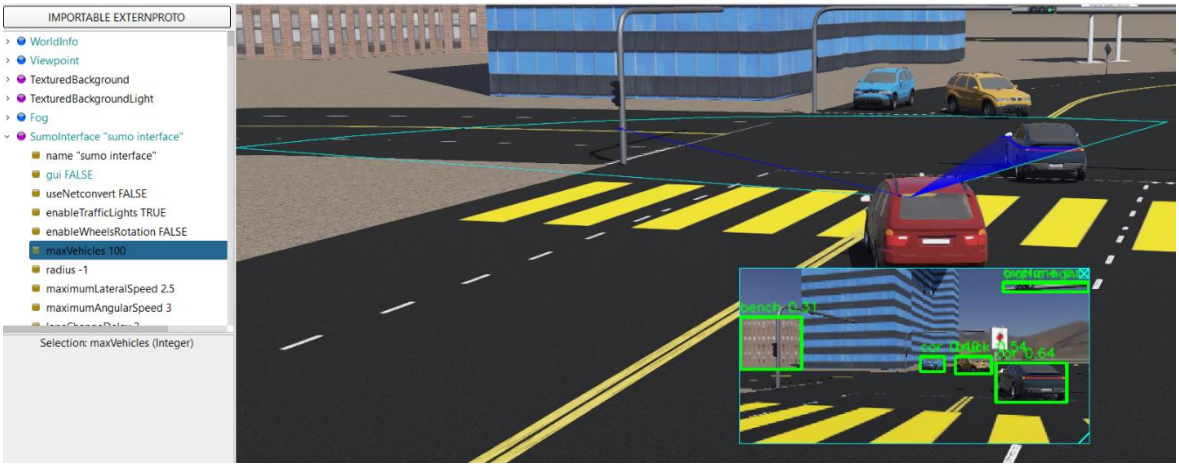
María Paula Gutiérrez Cervantes

Marco Antonio Arellano Hernández

Luis Eduardo Calvillo Corona

Juan Manuel Hernández Carrillo

Autonomous Car in Webots



Monterrey, N.L.

22 de junio del 2025

1. Introducción

El desarrollo de vehículos autónomos ha avanzado considerablemente en los últimos años, impulsado por mejoras en áreas como visión por computadora, machine learning y entornos de simulación. Uno de los enfoques más utilizados para entrenar modelos de conducción es el Behavioral Cloning, este método se basa en la idea de que, si un modelo es capaz de observar suficientes ejemplos de conducción humana, puede aprender a tomar decisiones similares, sin necesidad de programar explícitamente todas las reglas posibles del entorno.

En este proyecto final, trabajamos con un entorno simulado en Webots para entrenar una red neuronal convolucional que pueda predecir el ángulo de dirección del vehículo a partir de imágenes capturadas por la cámara frontal. La idea fue generar un dataset desde la simulación, entrenar un modelo tipo Nvidia en Python, y probar su desempeño conduciendo de forma autónoma.

Durante el desarrollo del proyecto se aplicaron conocimientos adquiridos a lo largo del trimestre, incluyendo el preprocesamiento de imágenes, el diseño e implementación de redes neuronales convolucionales (CNN) y la integración de simulación con control inteligente. Además, se documentaron de forma colaborativa los avances, obstáculos y decisiones técnicas, lo que permitió mantener una trazabilidad clara de las etapas del desarrollo.

Este documento presenta el fundamento teórico, la metodología empleada, el diseño del modelo implementado y los resultados obtenidos al evaluar el comportamiento del vehículo dentro del entorno de simulación. Asimismo, se reflexiona sobre los aprendizajes adquiridos, las áreas de oportunidad detectadas y las recomendaciones para futuras implementaciones más robustas.

2. Descripción del problema

A diferencia de ejercicios anteriores más controlados, en este caso se integra la herramienta SUMO para simular un entorno con tráfico, peatones y otros vehículos, lo que aumenta la complejidad del escenario. Esto permite que el modelo sea entrenado y evaluado en condiciones más dinámicas, con distracciones visuales y movimiento externo, similares a los que enfrentaría un sistema autónomo real.

El enfoque se centra en lograr una navegación estable dentro del carril sin intervención humana, utilizando únicamente entradas visuales. El sistema fue entrenado con un dataset generado por el propio equipo y evaluado posteriormente dentro del entorno simulado, analizando su capacidad para replicar el comportamiento humano en condiciones variadas.

