



GRADO EN INGENIERÍA DE ROBÓTICA SOFTWARE

Escuela Técnica Superior de Ingeniería de Telecomunicación

Curso académico 2021-2022

Trabajo fin de grado

Escribe el título del trabajo aquí
con la segunda línea aquí

Tutor: Julio Vega Pérez
Autor: Álvaro Mariscal Ávila



Este trabajo se distribuye bajo los términos de la licencia internacional CC BY-NC-SA International License (Creative Commons AttributionNonCommercial-ShareAlike 4.0). Usted es libre de *(a) compartir*: copiar y redistribuir el material en cualquier medio o formato; y *(b) adaptar*: remezclar, transformar y crear a partir del material. El licenciador no puede revocar estas libertades mientras cumpla con los términos de la licencia:

- *Atribución.* Usted debe dar crédito de manera adecuada, brindar un enlace a la licencia, e indicar si se han realizado cambios. Puede hacerlo en cualquier forma razonable, pero no de forma tal que sugiera que usted o su uso tienen el apoyo de la licenciatante.
- *No comercial.* Usted no puede hacer uso del material con propósitos comerciales.
- *Compartir igual.* Si remezcla, transforma o crea a partir del material, debe distribuir su contribución bajo la misma licencia del original.

Agradecimientos

Unas bonitas palabras...

Quizás un segundo párrafo esté bien. No te olvides de nadie.

Un tercero tampoco viene mal para contar alguna anécdota...

¿Alguien más? Aunque sean *actores* secundarios.

Un quinto párrafo como colofón.

*A alguien especial;
si no, tampoco pasa nada*

Madrid, xx de xxxxxx de 20xx

Tu nombre

Resumen

Escribe aquí el resumen del trabajo. Un primer párrafo para dar contexto sobre la temática que rodea al trabajo.

Un segundo párrafo concretando el contexto del problema abordado.

En el tercer párrafo, comenta cómo has resuelto la problemática descrita en el anterior párrafo.

Por último, en este cuarto párrafo, describe cómo han ido los experimentos.

Acrónimos

AERO *Autonomous Exploration Rover*

AI *Artificial Intelligence*

ANN *Artificial Neural Network*

API *Application Programming Interface*

HRI *Human-Robot Interaction*

AGV *Automated guided vehicle*

AMR *Autonomous mobile robot*

Índice general

1. Introducción	1
1.1. Coches autónomos	1
1.2. AMRs	2
1.3. Visión como sensor principal	3
1.4. Arquitectura	5
1.5. Deep Learning	6
2. Objetivos	8
2.1. Descripción del problema	8
2.2. Requisitos	9
2.3. Metodología	9
2.4. Plan de trabajo	10
3. Plataforma de desarrollo	11
3.1. Hardware	11
3.1.1. <i>NVIDIA Jetson Nano</i>	11
3.1.2. Motores <i>TT</i>	12
3.1.3. Controladora de motores <i>L298N</i>	12
3.1.4. Cámara <i>Xiaomi</i>	13
3.1.5. Batería 10000mAh	14
3.1.6. Chasis	14
3.2. Software	15
3.2.1. <i>Python</i>	15
3.2.2. <i>Blender</i>	15
3.2.3. <i>Gazebo</i>	16
3.2.4. <i>SDF</i>	16
3.2.5. <i>ROS</i>	17
3.2.6. <i>OpenCV</i>	17
3.2.7. <i>YOLO</i>	18

3.2.8. <i>Darknet</i>	19
3.2.9. <i>LabelIMG</i>	20
3.2.10. <i>PyQt</i>	20
3.2.11. <i>JetRacer</i>	21
4. Diseño	23
4.1. Entorno simulado	23
4.1.1. Modelo de la ciudad	23
4.1.2. Modelo del coche autónomo	25
4.1.3. Seguimiento de carril	30
4.1.4. Detección de objetos	33
4.1.5. Interfaz de usuario	36
4.2. Entorno real	37
4.2.1. Objetos	37
4.2.2. Circuito	37
4.3. Redes neuronales en el entorno real	38
4.3.1. Entrenamiento de la red <i>YOLO V3 Tiny</i> para detección de objetos propios	40
4.3.2. Controlador diseñado para el seguimiento de carril a partir de la salida de la red neuronal	44
4.3.3. Transformación del modelo de <i>PyTorch a TensorRT</i>	44
4.4. Diseño del software a partir de un diagrama de clases	46
5. Conclusiones	48
5.1. Conclusiones	48
5.2. Líneas futuras	49
Bibliografía	51

Índice de figuras

1.1.	<i>Tesla AutoPilot</i>	2
1.2.	<i>AMRs Kiva Systems</i> en almacenes de <i>Amazon</i>	3
1.3.	<i>PiCamera</i> usada en la placa <i>Raspberry Pi</i>	4
1.4.	<i>BMW's FIR-based Autoliv Night Vision System</i>	4
1.5.	Imagen de profundidad <i>Kinect</i>	5
1.6.	<i>Kinect</i> desarrollada por <i>Microsoft</i>	5
1.7.	<i>Raspberry Pi 4</i>	6
1.8.	<i>Jetson Nano</i>	6
1.9.	<i>LattePanda Alpha 864s</i>	6
1.10.	<i>AMD Zen</i> como CPU y <i>Navi 23</i> como GPU, usado en <i>Tesla Model S</i> [Ros, 2021]	6
3.1.	<i>NVIDIA Jetson Nano</i>	12
3.2.	Motores <i>TT</i>	12
3.3.	Controladora de motores <i>L298N</i>	13
3.4.	Cámara <i>Xiaomi</i>	13
3.5.	Batería 10000mAh	14
3.6.	Chasis	14
3.7.	Creación de una animación 3D con <i>Blender</i>	15
3.8.	Ciudad simulada en <i>Gazebo</i>	16
3.9.	Robot Spot de Boston Dynamics simulado en <i>Gazebo</i>	17
3.10.	Objetos detectados por <i>YOLO</i> en una carretera	18
3.11.	Arquitectura <i>YOLOv3</i> con 53 capas convolucionales	19
3.12.	Etiquetado de imágenes mediante la biblioteca <i>LabelIMG</i>	20
3.13.	Software de impresión 3D <i>Ultimaker Cura</i>	21
3.14.	Interfaz <i>notebook</i> para ajustar controlador P	22
4.1.	Modelo de la ciudad original y la ciudad modificada en <i>Gazebo</i>	24
4.2.	Semáforo y peatón usando <i>plugins</i> para <i>Gazebo</i>	25

4.3.	Diseño del chasis 2D en <i>FreeCAD</i>	26
4.4.	Diseño de las piezas 3D en <i>FreeCAD</i>	26
4.5.	Modelo 3D estático ensamblado en <i>Blender</i>	27
4.6.	Diagrama del modelo diseñado con <i>Phobos</i>	28
4.7.	Modelo dinámico del robot en <i>Gazebo</i>	30
4.8.	Imagen del <i>dataset</i> donde el ángulo de giro necesario es elevado.	31
4.9.	Valor del error al finalizar el entrenamiento en el entorno simulado.	32
4.10.	Salida de la red neuronal <i>ResNet-18</i> en el simulador <i>Gazebo</i>	33
4.11.	Salida de la red neuronal <i>YOLO V3 Tiny</i> en el simulador <i>Gazebo</i>	35
4.12.	Detección del semáforo mediante transformación a <i>HSV</i> y filtro de color.	35
4.13.	Capturas de la ejecución con dos redes neuronales.	36
4.14.	Interfaz de usuario desarrollada para controlar la simulación.	36
4.15.	Robot con <i>NVIDIA Jetson Nano</i>	37
4.16.	Objetos reales dinámicos y estáticos.	38
4.17.	Circuito inicial construido a partir de pistas Scalextric.	38
4.18.	Circuito con objetos.	39
4.19.	Imagen del <i>dataset</i> utilizado entrenado con luz artificial y error al finalizar el entrenamiento.	39
4.20.	Etiquetado de objetos propios con la herramienta <i>LabelIMG</i>	42
4.21.	Distribución de las clases de objetos del <i>dataset COCO</i>	42
4.22.	Gráfica de entrenamiento mostrando el error de la red <i>YOLO V3 Tiny</i>	44
4.23.	Visión del robot durante la ejecución en el entorno real.	46
4.24.	Diagrama de clases.	46
5.1.	Posibles usos alternativos del software implementado.	49

Listado de códigos

4.1.	Definición de estados y duraciones del semáforo.	24
4.2.	Configuración de <i>waypoints</i> , velocidad y distancia a obstáculos del peatón.	25
4.3.	Carga del <i>plugin Gazebo ROS Control</i>	29
4.4.	Crear cámara simulada en <i>Gazebo</i>	29
4.5.	Definición de los controladores de los <i>joints</i> del robot.	30
4.6.	Obtención de la salida de la red neuronal <i>ResNet-18</i>	33
4.7.	Contenido del mensaje <i>BoundingBox</i>	35
4.8.	Formato de etiquetado de objetos utilizado por <i>COCO</i>	41
4.9.	Formato de etiquetado de objetos utilizado por <i>YOLO</i>	41
4.10.	Contenido del archivo <i>obj.data</i> con las rutas de los archivos necesarios.	43
4.11.	Controlador P realizado para el seguimiento de carril.	45
4.12.	Conversión del modelo para realizar optimización y aumentar el rendimiento.	45

Listado de ecuaciones

Índice de cuadros

4.1.	<i>FPS</i> obtenidos en diferentes placas <i>NVIDIA</i>	34
4.2.	FPS obtenidos en <i>YOLO V3 Tiny</i> con la tarjeta gráfica <i>NVIDIA MX330</i>	34
4.3.	Comparación entre las tarjetas gráficas disponibles para entrenar la red <i>YOLO V3 Tiny</i>	40

