())
1085
1086   # Por hacer: dibujar img_global, img_gmask y img_np usando plt.imshow()
1087   # Para esto, es necesario modificar las dimensiones de las imágenes, y transformar
1088   # las imágenes que contienen labels a formato np.ubyte
1089   # Además img_global debe ser multiplicada por 255
1090   axa.imshow(transformar(img_global))
1091   axb.imshow(np.transpose(transformar(img_np))) # (W,H)
1092   axa.axis("off")
1093   axb.axis("off")
1094   j+=1
```

```
1095
1096 plt.figure()
1097 plt.plot(t_loss,label='4 capas')
1098 plt.plot(t_loss2,label='3 capas')
1099 plt.plot(t_loss3,label='2 capas')
1100 plt.ylabel('Loss function')
1101 plt.xlabel('Número de batches')
1102 plt.title('Evolución del error en el conjunto de entrenamiento')
1103 plt.legend()
1104
1105 plt.figure()
1106 plt.plot(v_loss,label='4 capas')
1107 plt.plot(v_loss2,label='3 capas')
1108 plt.plot(v_loss3,label='2 capas')
1109 plt.ylabel('Loss function')
1110 plt.xlabel('Número de batches')
1111 plt.ylim(0,8)
1112 plt.legend()
1113 plt.title('Evolución del error en el conjunto de validación')
```