3. Metodología

3.1. Carga de librerías y dependencias

Se incluyeron librerías estándar como os para la gestión de archivos, así como *numpy* y *math* para operaciones numéricas que permiten interpretar lecturas del sensor de dirección

y realizar cálculos de control. También se utiliza *cv2* (OpenCV) para la manipulación de imágenes, dado que las capturas de la cámara frontal se obtienen como matrices que deben ser procesadas, guardadas y eventualmente etiquetadas.

Por otro lado, se incorporó un módulo adicional para la gestión del modelo YOLO mediante la librería *ultralytics*, que permitió realizar detección de objetos en tiempo real dentro del entorno de simulación. Esta herramienta fue clave para incorporar la percepción de otros vehículos y peatones, facilitando decisiones de conducción más cercanas a un escenario realista. Finalmente, se importó *tensorflow.keras.models.load_model* para la carga del modelo previamente entrenado, el cual fue guardado en un archivo *.h5* e incluye tanto su arquitectura como sus pesos.

```
"""camera_pid controller."""

from controller import Display, Keyboard, Robot, Camera
from vehicle import Car, Driver
import numpy as np
import cv2
from datetime import datetime
import os
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import load_model
from ultralytics import YOLO
import math
#from roboflow import Roboflow
#import supervision as sv
#from inference_sdk import InferenceHTTPClient

model2_path=r'C:\Users\leduc\Documents\ITESM\MNA\Navegacion\MR4010ProyectoFinal2025\yolo11n.pt'
model2 = YOLO(model2_path)  #yolo11n, yolo12s
```

Figura 1.

3.2. Dataset.

El modelo fue entrenado utilizando un conjunto de aproximadamente 15,000 imágenes capturadas en un entorno simulado de conducción. Las imágenes provienen de un sistema de tres cámaras montadas en el vehículo: una frontal, una lateral izquierda y una lateral derecha. Esta configuración permite capturar distintas perspectivas del camino, lo cual es útil para enseñar al modelo a mantener el vehículo centrado y corregir desviaciones.
Cada imagen está asociada a un valor de ángulo de dirección (steering angle) que indica la acción que el vehículo debería tomar en ese momento. Las imágenes fueron redimensionadas a 66 × 200 píxeles y normalizadas antes de ser utilizadas como entrada de la red. Las tomas incluyen una variedad de condiciones de manejo, como curvas cerradas, caminos rectos, cambios de carril y distintos escenarios visuales, lo que ayuda al modelo a generalizar su comportamiento ante distintas situaciones.

3.3. Arquitectura del modelo

Esta red toma como entrada una imagen en color capturada por una cámara montada en el vehículo, con dimensiones de 66 píxeles de alto por 200 píxeles de ancho y tres canales (RGB). Antes de ser procesada por las capas convolucionales, la imagen pasa por una etapa de normalización que ajusta los valores de los píxeles a un rango adecuado para el entrenamiento del modelo, lo cual ayuda a estabilizar y acelerar el aprendizaje.

La sección convolucional de la red está compuesta por cinco capas. Las tres primeras utilizan filtros de tamaño 5x5 y tienen como objetivo extraer características espaciales como bordes, líneas y texturas relevantes para la navegación. Estas capas producen mapas de activación con profundidades crecientes (24, 36 y 48 filtros respectivamente) y reducen progresivamente la resolución espacial de la imagen. Las dos capas convolucionales restantes emplean filtros más pequeños, de 3x3, con 64 filtros cada una, lo que permite capturar patrones más locales y específicos en regiones visuales más compactas. A lo largo de esta etapa, el modelo prescinde del uso de capas de pooling, y en su lugar, la reducción de tamaño se logra mediante el uso de strides adecuados en las convoluciones.

Una vez finalizada la extracción de características, la salida de la última capa convolucional es aplanada (flattened) en un vector unidimensional, que es introducido en una serie de capas completamente conectadas (fully connected). Esta parte densa de la red está formada por cuatro capas con 1164, 100, 50 y 10 neuronas respectivamente. Su función es combinar de manera no lineal las características extraídas y generar una representación abstracta de alto nivel que sea útil para la tarea de predicción.