Capítulo 1

Introducción

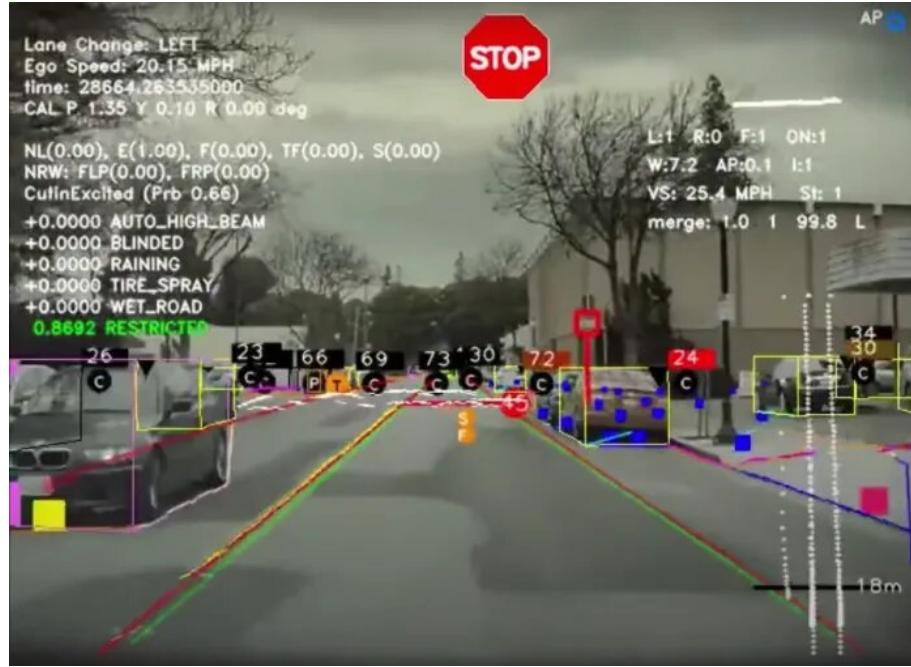
En la actualidad los vehículos autónomos están en auge, cada vez tenemos más ejemplos de tareas típicamente realizadas por humanos que ya, es posible realizar sin un humano al mando. La conducción autónoma tiene el potencial para cambiar la forma en que nos movemos, aportando seguridad y comfort, si bien es cierto que aún no vemos vehículos circulando sin conductor, mucha de la tecnología necesaria para hacerlo ya está presente en los vehículos actuales. La conducción autónoma va mucho más allá de, únicamente los vehículos que transitan las ciudades, es aplicable a muchos otros ámbitos, por ejemplo entornos industriales, de inspección o incluso de exploración donde el vehículo se enfrenta a situaciones impredecibles y ante las que debe saber reaccionar correctamente.

1.1. Coches autónomos

Cuando se nos viene a la cabeza el concepto de vehículo autónomo, se suele relacionar con coches autónomos, es decir, los vehículos que circulan a diario por las ciudades; coches, autobuses, furgonetas pero sin una persona al volante. Todavía no está presente en las ciudades pero en un corto plazo de tiempo lo estará, el principal inconveniente actual es la regulación, a diferencia de, por ejemplo, el entorno de la aviación, donde desde hace décadas el control de la aeronave es automático a excepción de tareas como el despegue, el aterrizaje o situaciones de emergencia.

Actualmente coches de última generación como *Tesla*, ya incorporan un grado de autonomía elevado en determinadas situaciones, pero siempre con un conductor al volante que debe permanecer atento para poder reaccionar.

Atendiendo al estándar *SAE J3016* los niveles de autonomía se pueden dividir en

Figura 1.1: *Tesla AutoPilot.*

cinco:

1. Sin automatización: avisos y asistencia puntualmente
2. Asistencia a la conducción: centrado de carril **o** control de crucero
3. Automatización parcial: centrado de carril **y** control de crucero
4. Automatización condicionada: conducción automática en atascos
5. Automatización elevada: conducción automática en algunas situaciones
6. Automatización completa: conducción automática en cualquier situación

[SAE, 2018]

1.2. AMRs

Los *AMRs* son robots móviles autónomos, capaces de navegar por entornos dinámicos, conviviendo con humanos a su alrededor y sabiendo sobreponerse a situaciones para las que no habían sido programados explícitamente. Son los sucesores de los *AGVs*, vehículos guiados automatizados, este tipo de vehículos requieren una cierta infraestructura dependiendo del tipo de guiado, ya sea filoguiados, a través de pintura o a través de cualquier otra técnica que haga que ese vehículo sólo pueda

funcionar cuando se sabe la infraestructura previa que estará presente en el entorno de trabajo.

Además, presentan muchas dificultades para relacionarse con obstáculos o humanos, donde ante un cambio pequeño del entorno, el robot se detendrá por seguridad. A diferencia de estos, los *AMRs*, son capaces de realizar multitud de tareas en entornos donde la infraestructura necesaria es casi nula, quizás tengan una serie de requisitos en cuanto a conectividad, pero con esa excepción, son robots que pueden ser diseñados para navegar por cualquier tipo de ambiente.

Un ejemplo muy representativo de *AMRs*, es *Kiva Systems*, empresa comprada por *Amazon* para automatizar sus almacenes en tareas de logística a nivel interno, maximizando la productividad y el almacenamiento, tanto en profundidad como en altura y minimizando el coste en personal.



Figura 1.2: *AMRs Kiva Systems* en almacenes de *Amazon*.

1.3. Visión como sensor principal

Gracias a un sensor como la cámara podemos obtener una información muy completa del entorno que rodea al robot y con ello poder actuar en consecuencia. La cámara se presenta como el sensor más interesante que puede equipar un robot, cuentan con un tamaño y peso muy reducido, como es el caso de la 1.3, además de un coste muy reducido. Sin embargo, presenta algunas dificultades cuando nos disponemos a tratar la imagen recibida; en cuanto a potencia del dispositivo, para procesar una imagen, y dependiendo de su resolución, es necesario un mínimo de requerimientos técnicos a nivel de *hardware*, para poder conseguir un procesamiento con un nivel adecuado de

fps, por otra parte, la imagen recibida deberá haber sido captada en un entorno con buenas condiciones lumínicas. Para ello existen diferentes tipos de sensores *EO/IR*, como por ejemplo, cámaras de visión nocturna, donde el espectro utilizado es *FIR*, con ello se consigue resaltar en la imagen lo que realmente es necesario.



Figura 1.3: *PiCamera* usada en la placa *Raspberry Pi*.

Un ejemplo es *BMW's FIR-based Autoliv Night Vision System* [Raiciu, 2009], trabaja con imágenes de 320 por 240 píxeles y cuenta con un rango de 300 metros.



Figura 1.4: *BMW's FIR-based Autoliv Night Vision System*.

Cuando se realiza el procesamiento de una imagen, es interesante conocer la distancia, por ejemplo, a la que se encuentra un objeto en concreto, para ello existen soluciones para poder conocer dicha información con una **única** cámara tradicional, es el caso de *cite Open Vision System for Low-Cost Robotics Education* o [Dal, 2021], sin embargo, estos algoritmos se basan en la suposición de que todos los objetos se encuentran situados en un plano imaginario situado en el suelo. Para eliminar esta

restricción, y poder conocer distancias de objetos que no se encuentran situados en el suelo, surgen las *cámaras RGBD*, en las que cada píxel de la imagen proporciona, además del color RGB, una tercera componente llamada *depth*. Una de las primeras cámaras *RGBD* comerciales es la *Kinect* desarrollada por *Microsoft* para su consola *Xbox 360*.



Figura 1.5: Imagen de profundidad *Kinect*.

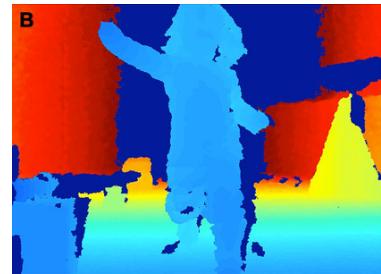


Figura 1.6: *Kinect* desarrollada por *Microsoft*.

La visión artificial ofrece multitud de posibilidades, no solo en robótica, tiene una gran de aplicaciones en campos tan dispares como la medicina, la realidad aumentada, el procesamiento de señales o la agricultura.

1.4. Arquitectura

La decisión de qué arquitectura utilizar en un robot es determinante. Es necesario evaluar multitud de factores; consumo de energía, potencia requerida, tamaño y peso, sistema operativo a utilizar, posibilidad de recibir respuesta en *real time*, precio etc...

Existen multitud de arquitecturas utilizadas en proyectos robóticos *x86*, *x86_64*, *ARMv6*, *ARMv7* o *AArch64*. Pero, actualmente hay dos arquitecturas predominantes *x86_64* y *AArch64*, ambas de *64 bits*, ya que los *32 bits* han quedado desfasados para la gran mayoría de aplicaciones, además, los nuevos sistemas operativos comienzan a no soportarlos.

[J.Pomeyrol, 2019]

Estas dos arquitecturas tienen grandes diferencias en cuanto al diseño de los procesadores y las instrucciones que utilizan:

- Reduced Instruction Set Computer (*RISC*)
- Complex Instruction Set Computer (*CISC*)

El conjunto de instrucciones utilizado por *AArch64* es *CISC*, este conjunto se compone de gran cantidad de instrucciones y muchas de ellas complejas para realizar tareas que, el conjunto *RISC*, puede realizar con varias instrucciones. Este último conjunto es utilizado por la arquitectura *x86_64*.

