CAPÍTULO 6. BIBLIOGRAFÍA

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag.

Chapman, P., Clinton, J., Kerber, R., Khabaza, T., Reinartz, T., Shearer, C. R. H. & Wirth, R. (2000). CRISP-DM 1.0: Step-by-step data mining guide.

Dalal, N. & Triggs, B. (2005). Histograms of oriented gradients for human detection. En *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)* (Vol. 1, 886-893 vol. 1).

Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A. & Koltun, V. (2017). CARLA: An Open Urban Driving Simulator. En *Proceedings of the 1st Annual Conference on Robot Learning* (pp. 1-16).

Gonzalez, R. C. & Woods, R. E. (2018). *Digital Image Processing* (4th). USA: Addison-Wesley Longman Publishing Co., Inc.

Goodfellow, I., Bengio, Y. & Courville, A. (2016). *Deep Learning*. http : / / www . deeplearningbook.org. MIT Press.

Granath, E. (2020). LiDAR systems: costs, integration, and major manufacturers.

Gujarati, D. (2003). *Basic Econometrics*. Economic series. McGraw Hill.

Halim, S. & Halim, F. (2013). *Competitive Programming 3: The New Lower Bound of Programming Contests*. Lulu.com.

Hastie, T., Tibshirani, R. & Friedman, J. (2001). *The Elements of Statistical Learning*. Springer Series in Statistics. New York, NY, USA: Springer New York Inc.

He, K., Gkioxari, G., Dollár, P. & Girshick, R. (2017). Mask R-CNN. En *2017 IEEE International Conference on Computer Vision (ICCV)* (pp. 2980-2988).

Hooper, J. (2004). From Darpa Grand Challenge 2004DARPA's Debacle in the Desert. *Popular Science*.

Karpathy, A. (2020). *AI for Full-Self Driving*. SCALEDML CONFERENCE.

Klette, R. (2014). *Concise Computer Vision: An Introduction into Theory and Algorithms*. Springer Publishing Company, Incorporated.

Krizhevsky, A., Sutskever, I. & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. En *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1* (pp. 1097-1105). NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc.

Lam, S. K., Pitrou, A. & Seibert, S. (2015). Numba: A LLVM-Based Python JIT Compiler. En *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. Austin, Texas: Association for Computing Machinery.

Lecun, Y., Cosatto, E., Ben, J., Muller, U. & Flepp, B. (2004). *DAVE: Autonomous Off-Road Vehicle Control Using End-to-End Learning* (inf. téc. N.º DARPA-IPTO Final Report). Courant Institute/CBLL. http://www.cs.nyu.edu/˜yann/research/dave/index.html.

LeCun, Y., Bottou, L., Bengio, Y. & Haffner, P. (1998). Gradient-Based Learning Applied to Document Recognition. En *Proceedings of the IEEE* (Vol. 86, *11*, pp. 2278-2324).

Lin, T., Goyal, P., Girshick, R., He, K. & Dollár, P. ([2017] 2020). Focal Loss for Dense Object Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *42*(2), 318-327.

List of self-driving car fatalities. (2020). Wikimedia Foundation.

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y. & Berg, A. C. (2016). SSD: Single Shot MultiBox Detector. En B. Leibe, J. Matas, N. Sebe & M. Welling (Eds.), *Computer Vision – ECCV 2016* (pp. 21-37). Cham: Springer International Publishing.

Markoff, J. (2010). Google Cars Drive Themselves, in Traffic. *The New York Times*.

Mohri, M., Rostamizadeh, A. & Talwalkar, A. (2018). *Foundations of Machine Learning* (2nd). The MIT Press.

Montabone, S. (2012). Using OpenCV in Sage.

Nelson, G. (2015). Tesla beams down 'autopilot' mode to Model S. *Automotive News*.

Redmon, J., Divvala, S. K., Girshick, R. B. & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 779-788.

Roberts, L. G. (1963). *Machine perception of three-dimensional solids* (Tesis doctoral, Massachusetts Institute of Technology).

Ronneberger, O., Fischer, P. & Brox, T. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. (Vol. 9351, pp. 234-241).

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533-536.

SAE. (2018). *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A. & Chen, L. (2018). MobileNetV2: Inverted Residuals and Linear Bottlenecks. En *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 4510-4520).

Stanford. (2020). CS231n Lecture Notes. Stanford.

Szeliski, R. (2010). *Computer Vision: Algorithms and Applications* (1st). Berlin, Heidelberg: Springer-Verlag.

Viola, P. & Jones, M. (2001). Rapid Object Detection using a Boosted Cascade of Simple Features. (Vol. 1, pp. I-511).

Wang, H., Wang, Z., Du, M., Yang, F., Zhang, Z., Ding, S., … Hu, X. (2020). Score-CAM: Score-Weighted Visual Explanations for Convolutional Neural Networks. En *The IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*.

Washington, T. (2016). *Clojure for Finance*. Packt Publishing.

Waymo. (2019). https://waymo.com/open/: Waymo Open Dataset.

Wirth, R. (2000). CRISP-DM: Towards a standard process model for data mining. En *Proceedings of the Fourth International Conference on the Practical Application of Knowledge Discovery and Data Mining* (pp. 29-39).

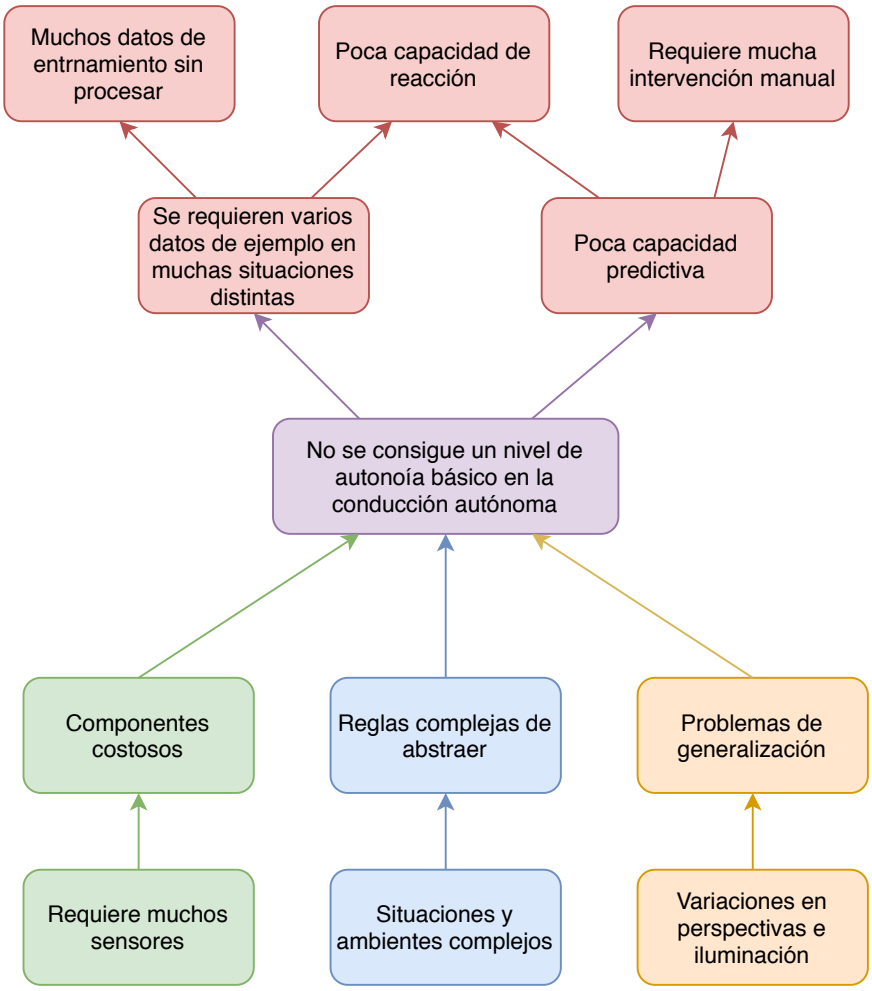
Wofk, Diana and Ma, Fangchang and Yang, Tien-Ju and Karaman, Sertac and Sze, Vivienne. (2019). FastDepth: Fast Monocular Depth Estimation on Embedded Systems. En *IEEE International Conference on Robotics and Automation (ICRA)*.