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Dense, Flatten, Lambda

model = Sequential()

model.add(Lambda(lambda x: x / 127.5 - 1.0, input_shape=(66, 200, 3)))

model.add(Conv2D(24, (5, 5), strides=(2, 2), activation='relu'))
model.add(Conv2D(36, (5, 5), strides=(2, 2), activation='relu'))
model.add(Conv2D(48, (5, 5), strides=(2, 2), activation='relu'))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Conv2D(64, (3, 3), activation='relu'))

model.add(Flatten())

model.add(Dense(1164, activation='relu'))
model.add(Dense(100, activation='relu'))
model.add(Dense(50, activation='relu'))
model.add(Dense(10, activation='relu'))

model.add(Dense(1))

model.compile(optimizer=tf.keras.optimizers.Adam(1e-4), loss='mse')
```

Finalmente, la última capa consiste en una sola neurona cuya salida corresponde al valor del control vehicular, típicamente el ángulo de dirección y el modelo fue guardado como “model.h5”

```python
MODEL_PATH = r'C:\Users\leduc\Documents\ITESM\MNA\Navegacion\MR4010ProyectoFinal2025\model.h5'
    # Carga del modelo (al inicio, una sola vez)
    model = load_model(MODEL_PATH)
```

Figura 2.

3.4. Preprocesamiento de imágenes

El preprocesamiento de imágenes es una etapa clave en el funcionamiento del modelo, ya que permite adaptar la información visual del entorno simulado al formato adecuado para ser interpretado por la red neuronal. Esta etapa comienza con la captura de imágenes desde la cámara frontal del vehículo, configurada como un sensor RGB en Webots.

Para transformar la señal capturada, se implementaron varias funciones específicas. Primero, la imagen es recuperada mediante *get_image(camera)*, donde se convierte a un arreglo de *numpy* y se reordena a formato RGB.

```python
#Getting image from camera
def get_image(camera):
    raw_image = camera.getImage()
    image = np.frombuffer(raw_image, np.uint8).reshape(
        (camera.getHeight(), camera.getWidth(), 4)
    )
    #return image
# --- LLAMADA A LA NUEVA FUNCIÓN AQUÍ ---
    image_rgb = rgba_to_rgb(image)
    return image_rgb # Retorna la imagen en formato RGB (3 canales)
```

Figura 3.

Si se requiere mostrar o manipular la imagen original en formato BGRA, se utiliza la función *get_image_disp(camera)*.

```python
def get_image_disp(camera):
    raw_image = camera.getImage()
    image = np.frombuffer(raw_image, np.uint8).reshape(
        (camera.getHeight(), camera.getWidth(), 4)
    )
    return image
```

Figura 4.

Además, el módulo *rgba_to_rgb* permite convertir imágenes de 4 canales (RGBA) a 3 canales (RGB), conservando únicamente la información útil para el modelo.

```python
def rgba_to_rgb(image_rgba):
    # La cámara de Webots devuelve RGBA, pero OpenCV usa BGR por defecto.
    # Primero, convertimos RGBA a BGRA, luego eliminamos el canal alfa, luego convertimos BGRA a BGR.
    # O, más directo: eliminar el canal alfa y luego convertir BGR a RGB si lo necesitas para otras funciones.
    # Para el modelo Keras, la mayoría de los modelos de visión esperan RGB (o BGR si fueron entrenados con OpenCV).
    # NVIDIA End-to-End Steering (el modelo de referencia), asume RGB.

    # Asegúrate de que la imagen sea de 4 canales antes de intentar slice
    if image_rgba.shape[2] == 4:
        # Eliminar el canal alfa
        image_rgb = image_rgba[:, :, :3]
    else:
        # Si no tiene 4 canales, asumimos que ya está en RGB o en un formato compatible
        image_rgb = image_rgba

    # Opcional: Si necesitas que sea BGR para otras funciones de OpenCV que esperan BGR:
    # image_bgr = cv2.cvtColor(image_rgb, cv2.COLOR_RGB2BGR)

    return image_rgb # Retorna una imagen con 3 canales RGB
```

Figura 5.

Posteriormente, la imagen se procesa para coincidir con el tamaño de entrada de la red. Se hace un redimensionamiento a 200x66 píxeles usando *cv2.resize()*. Esta dimensión es común en modelos inspirados en la arquitectura de NVIDIA, utilizada para conducción autónoma. Luego, la imagen es normalizada con la operación *(image / 127.5) - 1.0*, lo cual ajusta los valores de los píxeles al rango [-1, 1] y facilita la estabilidad numérica durante la inferencia.