Ejemplos representativos de *AArch64*:

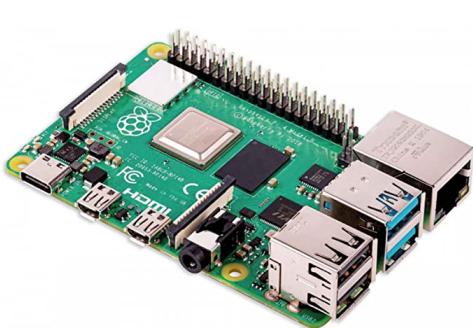


Figura 1.7: *Raspberry Pi 4*.



Figura 1.8: *Jetson Nano*.

Ejemplos representativos de *x86_64*:



Figura 1.9: *LattePanda Alpha 864s*.



Figura 1.10: *AMD Zen* como CPU y *Navi 23* como GPU, usado en Tesla Model S [Ros, 2021].

1.5. Deep Learning

El *Deep Learning* se basa en redes neuronales que parten del *Machine Learning*, que a su vez surge de la *Inteligencia Artificial (IA)*. Sus inicios se remontan al año 1979 cuando *Kunihiko Fukushima* desarrolló una red neuronal de entre 5 y 6 capas llamada *neocognitrón*[Fukushima, 1979], con el objetivo de reconocer caracteres japoneses.

Este tipo de redes neuronales tiene multitud de aplicaciones, pero todas comparten grandes cantidades de datos, conocidos como *datasets*, en cualquier formato; vídeo, imagen, sonido. Algunas de ellas son: clasificación de objetos, procesamiento natural del lenguaje, *Big Data*, análisis médico, conversión de imágenes en blanco y negro a color etc...

En los siguientes capítulos trataremos los objetivos a cumplir ...

Mapa

Capítulo 2

Objetivos

En este capítulo se detallan los objetivos del trabajo realizado, así como, los requisitos que este ha de cumplir, la metodología utilizada y el plan de trabajo seguido para completarlo.

2.1. Descripción del problema

El objetivo del trabajo es implementar un coche autónomo bajo una plataforma de bajo coste y reducido tamaño capaz de circular por un circuito o carretera en un entorno dinámico interactuando con objetos propios de una ciudad, como semáforos, señales de stop o peatones. Para ello se ha dividido el objetivo en general en dos subobjetivos:

- (a) Entorno simulado: utilizando el entorno de simulación *Gazebo* se desarrollará el problema anteriormente descrito con la finalidad de, posteriormente, realizar lo mismo en un entorno real.
- (b) Entorno real: utilizando un robot real diseñado a partir de la placa *NVIDIA Jetson Nano*, se desarrollará el problema anteriormente descrito sobre un circuito construido a partir de pistas de *Scalextric*¹.

Como se dijo anteriormente, en ambos entornos, se ha de completar dos objetivos:

- (a) Seguimiento de carril: el coche autónomo tendrá que ser capaz de realizar un seguimiento del carril utilizando una red neuronal que indicará el centro del carril al que el robot deberá ceñirse.

¹<https://scalextric.es/>

- (b) Detección de objetos: mientras el robot realiza el seguimiento del carril debe interactuar con objetos del entorno, esto es, reaccionar cuando ve una señal de stop o un semáforo y rojo y detenerse apropiadamente. Esta detección deberá ser realizada, preferiblemente, en tiempo real.

2.2. Requisitos

El trabajo tendrá una serie de requisitos que deberán ser respetados:

1. El sistema operativo utilizado, para ambos entornos, será *GNU/Linux*, concretamente la distribución *Ubuntu 20.04 LTS* ya que cuenta con soporte para múltiples arquitecturas y proporciona un gran rendimiento.
2. El entorno simulado requerirá la presencia de una tarjeta gráfica dedicada ya que es muy recomendable para trabajar con redes neuronales; concretamente de la marca *NVIDIA*, ya que se utilizará la plataforma *CUDA*.
3. El entorno real requerirá un robot con la placa de desarrollo *NVIDIA Jetson Nano* debido a ser una de las placas con *GPU* más económicas.
4. El lenguaje de programación utilizado será *Python* debido a las librerías utilizadas, que se detallarán en el próximo capítulo.

2.3. Metodología

Partiendo de los requisitos y objetivos previamente descritos, se procedió a evaluar el hardware necesario; a continuación, se realizó un análisis de diversas bibliotecas de código con el objetivo de seleccionar las que fuesen compatibles y tuviesen un mejor rendimiento en la plataforma de hardware elegida. El siguiente paso fue el diseño del software necesario y cómo integrarlo con las bibliotecas escogidas. Por último se realizaron pruebas periódicas tanto en simulador como en un entorno real, con el objetivo de ir afinando el software para conseguir el resultado final.

2.4. Plan de trabajo

El plan de trabajo se ha basado en reuniones semanales o quincenales con el tutor, dependiendo de la carga de trabajo, en las que se iban fijando objetivos específicos y fijando la estrategia para poder completar el proyecto.

Todo el trabajo realizado se ha ido subiendo a un repositorio de trabajo en *GitHub*². De esta forma se podía analizar de forma muy rápida los cambios realizados en el código. También se ha ido desarrollando una *Wiki*³ en *GitHub* a modo de bitácora en la que se iban detallando los avances del proyecto, así como, los problemas y las limitaciones que iban surgiendo a medida que se cumplían los objetivos.

²<https://github.com/jmvega/tfg-amariscal>

³<https://github.com/jmvega/tfg-amariscal/wiki>

Capítulo 3

Plataforma de desarrollo

En este capítulo, se explica el hardware y software elegido para desarrollar el trabajo y los motivos de dicha elección.

3.1. Hardware

3.1.1. *NVIDIA Jetson Nano*

La placa de desarrollo *NVIDIA Jetson Nano*¹ es una plataforma de bajo coste con grandes capacidades computacionales para implementar técnicas de inteligencia artificial gracias a su *GPU* dedicada *NVIDIA Maxwell*² con 128 *NVIDIA CUDA cores*³, además dispone de una CPU *Quad-core* basada en la arquitectura Aarch64 lo que permite ejecutar *GNU/Linux* sin dificultades y ser compatible con numerosas bibliotecas de código. La placa en cuestión, dispone además de pines *GPIO* lo que permite de forma muy sencilla conectar todo tipo de sensores y actuadores.

Los requisitos en lo que a alimentación se refiere no son excesivos, requiere un mínimo de 5 voltios (V) y 3 amperios (A), lo que permite que una simple *powerbank* de reducido tamaño sea capaz de alimentar la placa, si bien es cierto que la batería debe poder ofrecer tres amperios de forma estable, y no solo como intensidad pico. Existen numerosos proyectos en los que esta placa está presente, tales como *JetBot*⁴ o *JetRacer*, que se detallará en la Sección 3.2.11.

¹<https://developer.nvidia.com/embedded/jetson-nano>

²<https://developer.nvidia.com/maxwell-compute-architecture>

³<https://developer.nvidia.com/cuda-gpus>

⁴<https://github.com/NVIDIA-AI-IOT/jetbot>



Figura 3.1: *NVIDIA Jetson Nano*.

3.1.2. Motores *TT*

Se trata de unos motores⁵ de corriente continua con reductora utilizados en la multitud de proyectos de robótica de muy bajo coste⁶⁷. La tensión de alimentación tiene un rango de 3 a 6 voltios y la velocidad mínima en vacío tiene un rango de 90 a 200 revoluciones por minuto (RPM) dependiendo del voltaje, lo que permite conseguir una velocidad reducida para robots de pequeño tamaño.



Figura 3.2: Motores *TT*.

3.1.3. Controladora de motores *L298N*

Es un módulo⁸ capaz de controlar la dirección y la velocidad de los motores anteriormente citados. La tensión de alimentación es de un mínimo de 6 voltios, lo que hace imposible alimentarla con la placa *NVIDIA Jetson Nano* por lo que es necesario una batería externa. Otra posibilidad para utilizar una única batería sería utilizar la salida de 5 voltios (V) que nos ofrece la controladora, sin embargo, dicha salida nunca ofrecerá los 3 amperios (A) requeridos. Este componente permite invertir el sentido de la corriente lo que proporciona un control para mover los motores en el sentido de las agujas del reloj (CW) y en el sentido contrario a las agujas del reloj (CCW). La

⁵<https://www.verical.com/datasheet/adafruit-brushless-dc-motors-3777-5912007.pdf>

⁶<https://github.com/grimmp/tt-motor-mounting>

⁷<https://github.com/bhabegger/diy-telepresence-robot>

⁸<https://www.luisllamas.es/arduino-motor-corriente-continua-1298n/>

principal limitación de esta placa es que solo permite controlar dos motores, por lo que si se dispone de 4 motores, se podrán conectar a pares dependiendo del comportamiento deseado. El control se realiza a través de la técnica modulación por ancho de pulso, *pulse with modulation (PWM)*⁹, que permite enviar de forma precisa la velocidad deseada a través de una señal digital.

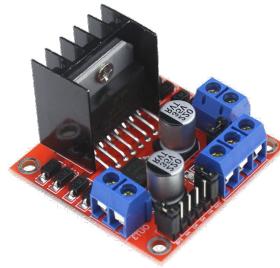


Figura 3.3: Controladora de motores *L298N*.

3.1.4. Cámara *Xiaomi*

Se trata de una cámara USB, disponible en la Figura 3.4, que permite recibir la imagen a través de dicho puerto con una tasa de 30 *frames* por segundo (FPS). Su resolución es 1080p¹⁰ pero permite obtener una imagen de menor resolución a través de su driver.



Figura 3.4: Cámara *Xiaomi*.

⁹<https://circuitdigest.com/tutorial/what-is-pwm-pulse-width-modulation>

¹⁰<https://xiaomiplanets.com/xiaovv-6320s-webcam-5/>

3.1.5. Batería 10000mAh

Su capacidad es de 10000 miliamperios hora (mAh), el voltaje de funcionamiento es 5 voltios (V) y su intensidad teórica es 3 amperios (A) por lo que permite alimentar la placa *NVIDIA Jetson Nano*.