Xuming He, Zemel, R. S. & Carreira-Perpinan, M. A. (2004). Multiscale conditional random fields for image labeling. En *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.* (Vol. 2, pp. II-II).

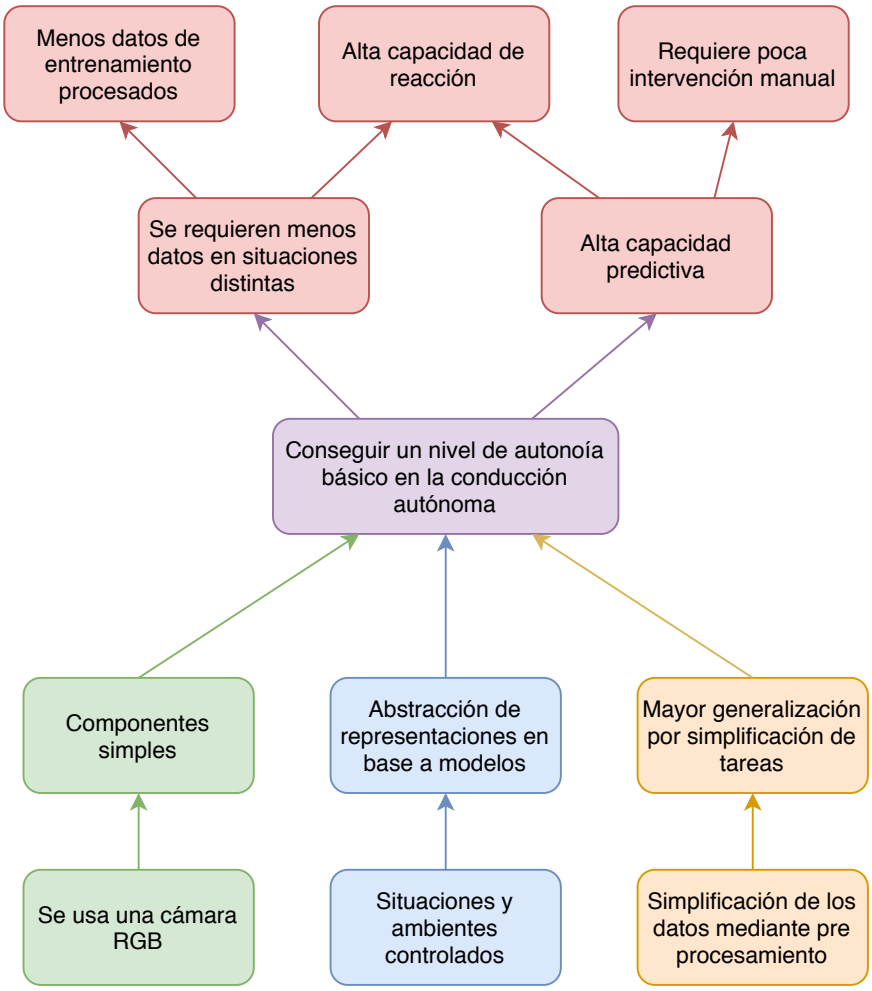
Zhang, A., Lipton, Z. C., Li, M. & Smola, A. J. (2020). *Dive into Deep Learning*. https://d2l.ai.

ANEXOS

ANEXO A - ÁRBOL DE PROBLEMAS



ANEXO B - ÁRBOL DE OBJETIVOS



ANEXO C - MARCO LÓGICO

Nivel	Resumen narrativo	Indicadores	Medios de verificacion	Supuestos
Fin	Contribuir con el desarrollo de la conducción autónoma	Investigaciones derivadas que mejores las propuestas	Tesis derivadas y comentarios sobre los resultados obtenidos	Es posible obtener resultados que apoyen en el avance de la conducción autónoma
Propósito	Plantear un modelo para la conducción autónoma que logre una autonomía básica en pistas y carreteras.	Rendimiento de los algoritmos y redes neuronales	Pruebas en un simulador	Los algoritmos y redes neuronales profundas permiten obtener buenas predicciones
Componentes	**a)** Diseñar un componente de aumentación y preprocesamiento de datos para extraer y crear un dataset con el fin de resolver la tarea. **b)** Reducir la complejidad de implementación del modelo mediante el uso de solamente una cámara. **c)** Modificar y entrenar redes neuronales con una alta exactitud en las predicciones utilizando menos requisitos de cómputo. **d)** Analizar las predicciones de los modelos entrenados para comprobar si las representaciones aprendidas son invariantes a los cambios de perspectiva, iluminación y objetos en la imagen. **e)** Combinar las salidas de algoritmos de visión computacional y modelos de aprendizaje profundo para mejorar la generalización de predicciones. **f)** Probar el rendimiento del modelo en una simulación, analizando casos de fallas y qué situaciones puede manejar correctamente.	**a)** los datos son útiles para el entrenamiento **b)** disminución de la complejidad **c)** % de exactitud en las métricas de predicciones **d)** invarianza en perspectiva e iluminación **e)** generalización en distintas situaciones de entrada **f)** % de fallos que causen accidentes en las simulaciones	Métricas del error para medir el rendimiento de los modelos Visualización de las representaciones aprendidas	**a)** datos con información importante para la tarea **b)** es posible realizar la tarea con imágenes de una cámara **c)** las modificaciones permiten entrenar redes que funcionen en la tarea de predicción **d)** es posible aprender representaciones invariantes **e)** los algoritmos de visión computacional funcionan bien junto con modelos de aprendizaje profundo **f)** el modelo funciona bien en el ambiente virtual realista y comete pocos fallos
Actividades	a) procesamiento de datos disponibles, y creación de un dataset. b) basar todos los modelos y algoritmos en imágenes de una cámara RGB c) adaptar redes neuronales que se saben que funcionan para esta tarea. d) recopilar un conjunto de datos procesados, entrenar distintas arquitecturas de redes neuronales hasta obtener buenos resultados. e) visualizar las áreas de atención del modelo, visualizar los filtros aprendidos, probar en casos extremos	Obtención de resultados funcionales	Conducción básica en un simulador	El modelo permitirá lograr una conducción autónoma básica