```python
    resized = cv2.resize(img, (INPUT_WIDTH, INPUT_HEIGHT))

    # La normalización para modelos NVIDIA suele ser (imagen / 127.5) - 1.0,
    # lo que escala los píxeles de [0, 255] a [-1, 1].
    normalized = resized / 127.5 - 1.0

    # Keras espera un batch dimension al inicio, incluso si es un solo sample.
    # El modelo espera (1, height, width, channels)
    return normalized.reshape((1, INPUT_HEIGHT, INPUT_WIDTH, 3))
```

Figura 6.

Además de este preprocesamiento para el modelo de conducción, se incluye una conversión adicional a escala de grises *(greyscale_cv2)*.

```python
#Image processing
def greyscale_cv2(image):
    gray_img = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    return gray_img
```

Figura 7.

y la detección de bordes mediante el algoritmo de Canny. Ambas son técnicas útiles para la detección de carriles o elementos visuales en la carretera.

```
66  def detect_edges(gray_img, low_threshold=50, high_threshold=100):
67      return cv2.Canny(gray_img, low_threshold, high_threshold,apertureSize=3)
```

Figura 8.

La función *region_of_interest* se emplea para aislar únicamente la parte inferior de la imagen, correspondiente al parabrisas, que representa la zona útil para la toma de decisiones.

```
476         image=region of interest(image)
```

Figura 9.

Una vez preprocesada, la imagen se pasa al modelo entrenado lo que permite generar una predicción de dirección de manejo.

```
478         proc = preprocess(image)
```

Figura 10.

Paralelamente, si se activa la detección de objetos, se invoca la función *detect_objects_yolo*, que transforma la imagen y aplica el modelo YOLO para identificar peatones y vehículos en el entorno.

```
399  def detect_objects_yolo(image):
400      # Convertir la imagen de BGRA a BGR
401      image_rgb = cv2.cvtColor(image, cv2.COLOR_BGRA2RGB)
402      # Realizar la detección
403      results = model2(image_rgb)[0]
```

Figura 11.

3.5. Inferencia del modelo y control del vehículo

Una vez que la imagen ha sido preprocesada, se alimenta directamente al modelo de red neuronal previamente entrenado. Esta inferencia se realiza en tiempo real dentro del simulador Webots, y su propósito principal es calcular el valor de dirección (steering angle) que debe adoptar el vehículo para mantenerse en el camino.

La imagen procesada *(proc)* se pasa como entrada al modelo cargado mediante la instrucción *angle = model.predict(np.array([proc]))*. Aquí, la imagen se empaqueta dentro de un arreglo para coincidir con la forma esperada por el modelo (batch_size = 1), y se obtiene el valor predicho para la dirección.

```
478         proc = preprocess(image)
479         #   2 Obtener predicción de ángulo
480         pred_angle = float(model.predict(proc, verbose=0)[0])
481
482         # cambia el ángulo del volante
483         if pred_angle == 0: #straight
484             print("derecha la flecha")
485         elif pred_angle > 0: #down
486             change_steer_angle_new(pred_angle)
487             print("derecha")
488         elif pred_angle < 0: #left
489             change_steer_angle_new(pred_angle)
490             print("izquierda")
```

Figura 12.

El valor de salida corresponde al ángulo de giro que debe tomar el vehículo. Para ejecutar ese giro, se actualiza el valor del motor de dirección con el método *steering_motor.setPosition(angle)*, también ubicado en el bloque principal del ciclo de simulación. Este mecanismo emula el comportamiento de un volante automático, ajustando la dirección del vehículo según las predicciones del modelo.

Este proceso de inferencia permite evaluar en tiempo real la capacidad del modelo para generalizar en diferentes entornos y situaciones no vistas durante el entrenamiento. Dado que el modelo toma decisiones basadas exclusivamente en la imagen actual, su desempeño depende críticamente de la calidad del preprocesamiento y de la robustez de los patrones aprendidos. Así, la integración entre percepción visual e inferencia directa es lo que habilita el ciclo autónomo de *percepción -> decisión -> ejecución* que caracteriza a los sistemas de conducción automática.

3.6. Detección de objetos usando YOLO

Para complementar las decisiones de navegación basadas en imágenes del camino, se integró un modelo de detección de objetos YOLO previamente entrenado. Este modelo se encarga de identificar elementos relevantes como peatones y vehículos dentro del entorno simulado en Webots, añadiendo una capa crítica de percepción para evaluar la complejidad del entorno vial.

La lógica de detección se desarrolla en la función *detect_objects_yolo(image),* sobre la cual la imagen es primero convertida al formato RGB, y posteriormente se realiza la inferencia con la instrucción *results = model2(image_rgb)*. Las predicciones obtenidas se procesan para dibujar los cuadros delimitadores y las etiquetas correspondientes, las cuales se sobreponen sobre la imagen original mediante *cv2.rectangle* y *cv2.putText.*