Figura 3.5: Batería 10000mAh.

3.1.6. Chasis

Se trata de un chasis de bajo coste muy común en proyectos relacionados con Arduino. Está representado en la Figura 3.6. Dispone de soportes para los Motores TT, lo que permite ensamblarlos de forma muy sencilla. El tamaño es suficiente para alojar todos los componentes elegidos utilizando los agujeros predefinidos en el chasis.



Figura 3.6: Chasis.

3.2. Software

3.2.1. Python

Python es un lenguaje de programación de código abierto, interpretado, orientado a objetos y de alto nivel. En la actualidad es el lenguaje más usado a nivel mundial¹¹. Está considerado como un lenguaje fácil de aprender gracias a su simple sintaxis, esto le ha llevado a crecer mucho en popularidad durante los últimos años. Además, al ser interpretado, no es necesario utilizar un compilador, lo que provoca un desarrollo mucho más rápido. *Python* cuenta con una enorme cantidad de bibliotecas y clientes para utilizar software desarrollado en otros lenguajes. Algunas de ellas se describirán a continuación.

3.2.2. Blender

*Blender*¹² es una plataforma de código abierto dedicada a la creación, simulación, renderizado y animación de modelos 3D con todo tipo de texturas, sombras etc.. Es un desarrollo de la *Blender Foundation* y está escrito principalmente en *C*. Permite crear diseños de robots desde cero de una forma relativamente rápida y con *addons* como *Phobos*, es posible exportar el modelo a *URDF* con el fin de realizar una simulación[von Szadkowski and Reichel, 2020].



Figura 3.7: Creación de una animación 3D con *Blender*.

¹¹<https://pypl.github.io/PYPL.html>

¹²<https://www.blender.org/>

3.2.3. *Gazebo*

*Ignition Gazebo*¹³ es un simulador 3D de código abierto desarrollado por la *Open Source Robotics Foundation (OSRF)*¹⁴, escrito en *C++*, usado principalmente para simular comportamientos con gran precisión y gráficos de alta calidad en los que intervienen robots en un entorno dinámico, un ejemplo de ello se puede observar en la Figura 3.8. Desde Utiliza, por defecto, el motor de físicas *ODE*¹⁵, escrito también en *C++*. Permite simular todo tipo de sensores y actuadores. Además, ofrece integración con *ROS* de forma muy sencilla.



Figura 3.8: Ciudad simulada en *Gazebo*.

3.2.4. *SDF*

El formato *SDF* (*Simulation Description Format*)¹⁶ o *SDFormat* permite describir entornos, objetos dinámicos o robots de una manera sencilla. Está basado en *XML* y escrito en *C++*. El objetivo de este formato es ejecutar comportamientos en un simulador. Originalmente fue desarrollado como parte de *Gazebo* por lo que existen numerosas bibliotecas donde se encuentran modelos o mundos desarrollados en *SDF* para este simulador¹⁷. Cabe destacar también, el formato *SURDF* que, a diferencia de *SDF*, únicamente puede describir un objeto o robot, pero no el mundo en el que vive¹⁸.

¹³<https://github.com/gazebosim/gz-sim>

¹⁴<https://www.openrobotics.org/>

¹⁵<https://bitbucket.org/odedevs/ode/src/master/>

¹⁶<https://github.com/gazebosim/sdformat>

¹⁷<https://github.com/HuyPhamG/simulatedswarm>

¹⁸<https://newscrewdriver.com/2018/07/31/ros-notes-urdf-vs-gazebo-sdf/>



Figura 3.9: Robot Spot de Boston Dynamics simulado en *Gazebo*.

19

3.2.5. *ROS*

*Robot Operating System ROS*²⁰ es un middleware robótico, de código abierto, con multitud de bibliotecas y herramientas desarrollado por la *Open Source Robotics Foundation (OSRF)*²¹, escrito en *C++* y *Python*. Es considerado actualmente el estándar en robótica. Permite desarrollar aplicaciones complejas en las intervienen diversos procesos llamados nodos²², que se comunican entre ellos mediante *topics*²³ y servicios²⁴. Una de las principales ventajas de utilizar un middleware como ROS es la capacidad de abstracción que proporciona, de forma que el usuario únicamente programa sobre una interfaz²⁵ dada, disponible en *C++* y *Python*, sin preocuparse por lo que pasa por debajo.

3.2.6. *OpenCV*

*OpenCV*²⁶ es una librería de visión artificial de código abierto desarrollado por Intel²⁷. Está escrita en *C/C++* y cuenta con soporte para aceleración por GPU basadas

²⁰<https://ros.org/>

²¹<https://www.openrobotics.org/>

²²<http://wiki.ros.org/Nodes>

²³<http://wiki.ros.org/Topics>

²⁴<http://wiki.ros.org/Services>

²⁵<http://wiki.ros.org/Client%20Libraries>

²⁶<https://github.com/opencv/opencv>

²⁷<https://opencv.org/opencv-platinum-membership/>

en *CUDA*²⁸ y *OpenCL*²⁹ y procesamiento de imagen en tiempo real. Es usada en todo tipo de aplicaciones en las que interviene la visión por ordenador, tales como, detección de objetos, realidad aumentada o reconocimiento de gestos. Además, está disponible en multitud de lenguajes de programación; *C++*, *Python*, *Java*, etc...

3.2.7. YOLO

Uno de los algoritmos más populares, capaz de detectar y clasificar objetos provenientes de una imagen es *YOLO* (*You Only Look Once*)[Redmon and Farhadi, 2018]. Sus principales ventajas son una gran precisión y la posibilidad, con el hardware adecuado, de ejecutar en tiempo real. Este algoritmo hace honor a su nombre y, por tanto, solo realiza una *propagación hacia delante* en cada ejecución.



Figura 3.10: Objetos detectados por *YOLO* en una carretera.

Se basa en el uso de redes nueronales convolucionales, *Convolutional neural network*. Se diferencia de una red neuronal tradicional, en que la operación de multiplicación de matrices, se sustituye por una operación matemática llamada convolución, consistente en mezclar dos fuentes de información para producir una tercera, en este caso dos funciones.

²⁸<https://developer.nvidia.com/cuda-zone>

²⁹<https://www.khronos.org/opencl/>

YOLO hace uso, esencialmente, de tres técnicas para conseguir reconocer objetos:

- División de la imagen en celdas: de esta forma se pueden detectar multitud de objetos en una imagen
- Creación de *bounding boxes*: cajas dibujando el contorno del objeto detectado y fijando la probabilidad de que el objeto detectado sea correcto
- Intersección sobre la unión: consiste en seleccionar el *bounding box* con mayor probabilidad cuando hay varios superpuestos

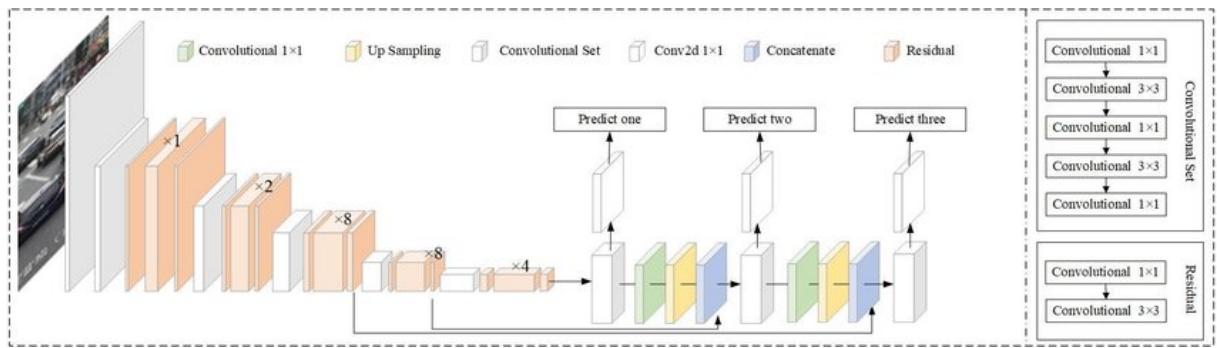


Figura 3.11: Arquitectura *YOLOv3* con 53 capas convolucionales.

Uno de los principales problemas de *YOLO* es la baja de probabilidad de detectar objetos de tamaño reducido, debido principalmente a la baja resolución de la red (está permitido incrementarla pero disminuye su rendimiento) que hace muy difícil su detección.

Además, existen versiones reducidas de este algoritmo como, *Tiny-YOLO* y *Fast YOLO* capaces de ser ejecutados en equipos de bajo coste y reducido tamaño.

3.2.8. *Darknet*

*Darknet*³⁰ es un framework de código abierto que permite ejecutar y entrenar redes neuronales en tiempo real, ya que soporta tanto computación por CPU como GPU. Está escrito en *C* y *CUDA*, gracias a estar escrito en un lenguaje considerado de bajo

³⁰<https://pjreddie.com/darknet/>

nivel, ofrece un rendimiento aceptable en plataformas de bajo coste como *NVIDIA Jetson Nano*.

3.2.9. *LabelIMG*

*LabelIMG*³¹ es una herramienta gráfica de código abierto que permite etiquetar objetos presentes en una imagen a través de la creación de cajas o *bounding boxes* tal y como se observa en la Figura 3.12. Resulta muy útil para el entramiento de redes neuronales en las que se necesita una archivo txt asociado a cada imagen donde se indique las coordenadas de cada objeto.