ANEXO D - PREPARACIÓN DE DATOS

D.1 - EXTRACCIÓN DE IMÁGENES

```
import pickle
import torch
import torchvision.transforms as transforms
from torchvision.models import mobilenet_v2
import torch.nn as nn
import glob
import os
import sys

sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
  sys.version_info.major,
  sys.version_info.minor,
  'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
sys.path.append('../carla')
sys.path.append('path/to/DriveNet')
sys.path.append('path/to/DepthNet')
sys.path.append('path/to/SemsegNet')

from sync_mode import CarlaSyncMode # Clase incluida en los ejemplos de Carla
import carla

import pygame
import numpy as np
import re
import cv2

def should_quit():
  # función incluida con la API de Carla
  for event in pygame.event.get():
    if event.type == pygame.QUIT:
      return True
    elif event.type == pygame.KEYUP:
      if event.key == pygame.K_ESCAPE:
        return True
  return False

def img_to_array(image):
  array = np.frombuffer(image.raw_data, dtype=np.dtype("uint8"))
  array = np.reshape(array, (image.height, image.width, 4))
  array = array[:, :, :3]
  return array


def show_window(surface, array, pos=(0,0)):
  if len(array.shape) > 2:
    array = array[:, :, ::-1]
  image_surface = pygame.surfarray.make_surface(array.swapaxes(0, 1))
```

```
  surface.blit(image_surface, pos)


def create_camera(cam_type, vehicle, pos, h, w, lib, world):
  cam = lib.find(f'sensor.camera.{cam_type}')
  cam.set_attribute('image_size_x', str(w))
  cam.set_attribute('image_size_y', str(h))
  camera = world.spawn_actor(
    cam,
    pos,
    attach_to=vehicle,
    attachment_type=carla.AttachmentType.Rigid)
  return camera

def find_weather_presets():
  # función incluida con la API de Carla
  rgx = re.compile('.+?(?:(?<=[a-z])(?=[A-Z])|(?<=[A-Z])(?=[A-Z][a-z])|$)')

  def name(x): return ' '.join(m.group(0) for m in rgx.finditer(x))

  presets = [x for x in dir(carla.WeatherParameters) if re.match('[A-Z].+', x)]
  return [(getattr(carla.WeatherParameters, x), name(x)) for x in presets]


if __name__ == '__main__':
  try:
    actor_list = []
    base_path = 'path/to/dataset'
    # Número de simulación
    n_sim = 1

    # Crear las carpetas para las imágenes de la simulación
    os.makedirs(f'{base_path}/Images/{n_sim}/rgb/')
    os.makedirs(f'{base_path}/Images/{n_sim}/depth/')
    os.makedirs(f'{base_path}/Images/{n_sim}/mask/')

    pygame.init()
    w, h = 240, 180
    display = pygame.display.set_mode((w, h),
                                      pygame.HWSURFACE | pygame.DOUBLEBUF)
    clock = pygame.time.Clock()

    data = {'throttle': [], 'brake': [], 'steer': [], 'junction': []}
    try:
      client = carla.Client('localhost', 2000)
      client.set_timeout(2.0)

      world = client.get_world()
      player = spawn_player(world)
      actor_list.append(player)

      weather_index = 0
```

```
weather_presets = find_weather_presets()
preset = weather_presets[weather_index]
world.set_weather(preset[0])

blueprint_library = world.get_blueprint_library()

cam_pos = carla.Transform(carla.Location(x=1.6, z=1.7))

camera_rgb = create_camera(cam_type='rgb',
             vehicle=player,
             pos=cam_pos,
             h=h, w=w,
             lib=blueprint_library,
             world=world)
actor_list.append(camera_rgb)

camera_semseg = create_camera(cam_type='semantic_segmentation',
                              vehicle=vehicle,
                              pos=cam_pos,
                              h=h, w=w,
                              lib=blueprint_library,
                              world=world)
actor_list.append(camera_semseg)

camera_depth = create_camera(cam_type='depth',
                             vehicle=vehicle,
                             pos=cam_pos,
                             h=h, w=w,
                             lib=blueprint_library,
                             world=world)
actor_list.append(camera_depth)

# Ceder el control del vahículo al Traffic Manager
player.set_autopilot(True)

with CarlaSyncMode(world, camera_rgb, camera_semseg, camera_depth,
                   fps=20) as sync_mode:
  frame = 0
  while True:
    if should_quit():
      return
    clock.tick()

    # Obtener los fotogramas síncronos de las cámaras
    snapshot, image_rgb, image_semseg, image_depth = sync_mode.tick(timeout=2.0)

    # Convertir la imagen a un 2D array
    rgb_arr = img_to_array(image_rgb)
    # Visualizar la cámara RGB
    show_window(display, rgb_arr)
    pygame.display.flip()
```

```
            c = player.get_control()

            if c.throttle != 0:
              location = player.get_location()
              wp = world.get_map().get_waypoint(location)

              # Convetir las otras cámaras en arreglos BGR
              depth_arr = img_to_array(image_depth)
              mask_arr = img_to_array(image_semseg)

              # Guardar las imagenes como archivos
              cv2.imwrite(f'{base_path}/Images/{n_sim}/rgb/{frame}.png', rgb_arr)
              cv2.imwrite(f'{base_path}/Images/{n_sim}/mask/{frame}.png', mask_arr)

              # Crear una entrada de datos en el diccionario
              data['throttle'].append(min(c.throttle, 0.4))
              data['brake'].append(c.brake)
              data['steer'].append(c.steer)
              data['junction'].append(wp.is_junction)

              frame += 1

              if frame == 8000:
                # Si se llega a los 8000 fotogramas, terminar simulacion
                return
              if frame % 1000 == 0:
                print(f'Frame: {frame}')
      finally:
        print('destroying actors.')
        for actor in actor_list:
            actor.destroy()
        world.destroy()
        pygame.quit()

        # Convertir el diccionario en un dataframe
        df = pd.DataFrame.from_dict(data)
        # Exportar el dataframe como csv para la simulación
        df.to_csv(f'{base_path}/Dfs/{n_sim}.csv', index=False)
        print('done.')

    except KeyboardInterrupt:
      print('\nFin')
```

D.2 - UNIFICACIÓN DE LOS DATAFRAMES

```
import pandas as pd
import os
from pathlib import Path

# Listar los CSVs en orden de creación
```

```
def listdir_date(dirpath):
    return map(lambda p: str(p).split('/')[-1],
               sorted(Path(dirpath).iterdir(), key=os.path.getmtime))

if __name__ == '__main__':
    path = 'path/to/dfs'

    # Cargar los CSVs ordenados por fecha como una tupla(nombre, objeto)
    dfs = [(file, pd.read_csv(f'{path}/{file}')) for file in listdir_date(path)]

    for filename, df in dfs:
        # Insertar colúmna de los nombres de archivos y número de simulación
        # correspondiente a cada csv
        names = [f'{filename.split(".")[0]}/{i}.png' for i in range(len(df))]
        df['filenames'] = names

    # Concatenar CSVs en uno solo
    whole_dataset = pd.concat(list(map(lambda x: x[1], dfs)))

    # Guardar como un nuevo archivo
    whole_dataset.to_csv('path/to/train_dataset.csv', index=False)
```

D.3 - PREPARACIÓN DE INTERSECCIONES

```
import pandas as pd
import os
from shutil import copy
from tqdm import tqdm

if __name__ == '__main__':
  df = pd.read_csv('path/to/train_dataset.csv')

  # Crear colúmnas para la carpeta y el archivo
  df['folder'] = [f.split('/')[0] for f in df['filenames'].tolist()]
  df['file'] = [f.split('/')[1].split('.')[0] for f in df['filenames'].tolist()]

  # Se elimina la colúmna que contiene ambos en uno
  df = df.drop(['filenames'], axis=1)

  # Seleccionar las filas de intersecciones
  df_junc = df.query('junction == True')

  # Extraer la lísta de índices
  idxs = df_junc.index.tolist()

  # Se buscan las intersecciones
  diffs = [0]*(len(idxs)-1)
  junc_idxs = []
  temp_idxs = []
  for i in range(len(idxs)-1):
```

```
    # Si la diferencia de índices es grande
    # son intersecciones distintas
    diff = idxs[i+1] - idxs[i]

    temp_idxs.append(idxs[i])
    if diff != 1 and (idxs[i+1] != 7999 and idxs[i] != 0):
        junc_idxs.append(temp_idxs)
        temp_idxs = []

  # Ubicación de las carpetas para cada intersección
  base_path = 'path/to/junctions'
  # Ubicación de las imágenes originales
  origin_path = 'path/to/Images'

  for i, junc_img_idxs in tqdm(enumerate(junc_idxs)):
    # Crear carpeta por intersección
    os.makedirs(f'{base_path}/{i}')
    for idx in junc_img_idxs:
      row = df.iloc[idx]
      folder, file = row['filenames'].split('/')

      # Copiar la imagen a su respectiva carpeta
      copy(f'{origin_path}/{folder}/rgb/{file}', f'{base_path}/{i}/{folder}_{file}')
```