```python
def detect_objects_yolo(image):
    # Convertir la imagen de BGRA a BGR
    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGRA2RGB)
    # Realizar la detección
    results = model2(image_rgb)[0]

    annotated = image_rgb.copy()
    for box, conf, cls in zip(results.boxes.xyxy, results.boxes.conf, results.boxes.cls):
        x1, y1, x2, y2 = box.int().tolist()
        label = model2.names[int(cls)]
        text = f"{label} {conf:.2f}"
        cv2.rectangle(annotated, (x1, y1), (x2, y2), (0,255,0), 2)

        cv2.putText(
            annotated, text, (x1, y1 - 5),
            fontFace=cv2.FONT_HERSHEY_SIMPLEX,
            fontScale=0.4,
            color=(0,255,0),
            thickness=1,
            lineType=cv2.LINE_AA
        )
    return annotated
```

Figura 13.

En el bucle principal del programa, la función es llamada y las detecciones se almacenan en *yolo_image*. Finalmente, la imagen con los objetos detectados se visualiza mediante un *display_image*, lo que permite confirmar visualmente el correcto funcionamiento del sistema.

```python
        yolo_image = detect_objects_yolo(image_disp)
        #print(f"yolo_image: {yolo_image}")
        display_image_test(display_img, yolo_image)
        #display_image(display_img_2, vis)
```

Figura 14.

3.7. Detección de carriles y cálculo del ángulo de dirección

Durante el desarrollo del sistema de navegación, decidimos complementar el comportamiento del vehículo con un módulo adicional de visión por computadora basado en técnicas clásicas. Esta decisión surgió a partir de nuestras observaciones en simulación, donde notamos que, en ciertos tramos del entorno, especialmente curvas o zonas poco definidas, el modelo basado en *Behavioral Cloning* podía mostrar comportamientos erráticos.

Por ello, integramos un sistema de detección de carriles utilizando bordes y transformada de Hough como apoyo visual. Aplicamos *cv2.Canny* para extraer los bordes de la imagen, y se utiliza *cv2.HoughLinesP* para detectar segmentos rectos que representen los límites del carril.

```python
# Hough line detection (Probabilistic)
# dst: Output of the edge detector. It should be a grayscale image (although in fact it is a binary one)
#lines: A vector that will store the parameters (r,θ) of the detected lines
#rho : The resolution of the parameter r in pixels. We use 1 pixel.
#theta: The resolution of the parameter θ in radians. We use 1 degree (CV_PI/180)
#threshold: The minimum number of intersections to "*detect*" a line

def detect_lines(edges, rho=1, theta=np.pi/180, threshold=30,
                 min_line_length=20, max_line_gap=50):
    return cv2.HoughLinesP(
        edges, rho, theta, threshold,
        minLineLength=min_line_length,
        maxLineGap=max_line_gap
    )

```

Figura 15.

Esta información es procesada en la función *set_steering_angle_from_hough_lines*, donde se calcula un ángulo con base en la pendiente de las líneas detectadas. En los casos donde no se detectan líneas, el sistema recurre a un valor por defecto para mantener la estabilidad direccional. El ángulo estimado por este método se combina con el ángulo predicho por la red neuronal, utilizando un promedio ponderado que permite suavizar las decisiones del sistema. Esta integración nos permitió reforzar la toma de decisiones en situaciones difíciles, logrando un control más confiable y estable dentro del entorno de simulación.