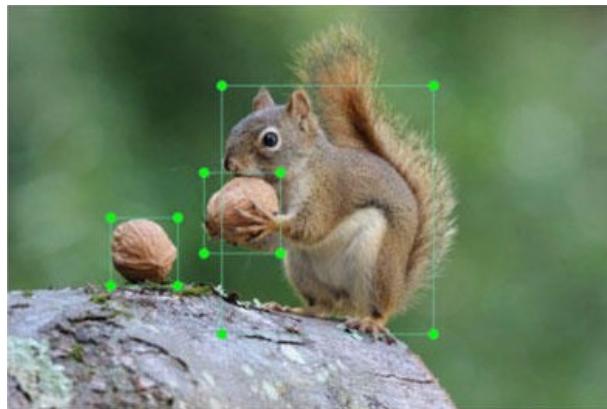


Figura 3.12: Etiquetado de imágenes mediante la biblioteca *LabelIMG*.

3.2.10. *PyQt*

*PyQt*³² es una plataforma de código abierto que permite crear interfaces gráficas (*GUI*) con el framework *Qt* utilizando *Python*, lo que simplifica mucho el desarrollo. Existen multitud de aplicaciones, desde un navegador³³ controlado únicamente con el teclado al estilo *VIM*³⁴, hasta un software de impresión 3D como *Cura*.

³¹<https://github.com/tzutalin/labelImg>

³²<https://pythonpyqt.com/what-is-pyqt/>

³³<https://github.com/qutebrowser/qutebrowser>

³⁴<https://github.com/vim/vim>

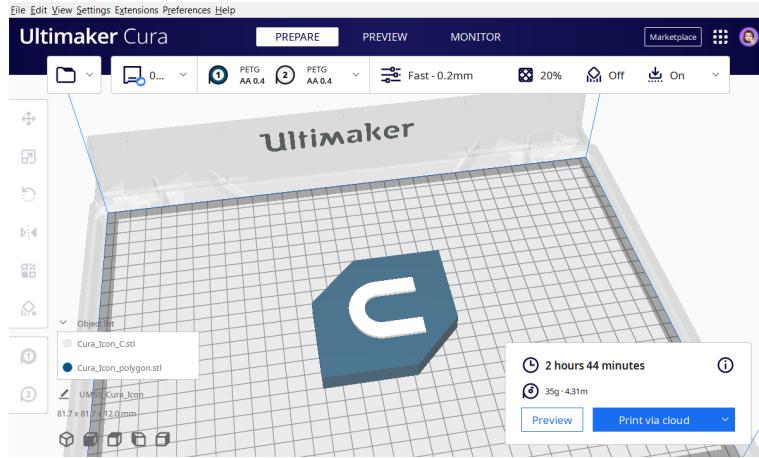


Figura 3.13: Software de impresión 3D *Ultimaker Cura*.

3.2.11. *JetRacer*

La biblioteca *JetRacer*³⁵ permite entrenar una red neuronal para seguir un circuito o una ruta determinada. Utiliza *notebooks* de *Jupyter* para poder reducir la complejidad en el entrenamiento de la red y en el ajuste del controlador tal y como representa la Figura 3.14.

A su vez esta librería utiliza otras tres cruciales para poder implementar su software:

- PyTorch³⁶: es una librería de código abierto con multitud de herramientas para implementar algoritmos de *Deep Learning* basada en *Python*[Today, 2021]. Es un desarrollo de *Facebook's AI Research Lab*. Soporta aceleración por GPU, lo que es esencial para poder ejecutar redes neuronales. Junto a *Tensorflow* y *Keras*, son los tres *frameworks* de referencia en lo que a *Deep Learning* se refiere.
- *PyTorch to TensorRT*³⁷: permite convertir modelos de *PyTorch* a modelos optimizados aprovechando los tensores de las gráficas dedicadas y realizando inferencia utilizando operaciones con *FP16* y *FP32*.
- *Torchvision*³⁸: contiene multitud de *datasets*, modelos preentrenados y algoritmos relacionados con el procesamiento de imagen.

³⁵<https://github.com/NVIDIA-AI-IOT/jetracer>

³⁶<https://github.com/pytorch/pytorch>

³⁷<https://github.com/NVIDIA-AI-IOT/torch2trt>

³⁸<https://github.com/pytorch/vision>

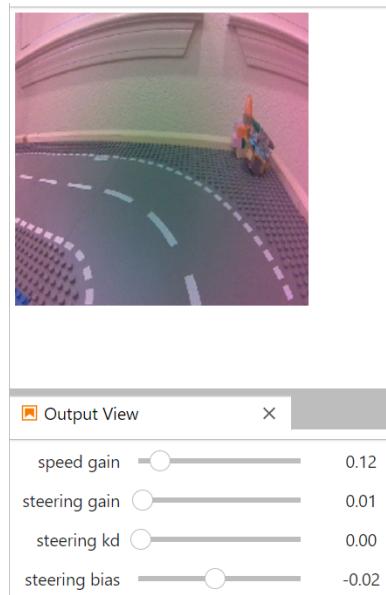


Figura 3.14: Interfaz *notebook* para ajustar controlador P.

Capítulo 4

Diseño

En este capítulo se describe el trabajo realizado y los experimentos llevados a cabo para validar el desarrollo.

El trabajo, como se explicó en el Capítulo 2, se ha dividido en dos fases; primero se realiza una simulación del coche autónomo en una ciudad simplificada, para a continuación, reproducir ese mismo escenario en un entorno real.

4.1. Entorno simulado

Para poder simular el comportamiento deseado en un entorno simulado es necesario disponer de dos elementos; un modelo del coche autónomo y un mundo dinámico con el que el vehículo interactuará, todo ello ejecutado dentro de un simulador, en este caso, como se explicó anteriormente, en *Gazebo*.

4.1.1. Modelo de la ciudad

Existen numerosos mundos diseñados para *Gazebo* disponibles en repositorios de *GitHub*¹². En el caso que se plantea, es necesario una ciudad, por lo que partiendo de una ciudad³ de gran tamaño se reduce su tamaño con el objetivo de, posteriormente, poder reproducir ese escenario en el mundo real. En la Figura 4.1 se observan ambas ciudades.

¹https://github.com/chaolmu/gazebo_models_worlds_collection

²<https://github.com/mlherd/Dataset-of-Gazebo-Worlds-Models-and-Maps>

³https://github.com/chaolmu/gazebo_models_worlds_collection/blob/master/worlds/small_city.world

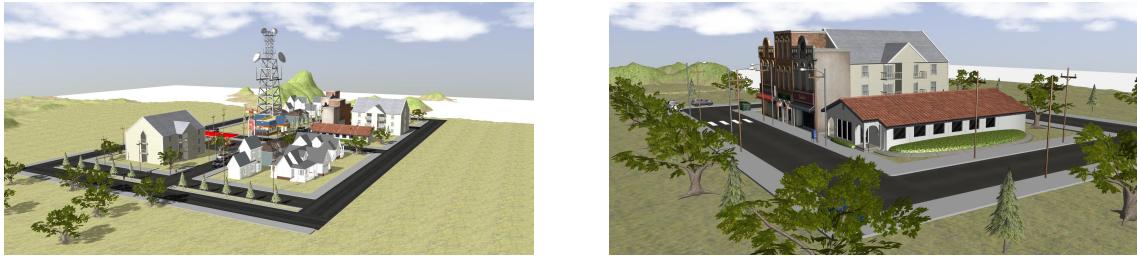


Figura 4.1: Modelo de la ciudad original y la ciudad modificada en *Gazebo*.

El modelo de esta ciudad es estático, es decir, no contiene elementos dinámicos, por lo que es necesario añadir dos tipos de elementos; un semáforo y un peatón que cruce un paso de peatones. Para ello, es necesario añadir dos *plugins* para *Gazebo*. En primer lugar, y usando un modelo *SDF* que integra el *plugin* de cambio de color en el semáforo⁴. Dicho *plugin* tiene asociado un fichero de configuración en formato *YAML*, Código 4.1, en el que se especifica la secuencia de estados del semáforo y su duración en el tiempo.

```
light_sequence:
- { color: "green", duration: 10, flashing: false }
- { color: "yellow", duration: 1, flashing: false }
- { color: "red", duration: 50, flashing: false }
```

Código 4.1: Definición de estados y duraciones del semáforo.

El segundo *plugin* proporciona movimiento a un modelo de un humano llamado actor. Está disponible en otro repositorio⁵ de *GitHub* y permite que el humano se desplace en el mundo de forma realista. El *plugin* citado también permite configuración para especificar la ruta de puntos o *waypoints* que el peatón ha de seguir, así como, su velocidad o la distancia mínima a la que debe situarse respecto a un obstáculo. Dicha configuración se realiza directamente en el fichero *SDF*, como muestra el Código 4.2.

Ambos modelos se sitúan en una paso de peatones, creado a partir de láminas blancas, produciendo el resultado de la Figura 4.2.

⁴https://github.com/robustify/gazebo_traffic_light

⁵<https://github.com/BruceChanJianLe/gazebo-plugin-autonomous-actor>

```

<actor name="actor">
  [...]
  <plugin name="trajectory" filename="libTrajectoryActorPlugin.so">
    <target>
      2.4028 -6.9143 1.1 1.570796 -0.0 3.141593
    </target>
    <target>
      2.4028 6.6816 1.1 1.570796 -0.0 3.141593
    </target>
    <velocity>0.75</velocity>
    <obstacle_margin>1.5</obstacle_margin>
    <obstacle></obstacle>
  </plugin>
</actor>

```

Código 4.2: Configuración de *waypoints*, velocidad y distancia a obstáculos del peatón.



Figura 4.2: Semáforo y peatón usando *plugins* para *Gazebo*.