D.4 - CREACIÓN DEL DATAFRAME FINAL

```
import pandas as pd
import os

if __name__ == '__main__':
  # CSV del conjunto de datos
  dataset = pd.read_csv('path/to/train_dataset.csv')
  # CSV de intersecciones etiquetadas
  junctions = pd.read_csv('path/to/juncs.csv')

  # Separar la colúmna filenames en una para carpetas y otra para nombre de archivos
  dataset['folder'] = [f.split('/')[0] for f in dataset['filenames'].tolist()]
  dataset['file'] = [f.split('/')[1].split('.')[0] for f in dataset['filenames'].tolist()]
  dataset = dataset.drop(['filenames'], axis=1)

  # Crear una lista vacía para cada etiqueta llena de ceros
  # (valores negativos en clasificación binaria)
  path_left = [0]*len(dataset)
  path_right = [0]*len(dataset)
  path_forward = [0]*len(dataset)

  action_left = [0]*len(dataset)
  action_right = [0]*len(dataset)
  action_forward = [0]*len(dataset)
  # Por defecto la "No acción" está marcada.
```

```
  no_action = [1]*len(dataset)

  # Ubicación de las imágenes de intersecciones
  junc_path = 'path/to/junctions'

  # Ordenar carpetas de intersección por el número de menor a mayor
  juncs = sorted(os.listdir(junc_path), key=lambda x: int(x))

  for junc in juncs:
    # Ordenar los archivos numéricamente
    files = sorted(os.listdir(f'{junc_path}/{junc}'),
                   key=lambda x: int(x.split('.')[0].split('_')[1]))

    # Indexar la fila correspondiente a la intersección
    row = junctions.iloc[int(junc)]
    for file in files:
      folder, img = list(map(int, file.split('.')[0].split('_')))
      idx = 8000*folder + img

      # Asignar los valores etiquetados
      path_left[idx] = row['path_left']
      path_right[idx] = row['path_right']
      path_forward[idx] = row['path_forward']

      action_left[idx] = row['action_left']
      action_right[idx] = row['action_right']
      action_forward[idx] = row['action_forward']
      # Como es una intersección se desmarca la "No acción"
      no_action[idx] = 0
```

ANEXO E - ENTRENAMIENTO DE REDES

E.1 - DRIVENET

```
import os
import time
import torch
import torch.nn as nn
import torch.optim
import torch.utils.data
import torchvision.transforms as transforms
from custom_mobilenet import CustomMobileNet

from PIL import Image
import numpy as np
import pandas as pd
from tqdm import tqdm

class DriveDataset(torch.utils.data.Dataset):
  def __init__(self, root_dir, transform=None):
```

```
    temp = root_dir.split('/')
    self.root_dir = '/'.join(temp[:-1])
    self.transform = transform

    self.data = pd.read_csv(root_dir)

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

  def __getitem__(self, idx):
    if torch.is_tensor(idx):
        idx = idx.tolist()

    row = self.data.iloc[idx]

    # Cargar imagen RGB
    folder, file = row['filenames'].split('/')
    img_rgb_path = os.path.join(
        self.root_dir, 'Images', folder, 'rgb', file)
    img_rgb = Image.open(img_rgb_path)

    if self.transform:
        img_rgb = self.transform(img_rgb)

    # leer etiquetas de la i-ésima muestra
    throttle = row['throttle']
    steering = row['steer']
    action_left = row['action_left']
    action_right = row['action_right']
    action_forward = row['action_forward']
    no_action = row['no_action']

    # Crear muestra
    sample = (img_rgb,
              # Tensor de parámetros de acción.
              torch.tensor([
                  float(action_left), float(action_right),
                  float(action_forward), float(no_action)
              ]),
              # Tensor de etiquetas de aceleración y giro.
              torch.tensor([
                  float(throttle), float(steering)
              ]))

    return sample


if __name__ == '__main__':
  model = CustomMobileNet(pretrained=True)

  model.cuda()
```

```
train_dir = 'path/to/train_dataset_final.csv'
val_dir = 'path/to/val_dataset_final.csv'

train_loader = torch.utils.data.DataLoader(
  dataset=DriveDataset(train_dir, transforms.Compose([
      transforms.Resize((224, 224), interpolation=Image.BICUBIC),
      transforms.ToTensor(),
      lambda T: T[:3]
  ])),
  batch_size=64,
  shuffle=True,
  num_workers=12,
  pin_memory=True
)

val_loader = torch.utils.data.DataLoader(
  dataset=DriveDataset(val_dir, transforms.Compose([
      transforms.Resize((224, 224), interpolation=Image.BICUBIC),
      transforms.ToTensor()
  ])),
  batch_size=64,
  shuffle=False,
  num_workers=12,
  pin_memory=True
)

criterion = nn.MSELoss().cuda()
optimizer = torch.optim.Adam(model.parameters())

losses = []

# Iterar para entrenar la red
for epoch in range(50):
  start = time.time()

  model.train()
  train_loss = 0
  # Definir barra de progreso interactiva
  train_progress = tqdm(enumerate(train_loader),
                        desc="train",
                        total=len(train_loader))

  # Iterar por cada minibatch de 64 muestras
  for i, (X, actions, y) in train_progress:
    # Copiar los datos a la GPU
    X = X.cuda(non_blocking=True)
    actions = actions.cuda(non_blocking=True)
    y = y.cuda(non_blocking=True)
    y_hat = model(X, actions)

    # Calcular el error cuadrático medio para la aceleración
    loss1 = criterion(y_hat[:, 0], y[:, 0])
```

```
    # Calcular el error cuadrático medio para la dirección
    loss2 = criterion(torch.tanh(y_hat[:, 1]), y[:, 1])
    # Combinar ambos errores
    loss = (loss1 + loss2)/2

    # Paso de optimización
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    train_loss += float(loss.detach())
    train_progress.set_postfix(loss=(train_loss/(i+1)))

  model.eval()

  val_loss = 0
  with torch.no_grad():
    model.eval()
    val_progress = tqdm(enumerate(val_loader),
                        desc="val",
                        total=len(val_loader))
    for i, (X, actions, y) in val_progress:
      X = X.cuda(non_blocking=True)
      actions = actions.cuda(non_blocking=True)
      y = y.cuda(non_blocking=True)
      y_hat = model(X, actions)

      loss1 = criterion(y_hat[:, 0], y[:, 0])
      loss2 = criterion(y_hat[:, 1], y[:, 1])
      loss = (loss1 + loss2) / 2

      val_loss += float(loss)
      val_progress.set_postfix(loss=(val_loss/(i+1)))

  end = time.time()

  t_loss = train_loss / len(train_loader)
  v_loss = val_loss / len(val_loader)
  print('epoch:', epoch, 'L:', t_loss, v_loss, 'Time:', end-start)

  torch.save(
    {
        'epoch': epoch,
        'arch': 'mobilenet_custom',
        'state_dict': model.state_dict()
    },
    f'weights/mob_drive_{epoch}.pth.tar')
  losses.append([epoch, t_loss, v_loss])
  np.save('hist_drive', np.array(losses))
```