```python
def set_steering_angle_from_hough_lines(lines):

    global angle, steering_angle, frames_without_detection, decay_rate

    if lines is None or not lines.any():
        print("No se detectaron líneas Hough.")
        frames_without_detection += 1
        steering_angle *= (1 - decay_rate) # Reducir gradualmente el ángulo
        angle = steering_angle
        print(f"Perdiendo carriles, ángulo actual: {steering_angle:.4f}")
        return

    frames_without_detection = 0

    line_angles = []
    for line in lines:
        for x1, y1, x2, y2 in line:
            angle_rad = math.atan2(y2 - y1, x2 - x1)
            angle_deg = math.degrees(angle_rad)
            line_angles.append(angle_deg)

    if not line_angles:
        print("No se encontraron ángulos de línea válidos.")
        return

    average_angle_degrees = np.mean(line_angles)
    steering_adjustment = -average_angle_degrees * 0.05

    desired_steering_angle = steering_angle + steering_adjustment

```

```python
    # --- Aplicar las limitaciones de dirección ---
    max_steering_change = 0.1
    if (desired_steering_angle - steering_angle) > max_steering_change:
        desired_steering_angle = steering_angle + max_steering_change
    if (desired_steering_angle - steering_angle) < -max_steering_change:
        desired_steering_angle = steering_angle - max_steering_change

    max_steering_angle = 0.1
    if desired_steering_angle > max_steering_angle:
        desired_steering_angle = max_steering_angle
    elif desired_steering_angle < -max_steering_angle:
        desired_steering_angle = -max_steering_angle

    steering_angle_hough = desired_steering_angle

    print(f"Ángulo de dirección aplicado (rad): {steering_angle_hough:.4f}")
    return(steering_angle_hough)

```

Figura 16.

3.8. Conducción del vehículo:

Se hace una suma ponderada de ambos ángulos de conducción para usar tanto la detección de carriles como el modelo. Le dimos un peso del 80% al modelo por un 20% al de Hough. Estos valores se podrían llegar a editar en base a resultados obtenidos y a más iteraciones.

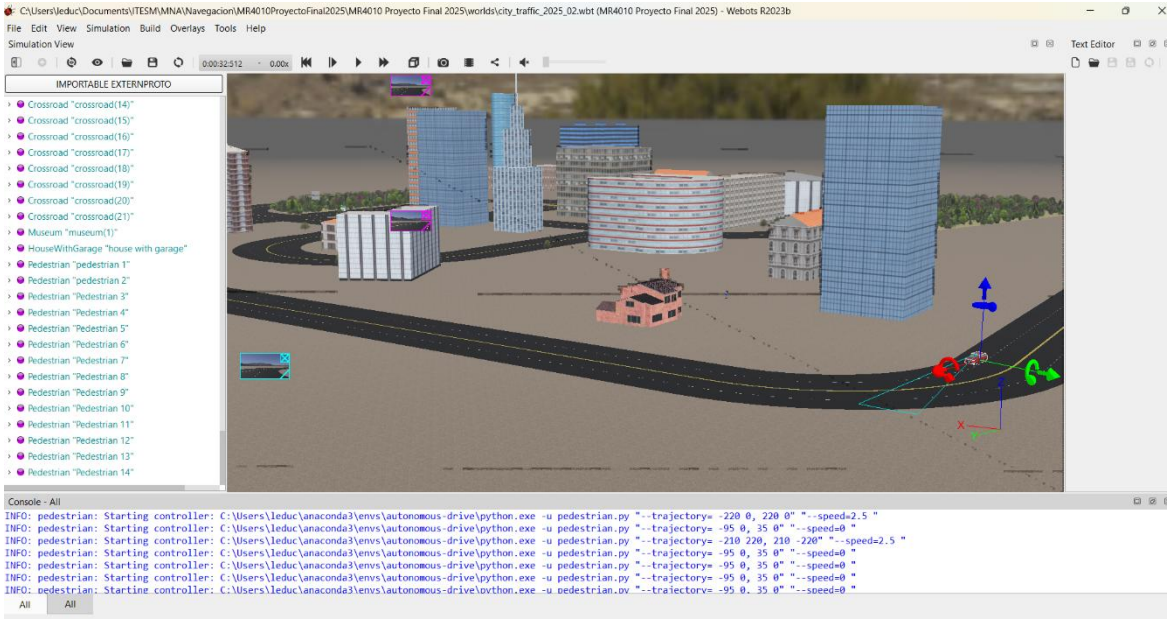
```python
#Se configura la excepción en caso de que regrese información de que no encontró una línea.
try:
    anguloH=set_steering_angle_from_hough_lines(lines)
    angulo_prom=(0.2*anguloH+0.8*pred_angle)
except:
    angulo_prom=0
    anguloH=0
info_overlay = draw_info_overlay(line_overlay, lidar_dist, angulo_prom, speed_drive)
display_image_test(display_Parameters, info_overlay)