4.1.2. Modelo del coche autónomo

El primer paso para desarrollar un modelo es diseñar las piezas a utilizar, para ello, como se explicó en el Capítulo 2, se utilizará *FreeCAD*. El segundo paso es ensamblar el robot con *Blender* utilizando el *add-on Phobos* para definir los *links* y *joints* del robot con el objetivo de dotar al robot de movimiento, generando un fichero *URDF*. Por último, es necesario modificar el *URDF* para añadir cámaras y controladores para los motores que serán usados en el simulador *Gazebo*.

Diseño de las piezas en *FreeCAD*

A través de la herramienta *Sketcher*⁶, disponible en *FreeCAD*, es posible realizar un diseño de piezas en 2D con restricciones de horizontalidad, verticalidad, igualdad entre rectas, medidas, ángulos etc y después convertir dicha pieza en 3D proporcionando un

⁶https://wiki.freecadweb.org/Sketcher_Workbench

volumen a la pieza. Utilizando dicha herramienta se diseña el chasis y las ruedas del vehículo tomando las medidas reales, lo que da como resultado el diseño de las piezas en 2D de la Figura 4.3.

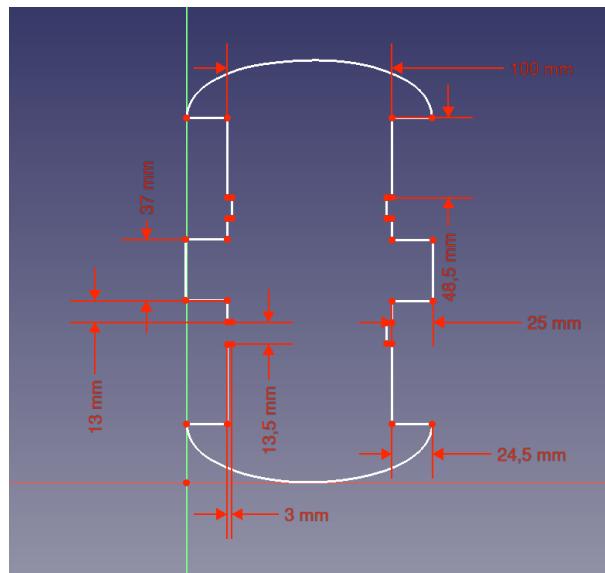


Figura 4.3: Diseño del chasis 2D en *FreeCAD*.

A continuación se proporciona un grosor al chasis de 3mm y a las ruedas de xmm, obteniendo el modelo 3D de la Figura 4.4.

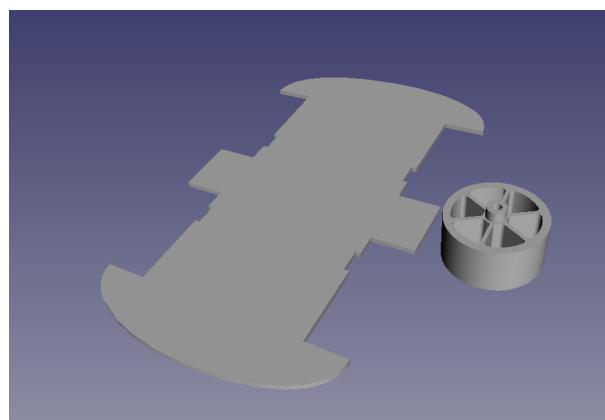


Figura 4.4: Diseño de las piezas 3D en *FreeCAD*.

El siguiente paso es exportar las piezas diseñadas en formato *STL* para utilizarlas en *Blender*.

Ensamblado del robot en *Blender*

Una vez importadas las piezas diseñadas en *FreeCAD*, se crean dos unidades del chasis y cuatro ruedas, a continuación se mueven y rotan en el espacio para situarlas correctamente. Con el objetivo de dotar de mayor realismo al modelo, se importan modelos de los motores y la placa utilizados, disponibles en [y en](#). Y se diseñan dos piezas para emular la batería y la cámara del robot. Con todo ello se conforma el modelo estático del robot, Figura 4.5.

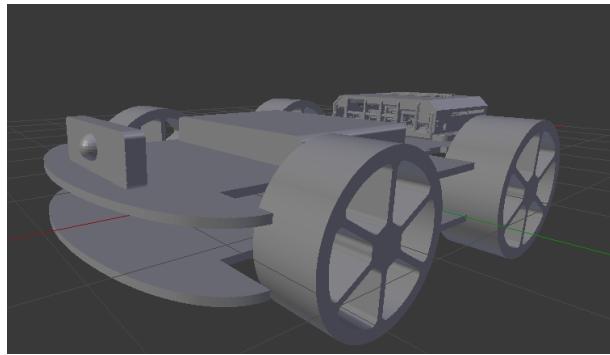


Figura 4.5: Modelo 3D estático ensamblado en *Blender*.

Este modelo es estático, es decir, no es posible simular un movimiento. Para ello es necesario un *add-on* para *Blender* como *Phobos*, que como se explicó anteriormente, permite crear *links* y *joints* y definir una jerarquía entre ellos.

En primer lugar, es necesario definir los elementos visuales que compondrán el modelo, en este caso serán todas las piezas del robot, dichas piezas serán del tipo *mesh*, ya que no son objetos primitivos, como cajas o esferas.

A continuación, se define un *link* principal que será el cuerpo del robot, de este *link* dependerán las cuatro ruedas del vehículo. Con el objetivo de simular colisiones, se crea un modelo de colisión de cada una de las piezas y un modelo inercial, que estará relacionado con el peso simulado de cada pieza.

Por último, se definen los *joints* de cada rueda. A través de esta funcionalidad es posible aplicar fuerza o velocidad a las ruedas del robot. Existen diferentes tipos de

joints en *Phobos*; *fixed*, *revolute*, *continuous*, *prismatic*, en este caso, son necesarios *joints* de tipo *continuous*, ya que las ruedas realizan un giro sin un límite fijado, y no un movimiento transversal.

Una vez definidos todos los elementos del modelo y su jerarquía, en el menú lateral de *Blender* se encontrará el diagrama del modelo, Figura 4.6.

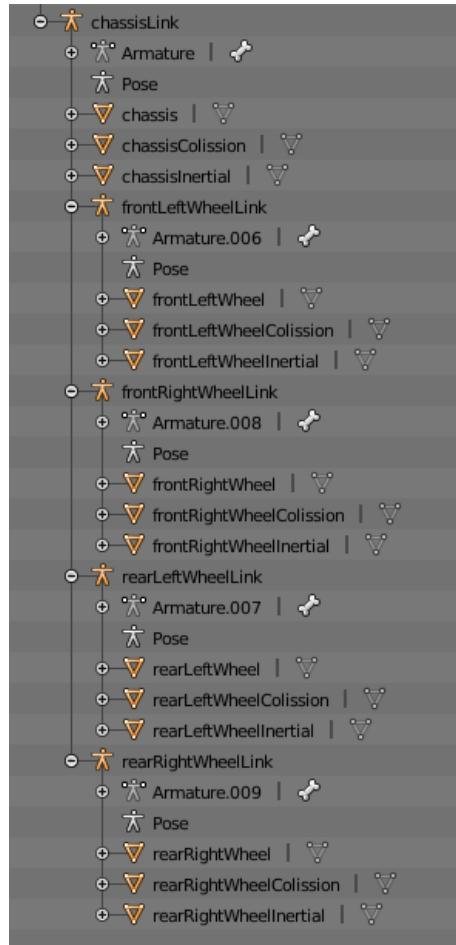


Figura 4.6: Diagrama del modelo diseñado con *Phobos*.

El siguiente paso es exportar el modelo a formato *URDF*, que como se explicó anteriormente, es el aceptado por *Gazebo* para simular modelos de robots. Las mallas, que componen el modelo visual y de colisiones, se exportan en formato *DAE*, y se cargan desde el modelo *URDF*.

Adición de plugins para *Gazebo*

Con el objetivo de mover el robot utilizando el *middleware ROS*, es necesario cargar el *plugin Gazebo ROS Control* mediante el Código 4.3.

```
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <robotNamespace>/autonomous_vehicle</robotNamespace>
  </plugin>
</gazebo>
```

Código 4.3: Carga del *plugin Gazebo ROS Control*.

Otro elemento a añadir en el fichero *URDF*, es la existencia de dos cámaras; una cámara *onboard* y otra situada encima del robot que proporcionará una visión de la ruta realizada. Dichas cámaras simuladas publicarán su imagen en un *topic* de *ROS* con una resolución y un formato fijado mediante el Código 4.4.

```
<gazebo reference="onboardCameraLink">
  <sensor name='cam_sensor' type='camera'>
    [...]
    <camera name='onboardCameraLink'>
      <horizontal_fov>1.570000</horizontal_fov>
      <image>
        <width>320</width>
        <height>240</height>
        <format>R8G8B8</format>
      </image>
      [...]
    </camera>
    [...]
  </sensor>
</gazebo>
```

Código 4.4: Crear cámara simulada en *Gazebo*.

Para mover el robot simulado también será necesario un fichero *YAML*, en el que se especifiquen los controladores de los *joints* y el nombre de los *topics* en los que se publicarán los mensajes mediante el Código 4.5.

Con todo ello, se implementa un *launcher*⁷ que lanza el modelo dinámico del robot

⁷<https://github.com/jmvega/tfg-amariscal/src/launcher.....>

```

autonomous_vehicle:
joint_state_controller:
  type: joint_state_controller/JointStateController
  publish_rate: 10
front_right_wheel_velocity_controller:
  type: velocity_controllers/JointVelocityController
  joint: frontRightWheelJoint
front_left_wheel_velocity_controller:
  type: velocity_controllers/JointVelocityController
  joint: frontLeftWheelJoint
rear_right_wheel_velocity_controller:
  type: velocity_controllers/JointVelocityController
  joint: rearRightWheelJoint
rear_left_wheel_velocity_controller:
  type: velocity_controllers/JointVelocityController
  joint: rearLeftWheelJoint

```

Código 4.5: Definición de los controladores de los *joints* del robot.

y se muestra en la ciudad simulada tal y como se representa en la Figura 4.7.