E.2 - DEPTHNET

```
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.optim
import torch.utils.data
import torchvision.transforms as transforms
from models import CustomMobilenet

from PIL import Image, ImageOps
import os
import numpy as np
import pandas as pd
import time

from tqdm import tqdm

import matplotlib.pyplot as plt

# Clase encargada de cargar los datos
class DriveDepthDataset(torch.utils.data.Dataset):
  def __init__(self, root_dir, transform=None, stills=True):
    temp = root_dir.split('/')
    self.root_dir = '/'.join(temp[:-1])
    self.transform = transform

    self.data = pd.read_csv(root_dir)
    # Si se desean mantener las imágenes donde el vehículo está quieto
    if stills:
        self.data = self.data['n_id'].tolist()
    else:
        self.data = self.data.query('throttle != 0.0')['n_id'].tolist()

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

  # Sobrecarga del operador [] para indexado
  def __getitem__(self, idx):
    if torch.is_tensor(idx):
        idx = idx.tolist()

    # Decidir si espejar la imagen aleatoriamente
    h_flip = np.random.random() < 0.5

    # Cargar imagen RGB
    img_rgb_path = os.path.join(self.root_dir, 'rgb', f'{self.data[idx]}.png')
    img_rgb = Image.open(img_rgb_path)
    if h_flip:
        img_rgb = ImageOps.mirror(img_rgb)
    if self.transform:
```

```
        img_rgb = self.transform(img_rgb)

    # Cargar imagen de profundidad
    img_depth_path = os.path.join(self.root_dir, 'depth', f'{self.data[idx]}.png')
    img_depth = Image.open(img_depth_path)
    if h_flip:
        img_depth = ImageOps.mirror(img_depth)

    # Convertir la imagen de 24 bits a un mapa de profundidades [0, 1000]
    img_depth = np.transpose(np.asarray(img_depth, dtype=np.float32), (2, 0, 1))
    target = img_depth[0, :, :] + img_depth[1, :, :] * 256 + img_depth[2, :, :] * 256 * 256

    # Truncar las distancias hasta 30 metros como máximo
    target = np.clip((target / (256 * 256 * 256 - 1)) * 1000, None, 30)
    target = torch.from_numpy(target).float()

    # Crear tupla de la muestra
    sample = (img_rgb, target.view(1, target.shape[0], target.shape[1]))

    return sample

if __name__ == '__main__':
  # Valores de normalización de la imagen
  normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                   std=[0.229, 0.224, 0.225])
  # Instanciar la red de profundidades
  model = CustomMobilenetDepth((180, 240), pretrained=False)
  # Enviar parámetros a la GPU
  model.cuda()

  train_dir = 'path/to/train_data.csv'
  val_dir = 'path/to/val_data.csv'

  # Instanciar el DataLoader para el conjunto de entrenamiento
  train_loader = torch.utils.data.DataLoader(
    dataset=DriveDepthDataset(train_dir, transforms.Compose([
        transforms.ToTensor(),
        lambda T: T[:3],
        normalize
    ])),
    batch_size=64,
    shuffle=True,
    num_workers=12,
    pin_memory=True
  )

  # Instanciar el DataLoader para el conjunto de validación
  val_loader = torch.utils.data.DataLoader(
    dataset=DriveDepthDataset(val_dir, transforms.Compose([
        transforms.ToTensor(),
        normalize
    ])),
```

```
  batch_size=64,
  shuffle=True,
  num_workers=12,
  pin_memory=True
)

# Definir función de costo a optimizar
criterion = nn.MSELoss().cuda()
# Definir optimizador
optimizer = torch.optim.Adam(model.parameters())

# Historial de errores
losses = []

# Entrenar por un número de iteraciones o hasta detener el proceso
for epoch in range(50):
  # Guardar tiempo de inicio de iteración
  start = time.time()

  # Modelo en modo entrenamiento
  model.train()
  train_loss = 0
  # Iterar por los bloques de datos de 64 en 64
  for i, (X, y) in tqdm(enumerate(train_loader)):
    X = X.cuda(non_blocking=True)
    y = y.cuda(non_blocking=True)
    # Realizar la predicción
    y_hat = model(X)

    # Calcular el error
    loss = criterion(y_hat, y)
    # Acumular el error
    train_loss += float(loss.detach())

    # Derivar y actualizar los parámetros
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

  # Evaluar la red en el conjunto de validación cruzada
  val_loss = 0
  with torch.no_grad():
    model.eval()
    for i, (X, y) in enumerate(val_loader):
      X = X.cuda(non_blocking=True)
      y = y.cuda(non_blocking=True)
      y_hat = model(X)

      loss = criterion(y_hat, y)
      val_loss += float(loss)

  # Guardar el tiempo de finalización de la iteración
```

```
    end = time.time()

    # Calcular el error medio de la iteración
    t_loss = train_loss / len(train_loader)
    v_loss = val_loss / len(val_loader)
    print('epoch:', epoch, 'L:', t_loss, v_loss, 'Time:', end - start)

    # Guardar los parámetros de la i-ésima iteración
    torch.save(
      {
        'epoch': epoch,
        'arch': 'mobilenet_depth',
        'state_dict': model.state_dict()
      },
      f'weights/c_mob_{epoch}.pth.tar')
    # Guardar los errores
    losses.append([epoch, t_loss, v_loss])
    np.save('hist', np.array(losses))
```

E.3 - SEMSEGNET

```
import torch
import torch.nn as nn
import torch.optim
import torch.utils.data
import torchvision.transforms as transforms
from semseg_model import CustomMobilenetSemseg

from PIL import Image, ImageOps
import os
import numpy as np
import pandas as pd
import time

from tqdm import tqdm


class DriveSemsegDataset(torch.utils.data.Dataset):
  def __init__(self, root_dir, transform=None, stills=True, train=True):
    temp = root_dir.split('/')
    self.root_dir = '/'.join(temp[:-1])
    self.transform = transform

    self.data = pd.read_csv(root_dir)
    if stills:
        self.data = self.data['filenames'].tolist()
    else:
        self.data = self.data.query('throttle != 0.0')
        self.data = self.data['filenames'].tolist()
```

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

  def __getitem__(self, idx):
    if torch.is_tensor(idx):
        idx = idx.tolist()

    h_flip = np.random.random() < 0.5

    # Imagen RGB
    folder, file = self.data[idx].split('/')
    img_rgb_path = os.path.join(self.root_dir, 'Images', folder, 'rgb', file)
    img_rgb = Image.open(img_rgb_path)
    if h_flip:
        img_rgb = ImageOps.mirror(img_rgb)
    if self.transform:
        img_rgb = self.transform(img_rgb)

    # Máscara de segmentación
    img_semseg_path = os.path.join(self.root_dir, 'Images', folder, 'mask', file)
    img_semseg = Image.open(img_semseg_path)
    if h_flip:
        img_semseg = ImageOps.mirror(img_semseg)
    # Se extrae el canal R de la imagen en el que está codificada la máscara
    img_semseg = np.asarray(img_semseg)[:, :, 0].copy()
    sample = (img_rgb, torch.from_numpy(img_semseg))

    return sample


if __name__ == '__main__':
  # Se fija una semilla para hacer el experimento reproducible
  np.random.seed(42)

  model = CustomMobilenetSemseg((180, 240), pretrained=False)
  model = model.cuda()

  train_dir = 'path/to/train_dataset.csv'
  val_dir = 'path/to/val_dataset.csv'

  train_loader = torch.utils.data.DataLoader(
    dataset=DriveSemsegDataset(train_dir, transforms.Compose([
        transforms.ToTensor(),
        lambda T: T[:3],
    ]), stills=False),
    batch_size=64,
    shuffle=True,
    num_workers=12,
    pin_memory=True
  )

  val_loader = torch.utils.data.DataLoader(
```

```
  dataset=DriveSemsegDataset(val_dir, transforms.Compose([
      transforms.ToTensor(),
  ]), stills=False, train=False),
  batch_size=64,
  shuffle=True,
  num_workers=12,
  pin_memory=True
)