#update angle and speed para la conducción del vehículo de forma autónoma.
driver.setSteeringAngle(angulo_prom)
driver.setCruisingSpeed(speed_drive)
```

4. Resultados:

4.1 Se habilitó la interface con SUMO para generar 100 automóviles de tráfico urbano.

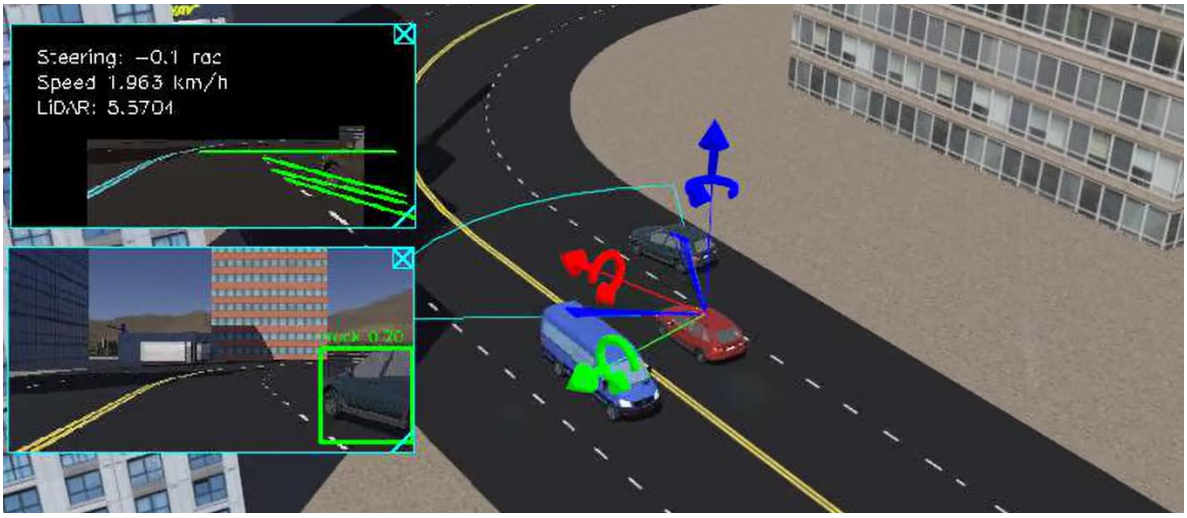
4.2 Al menos 10 peatones en el mundo de Webots:



4.3 Se colocaron 3 sensores: 2 cámaras y 1 LIDAR.

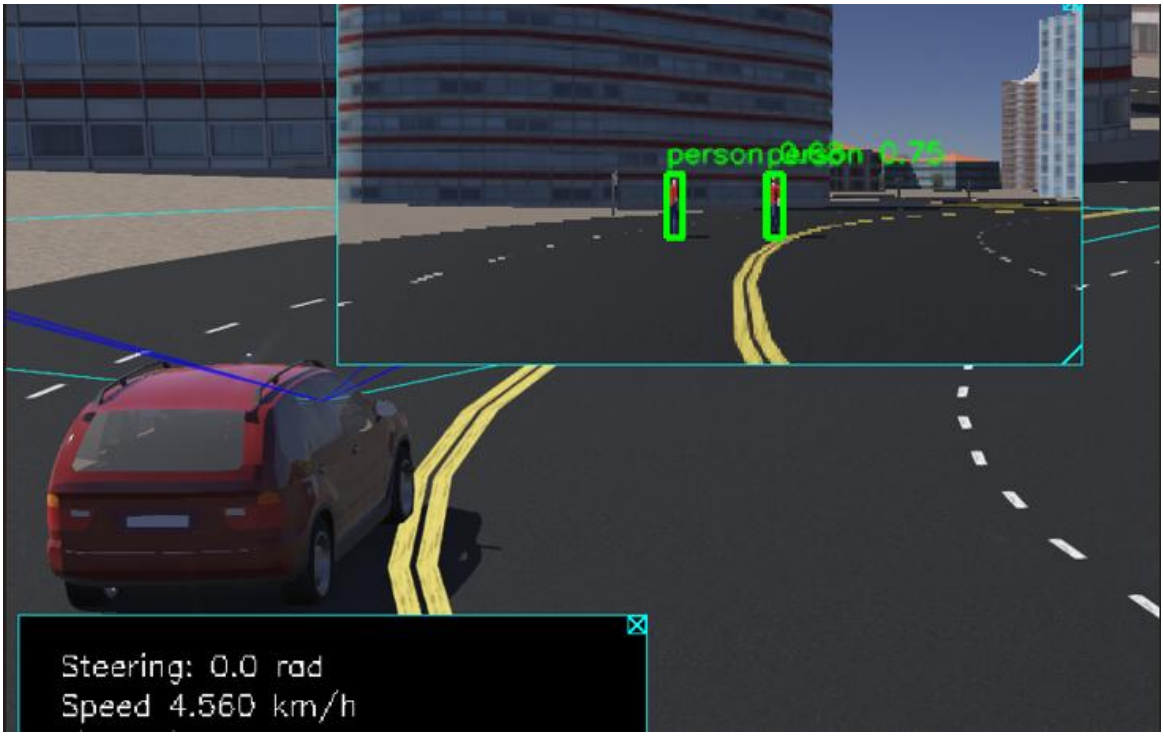
4.5 Optional Rendering y detección de vehículos.

Se habilita el “Optional Rendering” para mostrar la presencia y operación de los sensores, los cuales se visualizan en dos display. Se puede observar que detecta la presencia de objetos como vehículos y reduce la velocidad. Se muestra la detección del modelo YOLO al detectar con un 70% de probabilidad una camioneta (truck) y el LIDAR marca una distancia de 5.57 m con respecto a nuestro vehículo.



4.6 Detección de peatones.

YOLO detecta a las personas y el controlador externo reduce la velocidad.



4.7 Evidencia del recorrido en YouTube.

Enlace al video en plataforma YouTube:

https://www.youtube.com/watch?v=CQvCHIXKTTk

5. Conclusión:

Durante la realización del proyecto encontramos una serie de desafíos: como primer punto fue la puesta en operación del simulador “Simulation Urban Mobility (SUMO)” ya que, de las 4 computadoras en nuestro equipo, sólo en una funcionó para realizar el proyecto.

Segundo, la captura de las imágenes para la generación del conjunto de datos para el entrenamiento del modelo supuso realizar un recorrido en el mundo virtual y crear el archivo con ayuda de python.