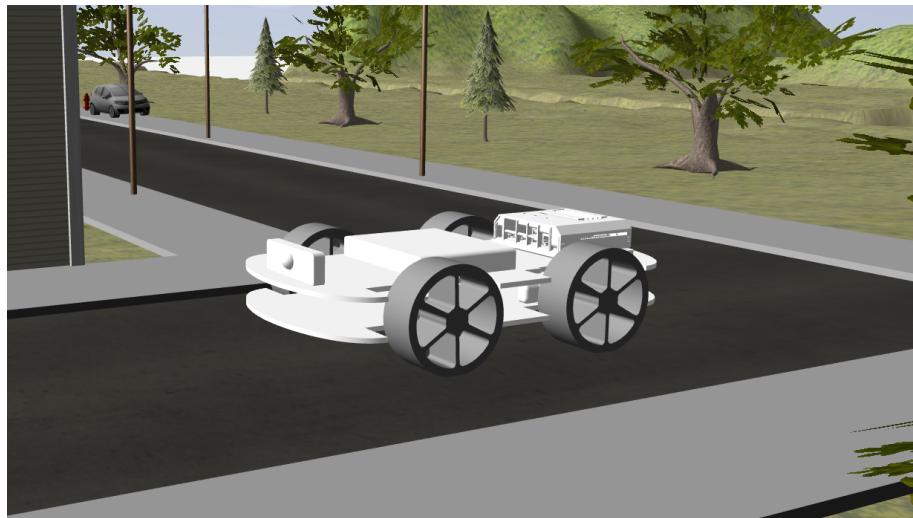


Figura 4.7: Modelo dinámico del robot en *Gazebo*.

4.1.3. Seguimiento de carril

⁸ Como se explicó en el Capítulo 2, la biblioteca elegida para realizar el seguimiento de carril, es *JetRacer* tal y como se expuso en la Sección 3.2.11. Dicha biblioteca se basa

⁸https://github.com/jmvega/tfg-amariscal/src/.....interactive_regression_ros

en la utilización de una red *ResNet-18* preentrenada, consistente en una red neuronal convolucional de 18 capas, sobre la que a su vez, se entrena con un *dataset* de un escaso mínimo número de imágenes. Proporciona diversos *notebooks* de *Jupyter* que hacen más la sencilla la obtención del *dataset*, el entrenamiento y el ajuste del controlador.

El primer paso es la obtención del *dataset* para posteriormente entrenar la red. Para ello, es necesario asociar cada imagen con el centro del carril, definido a partir de una coordenada en x y otra en y. La librería utiliza un *widget* para *Jupyter* llamado *Jupyter Clickable Image Widget*⁹ lo que permite que al hacer click sobre la imagen recibida a partir de un flujo de vídeo, esta se guarde con un nombre atendiendo al siguiente formato: *x_y-identificador_unico.jpg*. Es importante entrenar también con imágenes donde el ángulo del giro del robot sea elevado, tal y como muestra la Figura 4.8 ya que de esa forma el robot sabrá reaccionar ante situaciones difíciles.



Figura 4.8: Imagen del *dataset* donde el ángulo de giro necesario es elevado.

Para mover el robot dentro del simulador es necesario utilizar los controladores de los motores anteriormente desarrollados y para mayor comodidad, se realizó un control manual a través de una interfaz con *PyQT* que se detallará en la Sección 4.1.5.

Una vez obtenido el *dataset*, se descarga el modelo preentrenado *ResNet-18* utilizando la librería *Torchvision* y se realiza una conversión para aprovechar los núcleos de la tarjeta gráfica mediante la plataforma *CUDA* a través de la función `to(device)` donde `device` es `torch.device('cuda')`.

El siguiente paso es el entrenamiento de la red. Para ello, en cada *epoch* se calcula

⁹https://github.com/jaybdub/jupyter_clickable_image_widget

el error cuadrático medio, *mean squared error (MSE loss)* entre una de las imágenes del *dataset* y la salida actual de la red. Una vez calculado este error se realiza la propagación hacia detrás, *backpropagation*, consistente en la propagación de los errores desde la capa de salida hasta la primera capa. Además se realiza una optimización de los pesos a través del algoritmo *Adam* [Kingma and Ba, 2014], el cual es un método de optimización estocástica basado en el cálculo de las tasas individuales adaptables de entrenamiento, *individual adaptive learning rates* a partir de estimaciones del primer y segundo momento del gradiente.

Para realizar el entrenamiento hay que fijar un número de *epochs* que será el momento en el que el entrenamiento finalizará. En cada *epoch* el error va bajando hasta llegar a un valor cercano a 0, dependiendo del número de imágenes del *dataset* el tiempo invertido en realizar una *epoch* será mayor o menor. En el caso del entrenamiento realizado, en 10 *epochs* el error alcanzó el valor de 0.0117 tal y como representa la Figura 4.9.

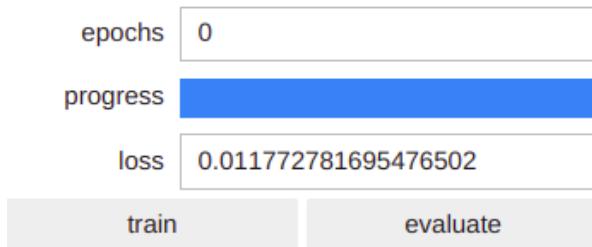


Figura 4.9: Valor del error al finalizar el entrenamiento en el entorno simulado.

Tanto el entrenamiento como la ejecución de la red neuronal recibe imágenes de una baja resolución, 240 píxeles, ya que está diseñada para funcionar en placas de bajas prestaciones. Tras realizar el entrenamiento se genera un modelo *.pth*. Para poder obtener la salida de la red es necesario pasar como argumento al modelo, el resultado de la función `preprocess(image)`, que devuelve un tensor a partir de una imagen en formato `numpy.ndarray`, tal y como se observa en el Código 4.6. Tras obtener la salida es posible representar el centro del carril con el resultado de la Figura 4.10.

En el caso del simulador y al ser una plataforma de pruebas, el objetivo principal es entrenar la red y probar su efectividad en un entorno simulado, y no desarrollar

```
image = camera.read()
preprocessed = preprocess(image)
output = model(preprocessed).detach().cpu().numpy().flatten()
```

Código 4.6: Obtención de la salida de la red neuronal *ResNet-18*.

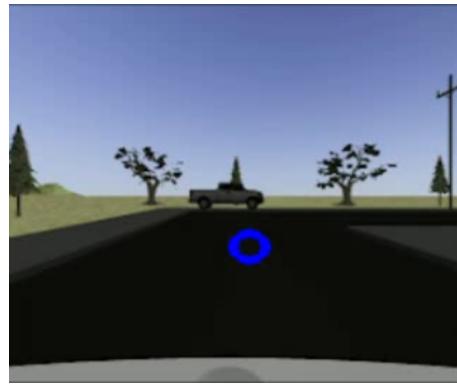


Figura 4.10: Salida de la red neuronal *ResNet-18* en el simulador *Gazebo*.

un controlador a partir de la salida de la red. Por lo que el seguimiento del carril se realizará mediante un autómata que recorrerá la ciudad de forma que se pueda observar el comportamiento de la red neuronal.

4.1.4. Detección de objetos

La detección de objetos se ha llevado a cabo mediante una red neuronal convolucional llamada *YOLO V3 Tiny* debido a su gran rendimiento en placas de reducida potencia. La versión *Tiny-YOLO* reduce el número de capas y precisión de detección pero aumenta la tasa de *FPS* por lo que es muy conveniente su uso en dispositivos como *NVIDIA Jetson Nano*. Existen distintas versiones de *Tiny-YOLO*. La tercera versión cuenta con 15 capas convolucionales¹⁰ a diferencia de la cuarta que cuenta con 29 capas¹¹. En el Cuadro 4.1 se puede observar una comparación entre ambas versiones de la red con una resolución de 416 píxeles, extraída de la siguiente fuente[Ayoub and Schneider-Kamp, 2021], que arroja como resultado un mejor rendimiento en la tercera versión, si bien es cierto que atendiendo al artículo de *YOLO V4* [Bochkovskiy et al., 2020] la precisión de la red ha sido incrementada en un 10% en decremento de la tasa de *FPS*.

¹⁰<https://github.com/pjreddie/darknet/blob/master/cfg/yolov3-tiny.cfg>

¹¹<https://github.com/AlexeyAB/darknet/blob/master/cfg/yolov4-tiny.cfg>

Placa	YOLO V3 Tiny	YOLO V4 Tiny
NVIDIA Jetson Nano	15	7.8
NVIDIA Jetson TX2	19	11.5
NVIDIA AGX Xavier	32	22

Cuadro 4.1: *FPS* obtenidos en diferentes placas *NVIDIA*

La ejecución de la simulación se ha realizado utilizando una tarjeta gráfica *NVIDIA MX330*. Se trata de una tarjeta de gama baja y consumo reducido diseñada para portátiles. En cuanto a la decisión de la resolución de la imagen con la que trabaja la red neuronal se ha realizado una comparación disponible en el Cuadro 4.2.