# Entropía cruzada como criterio de erro (Log loss)
criterion = nn.CrossEntropyLoss().cuda()
# Optimizador Adam con un learning rate de 0.01
optimizer = torch.optim.Adam(model.parameters(), lr=0.01, amsgrad=True)

losses = []

train_len = len(train_loader)
val_len = len(val_loader)
for epoch in range(50):
  start = time.time()

  model.train()
  train_loss = 0

  train_progress = tqdm(enumerate(train_loader), desc="train", total=train_len)
  for i, (X, y) in train_progress:
    X = X.cuda(non_blocking=True)
    y = y.cuda(non_blocking=True)
    y_hat = model(X)

    # Se calcula el error softmax 2D
    loss = criterion(y_hat, y.long())

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    loss_val = loss.detach()
    train_loss += float(loss_val)

    train_progress.set_postfix(loss=(train_loss/(i+1)))

  torch.save({
    'epoch': epoch,
    'arch': 'mobilenet_depth',
    'state_dict': model.state_dict()
  },
  f'weights/s_mob_{epoch}.pth.tar')

  val_loss = 0
  with torch.no_grad():
    model.eval()
```

```
      val_progress = tqdm(enumerate(val_loader), desc="val", total=val_len)
      for i, (X, y) in val_progress:
        X = X.cuda(non_blocking=True)
        y = y.cuda(non_blocking=True)
        y_hat = model(X)

        loss = criterion(y_hat, y.long())

        val_loss += float(loss)

        val_progress.set_postfix(loss=(val_loss/(i+1))) # Loss info

    end = time.time()

    t_loss = train_loss / len(train_loader)
    v_loss = val_loss / len(val_loader)
    print('epoch:', epoch, 'L:', t_loss, v_loss, 'Time:', end - start)

    losses.append([epoch, t_loss, v_loss])
    np.save('hist', np.array(losses))
```

ANEXO F - INFERENCIA

F.1 - COLA DE PRIORIDAD

```
from heapq import *
# Tipos de datos para anotar cada variable
from typing import List, Tuple, Any, Dict, Union, Optional

class PriorityQueue:
    def __init__(self):
        self.h = []

    def empty(self) -> bool:
        return False if self.h else True

    def push(self, x: Any) -> None:
        heappush(self.h, x)

    def pop(self) -> Any:
        return heappop(self.h)

    def top(self) -> Any:
        return self.h[0] if self.h else None

    def __len__(self) -> int:
        return len(self.h)

    def __str__(self) -> str:
```

```
        res = ''
        aux = PriorityQueue()

        while not self.empty():
            e = self.pop()
            res += str(e) + '\n'
            aux.push(e)

        self.h = aux.h
        return res.strip()
```

F.2 - CONSTANTES DE IMPLEMENTACIÓN

```
junction_data = {
  (1, 1, 230): ['l', 'f'],
  (18, -1, 195): ['l', 'f'],
  (13, -1, 20): ['l', 'f'],
  (14, -1, 160): ['l', 'f'],
  (0, -1, 230): ['r', 'f'],
  (19, 1, 195): ['r', 'f'],
  (14, 1, 20): ['r', 'f'],
  (15, 1, 160): ['r', 'f'],
  (4, 1, 230): ['l', 'r'],
  (4, -1, 125): ['r', 'f'],
  (8, -1, 125): ['l', 'r'],
  (5, 1, 125): ['l', 'f'],
  (7, 1, 195): ['l', 'r'],
  (10, 1, 20): ['l', 'r'],
  (6, -1, 160): ['l', 'r'],
  (6, 1, 265): ['l', 'f'],
  (5, -1, 265): ['r', 'f'],
  (9, -1, 265): ['l', 'r'],
  (9, 1, 90): ['l', 'r'],
  (10, -1, 90): ['r', 'f'],
  (11, 1, 90): ['l', 'f'],
  (11, -1, 55): ['l', 'r'],
  (8, 1, 55): ['l', 'f'],
  (7, -1, 55): ['r', 'f']
}

contours = np.array([
  [0,139], [0, 0],
  [240, 0], [240, 139],
  [191, 139], [135, 90],
  [110,90], [51, 139],
  [0, 139], [0, 180],
  [240, 180], [240, 139], [0, 139]
])

directions = np.asarray([
```

```
  (-1, -1),
  (0, -1),
  (1, -1),
  (-1, 0),
  (1, 0),
  (-1, 1),
  (0, 1),
  (1, 1)
])

weather_idxs = [0, 1, 2, 3, 4, 9, 10, 11, 12, 13, 14]
```

F.3 - IMPLEMENTACIÓN DEL MODELO

```
import pickle
import torch
import torchvision.transforms as transforms
from torchvision.models import mobilenet_v2
import torch.nn as nn
import glob
import os
import sys


sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
  sys.version_info.major,
  sys.version_info.minor,
  'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
sys.path.append('../carla')
sys.path.append('path/to/DriveNet')
sys.path.append('path/to/DepthNet')
sys.path.append('path/to/SemsegNet')

from sync_mode import CarlaSyncMode
from custom_mobilenet import CustomMobileNet, CustomMobileNetExt
from models import CustomMobilenetDepth
from semseg_model import CustomMobilenetSemseg
from constants import junction_data, contours, directions, weather_idxs

import carla

from collections import Counter
import pygame
import numpy as np
import queue
import re
import math
import weakref
import collections
import cv2
```

```
from PIL import Image
from priority_queue import PriorityQueue

from numba import njit

def should_quit():
  # función incluida con la API de Carla
  for event in pygame.event.get():
    if event.type == pygame.QUIT:
      return True
    elif event.type == pygame.KEYUP:
      if event.key == pygame.K_ESCAPE:
        return True
  return False


def find_weather_presets():
  # función incluida con la API de Carla
  rgx = re.compile('.+?(?:(?<=[a-z])(?=[A-Z])|(?<=[A-Z])(?=[A-Z][a-z])|$)')

  def name(x): return ' '.join(m.group(0) for m in rgx.finditer(x))

  presets = [x for x in dir(carla.WeatherParameters) if re.match('[A-Z].+', x)]
  return [(getattr(carla.WeatherParameters, x), name(x)) for x in presets]


def img_to_array(image):
  array = np.frombuffer(image.raw_data, dtype=np.dtype("uint8"))
  array = np.reshape(array, (image.height, image.width, 4))
  array = array[:, :, :3]
  return array


def show_window(surface, array, pos=(0,0)):
  if len(array.shape) > 2:
    array = array[:, :, ::-1]
  image_surface = pygame.surfarray.make_surface(array.swapaxes(0, 1))
  surface.blit(image_surface, pos)


def create_camera(cam_type, vehicle, pos, h, w, lib, world):
  cam = lib.find(f'sensor.camera.{cam_type}')
  cam.set_attribute('image_size_x', str(w))
  cam.set_attribute('image_size_y', str(h))
  camera = world.spawn_actor(
    cam,
    pos,
    attach_to=vehicle,
    attachment_type=carla.AttachmentType.Rigid)
  return camera

def set_random_weather(world):
```

```
  weather_presets = find_weather_presets()
  weather_index = np.random.choice(weather_idxs)
  preset = weather_presets[weather_index]
  world.set_weather(preset[0])
  def spawn_player(world):
  world_map = world.get_map()
  player = None

while player is None:
  spawn_points = world_map.get_spawn_points()
  spawn_point = np.random.choice(spawn_points) if spawn_points else carla.Transform()
  player = world.try_spawn_actor(world.get_blueprint_library().filter('model3')[0],
                                 spawn_point)
  return player

def load_network(model_class, path, shape=None, pretrained=None):
  if isinstance(shape, type(None)):
    model = model_class()
  else:
    model = model_class((180, 240), pretrained=False)

  epoch, arch, state_dict = torch.load(path).values()
  model.load_state_dict(state_dict)
  model = model.cuda()
  model.eval()
  return model

def take_action(world, junction_data, road_id, lane_id, action, position):
  wp = world.get_map().get_waypoint(position)
  if wp.is_junction:
    junc = wp.get_junction()
    k = (road_id, lane_id, junc.id)
    # Si llega a una intersección por lado desconocido
    if not k in junction_data.keys():
      action = [0, 0, 0, 1]
    else:
      choice = np.random.choice(junction_data[k])
      if action == [0, 0, 0, 1]:
        if choice == 'l':
          action = [1, 0, 0, 0]
        elif choice == 'r':
          action = [0, 1, 0, 0]
        elif choice == 'f':
          action = [0, 0, 1, 0]
  else:
    road_id, lane_id = wp.road_id, wp.lane_id
    action = [0, 0, 0, 1]
  return road_id, lane_id, action

def steer_correction(action, mutable_params, threshold):
  if action == [1, 0, 0, 0]:
    action_str = 'left'
```