Observamos que la velocidad de reproducción del mundo “City_Traffic_2025_02” en Webots se ve muy afectada por la falta de un hardware como GPU dedicado y esto se traduce en un retraso en la predicción en los modelos.

También nos dimos cuenta de la importancia de la navegación autónoma y las múltiples herramientas que se requieren para poder llevarla a cabo de manera eficiente. Con lo que podemos tener un mejor entendimiento de la industria automotriz, pero de igual manera pensar en otras aplicaciones sobre todo en el ámbito de la robótica. Poder entender una tecnología a este nivel nos da una gran ventaja como consumidores y usuarios, pero nos da una gran oportunidad como desarrolladores.

Bibliografía

- Geron, A. (2022). Hands-on machine learning with scikit-learn, keras, and TensorFlow 3e: Concepts, tools, and techniques to build intelligent systems (3a ed.). O’Reilly Media.
- Joshi, P., & Garrido, G. (2018). *OpenCV 3.x with Python by Example* (2nd ed.). Packt Publishing.
- Lee, W.-M. (2019). *Python machine learning*. John Wiley & Sons.
- OpenCV. (n.d.). *Canny Edge Detection*. OpenCV Documentation. https://docs.opencv.org/
- Ranjan, S., & Senthamilarasu, S. (2020). Road sign detection using deep learning. En Applied deep learning and computer vision for self-driving cars (pp. 203–216). Packt Publishing.
- Stallkamp, J., Schlipsing, M., Salmen, J., & Igel, C. (2011). *The German Traffic Sign Recognition Benchmark: A multiclass classification competition*. En *Proceedings of the 2011 International Joint Conference on Neural Networks (IJCNN)*. https://www.researchgate.net/publication/224260296_The_German_Traffic_Sign_Recognition_Benchmark_A_multi-class_classification_competition.
- Webots. (n.d.). *User Guide*. Cyberbotics Ltd. https://cyberbotics.com/doc/