```
      # Si no está en un rango de giro mínimo
      if not mutable_params['s'] < -threshold:
        action_str = 'force left'
        if not mutable_params['s'] < -threshold:
        mutable_params['s'] = max(-threshold - mutable_params['ac'], -0.8)
        mutable_params['ac'] += 0.02
      else:
        mutable_params['ac'] = 0
      ema_str = ''
    elif action == [0, 1, 0, 0]:
      action_str = 'right'
      # similar al caso de arriba pero al otro lado
      if not mutable_params['s'] > threshold:
        action_str = 'force right'
        mutable_params['s'] = min(threshold + mutable_params['ac'], 0.8)
        mutable_params['ac'] += 0.02
      else:
        mutable_params['ac'] = 0
      ema_str = ''
    elif action == [0, 0, 1, 0]:
      action_str = 'forward'
      if not abs(mutable_params['s']) <= threshold:
        mutable_params['s'] = -mutable_params['s']
      mutable_params['ac'] = 0
      ema_str = ''
    elif action == [0, 0, 0, 1]:
      if isinstance(mutable_params['ema'], type(None)): # inicializar EMA
        mutable_params['ema'] = mutable_params['s']
      else:
        mutable_params['ema'] = mutable_params['alpha']*mutable_params['s']+\
          (1-mutable_params['alpha'])*mutable_params['ema']
      if abs(mutable_params['s']) <= threshold:
        s = mutable_params['ema']
      else:
        ema_str = ''
      mutable_params['ac'] = 0
      action_str = 'no action'

      return action_str, ema_str # dbg

def get_class_semseg(segmentation, class_id):
  seg = ((segmentation == class_id)*255).astype(np.uint8)
  seg = cv2.morphologyEx(seg, cv2.MORPH_OPEN, np.ones((5, 5)))
  seg = cv2.dilate(seg, np.ones((5, 5)), iterations = 2)
  seg = cv2.erode(seg, np.ones((5, 5)), iterations = 1)
  return seg

def extract_objects_depth(depth_map, contours, veh, poles, ped):
  objects_depth = np.round(depth_map).astype(np.uint8)
  depth_vals_veh = cv2.bitwise_and(objects_depth, veh)
  depth_vals_pol = cv2.bitwise_and(objects_depth, poles)
  objects_depth = cv2.bitwise_or(depth_vals_veh, depth_vals_pol)
```

```
  cv2.fillPoly(objects_depth, pts=[contours], color=0)
  return objects_depth

def bounding_box(min_area, signals, directions, x_min_thresh=0):
  pq = PriorityQueue()
  if signals.sum() > 0:  # Comprobar si contiene semáforos
    bbox = find_contours(signals, directions, x_min_thresh)
    for x1, y1, x2, y2 in bbox:
      area = abs(x1 - x2)*abs(y1 - y2)
      if area > min_area and x1 > x_min_thresh:
        pq.push((-area, x1, y1, x2, y2))
    valid = True if not pq.empty() and abs(pq.top()[0]) > min_area else False
    if valid:
    _, x1, y1, x2, y2 = pq.top()
    x, y = round(x1), round(y1)
    w, h = round(abs(x1 - x2)), round(abs(y1 - y2))
        return x, y, w, h
  return None

def cluster(crop, color_code):
  cod_val = 'na'
  inp = np.float32(crop.reshape((-1,3)))
  criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
  K = 4
  _, label, center = cv2.kmeans(inp, K, None, criteria,10, cv2.KMEANS_RANDOM_CENTERS)
  res = np.uint8(center[label.flatten()]).reshape((crop.shape))
  res = res * (res > 200)
  rgb_sig = [res[:, :, 0].sum(), res[:, :, 1].sum(), res[:, :, 2].sum()]
  rc, gc, bc = rgb_sig
  if rc > 0 and gc > 0:
    if bc > 0:
      cod_val = 'g'
    else:
      cod_val = color_code[np.argmax(rgb_sig)]
  elif rc > 0 or gc > 0 or bc > 0:
    cod_val = color_code[np.argmax(rgb_sig)]
      return cod_val, res, rc, gc, bc

@njit
def ff(mat, i, j, x1, y1, x2, y2, directions):
  if (0 <= i < mat.shape[0]) and (0 <= j < mat.shape[1]) and mat[i, j] != 0:
    mat[i, j] = 0
    x1 = min(x1, j)
    y1 = min(y1, i)
    x2 = max(x2, j)
    y2 = max(y2, i)

    for dx, dy in directions:
      x1, y1, x2, y2 = ff(mat, i+dy, j+dx, x1, y1, x2, y2, directions)
    return x1, y1, x2, y2

@njit
```

```
def find_contours(img, directions, x_min_thresh=0, pad=1, copy=True):
  if copy:
    local_img = img.copy()
  else:
    local_img = img
  boxes = []

  for i in range(0, local_img.shape[0]):
    for j in range(x_min_thresh, local_img.shape[1]):
      if local_img[i, j] != 0:
        x1, y1, x2, y2 = ff(local_img, i, j,
                            np.inf, np.inf, -np.inf, -np.inf, directions)
        boxes.append((int(max(x1-pad, 0)),
               int(max(y1-pad, 0)),
               int(min(x2+pad, local_img.shape[1])),
               int(min(y2+pad, local_img.shape[0]))))

  # retorna en formato x1, y1, x2, y2
  return boxes

if __name__ == '__main__':
  try:
      actor_list = []

  # 1. Definir la pantalla de visualización de imágenes mediante la librería PyGame.
    w, h = 240, 180
    pygame.init()
    display = pygame.display.set_mode((2*w, 2*h),
                                      pygame.HWSURFACE | pygame.DOUBLEBUF)
    clock = pygame.time.Clock()

    try:
  # 2. Conectar con el simulador e inicializar el vehículo en el mapa.
      client = carla.Client('localhost', 2000)
      client.set_timeout(2.0)

      world = client.get_world()
      player = spawn_player(world)
      actor_list.append(player)

      set_random_weather(world)

      blueprint_library = world.get_blueprint_library()
          font = cv2.FONT_HERSHEY_SIMPLEX

      cam_pos = carla.Transform(carla.Location(x=1.6, z=1.7))
      camera_rgb = create_camera(cam_type='rgb',
                   vehicle=player,
                   pos=cam_pos,
                   h=h, w=w,
                   lib=blueprint_library,
                   world=world)
```

```
    actor_list.append(camera_rgb)

# 3. Cargar los parámetros de las redes.
    model_depth = load_network(CustomMobilenetDepth,
                 'path/to/weights,
                 (180, 240), False)

    normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                  std=[0.229, 0.224, 0.225])
    depth_preprocess = transforms.Compose([
      transforms.ToTensor(),
      lambda T: T[:3],
      normalize
    ])

    model_drive = load_network(CustomMobileNet, 'path/to/weights')

    drive_preprocess = transforms.Compose([
      transforms.Resize((224, 224), interpolation=Image.BICUBIC),
      transforms.ToTensor(),
    ])

    model_semseg = load_network(CustomMobilenetSemseg, 'path/to/weights',
                                (180, 240), False)

    with CarlaSyncMode(world, camera_rgb, fps=30) as sync_mode:
      road_id, lane_id = 0, 0
      action = [0, 0, 0, 1]

      mutable_params = {'ema': None, 'alpha': 0.75, 'ac': 0,
                        's': None, 't': None, 'b': 0}

      threshold = 0.2
      min_area = 95
      x_min_thresh = 120
      color_code = ['r', 'g', 'b']

# 4. Iniciar un bucle de iteraciones de la simulación a 30 fps.
      while True:
        if should_quit():
          return
        clock.tick()

        # Avanzar un tick de la simulación
        snapshot, image_rgb = sync_mode.tick(timeout=2.0)

        # Convertir a un arreglo BGR
        rgb_arr = img_to_array(image_rgb)

        # Convertir a RGB
        X_img = Image.fromarray(cv2.cvtColor(rgb_arr, cv2.COLOR_BGR2RGB))
```

```
        X_drive = drive_preprocess(X_img).view(1, 3, 224, 224).cuda()
        X_depth = depth_preprocess(X_img).view(1, 3, 180, 240).cuda()

        position = player.get_location()
# 5. Verifica con el simulador si está en una intersección y decide el camino
        road_id, lane_id, action = take_action(world, junction_data, road_id,
                                               lane_id, action, position)
        action_tensor = torch.tensor(action).view(1, -1).cuda()

# 6. Ingresar los valores de entrada a cada red neuronal y recibir su salida.
        with torch.no_grad():
          pred_drive = model_drive(X_drive, action_tensor).cpu()
          depth_map = model_depth(X_depth)[0, 0].cpu().numpy()
          segmentation = model_semseg(X_drive).cpu().numpy()[0]\
                                                  .argmax(axis=0).astype(np.uint8)

        mutable_params['t'] = round(float(pred_drive[0, 0]), 3)
        mutable_params['s'] = round(float(pred_drive[0, 1]), 3)
        mutable_params['t'] = min(mutable_params['t'], 0.5)


# 7. Corregir las oscilaciones con EMA.
# 8. Si el vehículo gira lo suficiente, se da un impulso mediante un acumulador.
        action_str, ema_str = steer_correction(action, mutable_params, threshold)

# 9. Se procesan las predicciones de la segmentación semántica para obstáculos.
        walkers = get_class_semseg(segmentation, 4)
        vehicles = get_class_semseg(segmentation, 10)
        poles = get_class_semseg(segmentation, 5)

        v_boxes = find_contours(vehicles, directions, pad=1, copy=True)
        w_boxes = find_contours(walkers, directions, pad=1, copy=True)
        p_boxes = find_contours(poles, directions, pad=1, copy=True)

        objects_depth = extract_objects_depth(depth_map, contours,
                                              vehicles, poles, walkers)

# 10. Se calcula la moda de las distancias de objetos cercanos.
        if np.sum(objects_depth) != 0:
          c = sorted(Counter(list(objects_depth.ravel())).items(),
                     key=lambda x: -x[1])
          if c[0][0] == 0:
            moda = c[1][0]
            count = c[1][1]
          else:
            moda = c[0][0]
            count = c[0][1]
        else:
          moda = np.inf
          count = np.inf

        if moda <= 4 and 40 <= count < np.inf:
```

```
            mutable_params['s'], mutable_params['t'] = 0, 0
            mutable_params['b'] += 0.10
            mutable_params['b'] = round(min(max(0, mutable_params['b']), 1), 3)
          else:
            mutable_params['b'] = 0

# 11.  Se detecta la posición de los semáforos.
          signals = get_class_semseg(segmentation, 12)
          box = bounding_box(min_area, signals, directions, x_min_thresh)

# 12. Se predice un código de color, r para rojo o g para verde.
          code_val = 'na'
          signal_img = np.ones((180, 60, 3))
          rc, gc, bc = 0, 0, 0
          k_img = None
          if isinstance(box, tuple):
            x, y, w, h = box
            crop = cv2.resize(cv2.cvtColor(rgb_arr[y:y+h, x:x+w, :],
                                           cv2.COLOR_BGR2RGB), None, fx=3, fy=3)
            # K-MEANS
            code_val, k_img, rc, gc, bc = cluster(crop, color_code)

# 13. Se usa el código de color para decidir si frenar o no.
          if code_val == 'r':
            mutable_params['b'] = 1
            mutable_params['s'] = 0
            mutable_params['t'] = 0

# 14. Se envían las decisiones finales de control al vehículo.
          player.apply_control(carla.VehicleControl(throttle= mutable_params['t'],
                               steer= mutable_params['s'],
                               brake= mutable_params['b']))
          line1 = f"t: {round(mutable_params['t'], 2)} \
                    b: {round(mutable_params['b'], 2)} \
                    s: {round(mutable_params['s'], 2)} \
                    color: {code_val}"
          line2 = f"moda: {moda}, r: {int(rc>0)} g: {int(gc>0)} b: {int(bc>0)}"
          line3 = f"act: {action_str} {action}"
          rgb_arr = cv2.resize(rgb_arr, None, fx=2, fy=2,
                               interpolation=cv2.INTER_CUBIC)
          pos1 = (10, 30)
          pos2 = (10, 60)
          black, green = (0, 0, 0), (156, 237, 58)
          cv2.putText(rgb_arr, line1, pos1, font, 0.6, black, 3, cv2.LINE_AA)
          cv2.putText(rgb_arr, line1, pos1, font, 0.6, green, 1, cv2.LINE_AA)
          cv2.putText(rgb_arr, line2, pos2, font, 0.6, black, 3, cv2.LINE_AA)
          cv2.putText(rgb_arr, line2, pos2, font, 0.6, green, 1, cv2.LINE_AA)
          cv2.putText(rgb_arr, line3, pos3, font, 0.6, black, 3, cv2.LINE_AA)
          cv2.putText(rgb_arr, line3, pos3, font, 0.6, green, 1, cv2.LINE_AA)

          # visualización de los rectángulos delimitando los objetos
          if not isinstance(box, type(None)):
```

```
            color = (58, 58, 237) if code_val == 'r' else (58, 237, 67)
            cv2.rectangle(rgb_arr, (box[0]*2, box[1]*2),
                          ((box[0]+box[2])*2, (box[1]+box[3])*2), color, 2)

          for x1, y1, x2, y2 in v_boxes:
            cv2.rectangle(rgb_arr, (x1*2, y1*2), (x2*2, y2*2), (237, 152, 58), 2)
          for x1, y1, x2, y2 in p_boxes:
            cv2.rectangle(rgb_arr, (x1*2, y1*2), (x2*2, y2*2), (115, 58, 237), 2)
          for x1, y1, x2, y2 in w_boxes:
            cv2.rectangle(rgb_arr, (x1*2, y1*2), (x2*2, y2*2), (58, 161, 237), 2)

          show_window(display, rgb_arr)

          # visualizar el semáforo segmentado
          if not isinstance(k_img, type(None)):
            k_img = cv2.cvtColor(k_img, cv2.COLOR_RGB2BGR)
            k_img = cv2.resize(k_img, (31, 55), interpolation=cv2.INTER_CUBIC)
            show_window(display, k_img, (rgb_arr.shape[1]-k_img.shape[1], 0))

          pygame.display.flip()
      finally:
        for actor in actor_list:
          actor.destroy()
        pygame.quit()
    except KeyboardInterrupt:
      print('\nFin')
```