Resolución	FPS
416 x 416	41.1
608 x 608	21.6
832 x 832	12.3

Cuadro 4.2: FPS obtenidos en *YOLO V3 Tiny* con la tarjeta gráfica *NVIDIA MX330*

Se ha utilizado la red *YOLO V3 Tiny* sobre *Darknet ROS*¹². Se trata de un paquete que implementa *Darknet* utilizando el *framework ROS*. Su funcionamiento se basa en la recepción de la imagen a través de un *topic* fijado en el fichero de configuración *ros.yaml* y la publicación de objetos detectados mediante *bounding boxes* en un *topic* llamado */darknet_ros/bounding_boxes* en el que se publican mensajes con los datos del Código 4.7. De esa forma se puede extraer las coordenadas del *bounding box*, el objeto detectado y su probabilidad. Existe un fichero *YAML* en el que se especifican los objetos a detectar y en el que es necesario especificar límite o *threshold* debajo del cual el objeto detectado será descartado. En el caso del entorno simulado los colores son fijos y los objetos son muy genéricos por lo que el *threshold* se pudo subir hasta 0.8. De esta forma se evitan falsos positivos y se consigue una detección mucho más fiable.

De esta forma se pueden detectar objetos como semáforos, personas o coches tal y como se puede observar en la Figura 4.11.

En el caso de la detección del semáforo es necesario tratar la imagen para detectar el color. Para ello, se extrae la imagen del *bounding box* mediante sus coordenadas,

¹²https://github.com/leggedrobotics/darknet_ros

```

float64 probability
int64 xmin
int64 ymin
int64 xmax
int64 ymax
int16 id
string Class

```

Código 4.7: Contenido del mensaje *BoundingBox*.

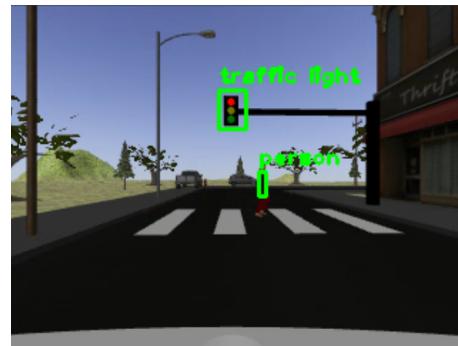


Figura 4.11: Salida de la red neuronal *YOLO V3 Tiny* en el simulador *Gazebo*.

contenidas en el mensaje tal y como consta en el Código 4.7, se transforma al espectro *HSV* con el objetivo de posteriormente realizar un filtro de color. Al aplicar la máscara sobre una imagen en *HSV* y no en *RGB* obtenemos una detección de mayor fiabilidad, tal y como se puede observar en la Figura 4.12.

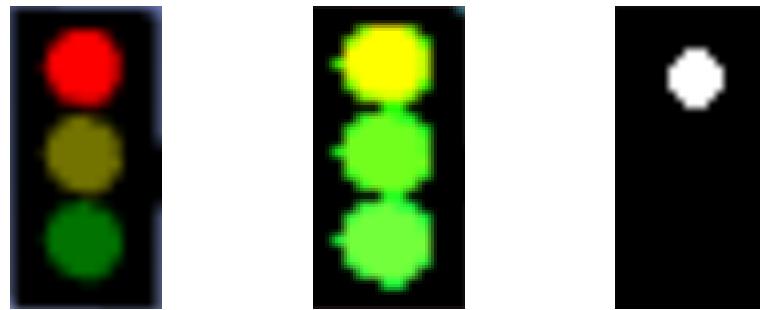


Figura 4.12: Detección del semáforo mediante transformación a *HSV* y filtro de color.

El robot se detiene al detectar el semáforo en rojo y en el momento que cambie a verde reanuda la marcha. En el caso de detectar una señal de stop se parará durante 5 segundos. Todo ello se puede observar en capturas de la ejecución representadas en la Figura 4.13.



Figura 4.13: Capturas de la ejecución con dos redes neuronales.

4.1.5. Interfaz de usuario

Con el objetivo de realizar un control sencillo de la simulación se ha realizado una interfaz gráfica desarrollada con la biblioteca *PyQT*. Dicha interfaz dispone de una cruceta para realizar un control manual del robot. Incluye también un botón para iniciar la simulación y otro para reiniciarla, de modo que, el robot vuelve al punto inicial. Así como un botón para activar las dos cámaras. Dispone de una cámara alojada en el propio robot llamada *onboard* y otra que se desplaza cuando el robot avanza de modo que aporta una visión del recorrido que realiza el robot. La interfaz descrita se puede observar en la Figura 4.14.

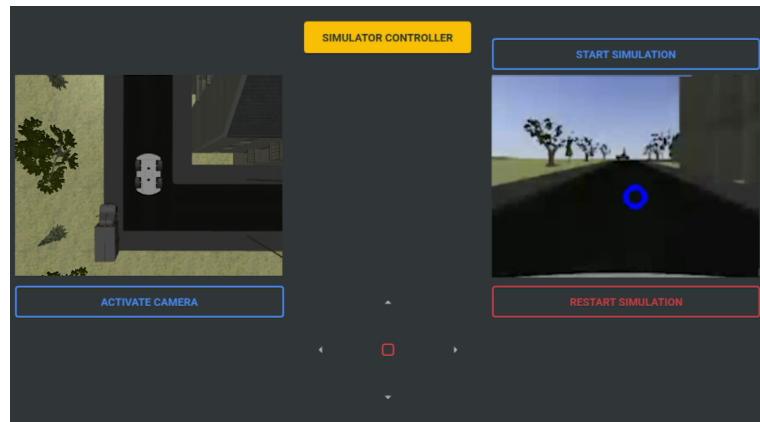


Figura 4.14: Interfaz de usuario desarrollada para controlar la simulación.

4.2. Entorno real

En esta sección se exponen los elementos utilizados para componer un entorno real en el que un robot, representado en la Figura 4.15, equipado con una cámara como sensor principal circula reaccionando ante objetos estáticos y dinámicos. Así como las particularidades del sistema a la hora de realizar la transición desde el entorno simulado al real.

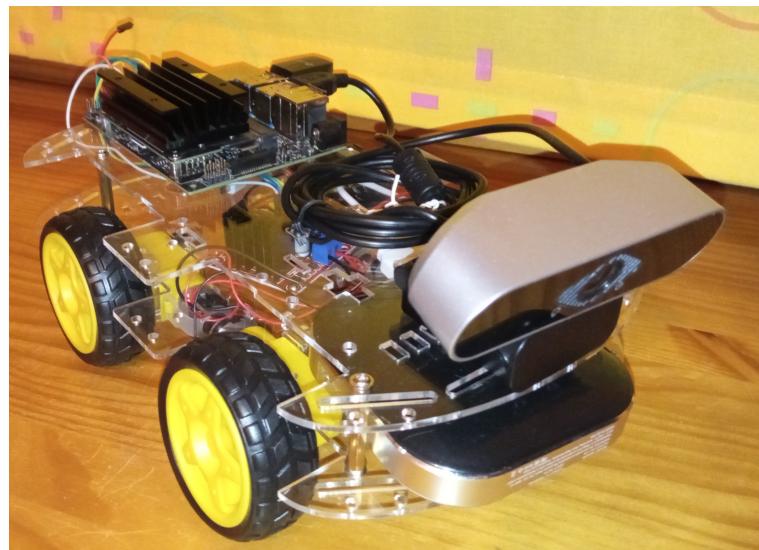


Figura 4.15: Robot con *NVIDIA Jetson Nano*.

4.2.1. Objetos

El entorno real se compone de diversos objetos; un circuito elaborado con pistas Scalextric sobre el que se sitúan objetos como un semáforo dinámico que cambia de color en un modo automático cada cierto tiempo, una señal de stop y un peatón. Dichos objetos se pueden observar en la Figura 4.16.

4.2.2. Circuito

En principio se construyó un circuito inicial, representado en la Figura 4.17, en el que se entrenó la red neuronal de seguimiento de carril. Posteriormente se fue agrandando en el espacio introduciendo rectas más largas para poder interactuar mejor



Figura 4.16: Objetos reales dinámicos y estáticos.

con objetos.



Figura 4.17: Circuito inicial construido a partir de pistas Scalextric.

Tras ubicar todos los objetos alrededor del circuito, da como resultado la imagen representada en la Figura 4.18.

4.3. Redes neuronales en el entorno real

El entrenamiento de la red neuronal *ResNet-18* se realiza de igual forma que en el entorno simulado, la principal diferencia a tener en cuenta es la luz ambiente y los colores, que a diferencia del simulador, cambian dependiendo de como incida la luz. Por ello, es interesante comprobar el comportamiento de la red neuronal cuando hay



Figura 4.18: Circuito con objetos.

cambios de luz. Además, existen multitud de diferencias entre una luz artificial y la luz natural, cuando se trabaja con luz natural es necesario tener en cuenta la incidencia que los rayos del sol pueden hacer sobre la lente de la cámara. Por ello se decidió entrenar la red con luz artificial, con imágenes como la que representa la Figura 4.19, obteniendo el valor de 0.0034 como error. La librería *JetRacer* recomienda entrenar la red con un mínimo de 50 imágenes¹³, por lo que se decidió entrenar con 100 imágenes para que el tiempo de entrenamiento no fuese demasiado elevado y no provocara un aumento muy grande de temperatura en la placa *NVIDIA Jetson Nano*, que al no disponer de disipador activo, alcanza temperaturas muy elevadas.



Figura 4.19: Imagen del *dataset* utilizado entrenado con luz artificial y error al finalizar el entrenamiento.

Una vez probada la red neuronal *YOLO V3 Tiny* utilizada en el entorno simulado,

¹³<https://github.com/NVIDIA-AI-IOT/jetracer/blob/master/docs/examples.md>

se comprobó que era necesario bajar demasiado el límite o *threshold* de detección, hasta 0.3, para poder detectar los objetos elegidos. A diferencia del entorno de simulación, en el que dicho límite se fijó a 0.8. Esto provocaba falsos positivos y por tanto una detección poco fiable. Dicha circunstancia puede ser debida, entre otros motivos, a unos objetos reales algo distintos respecto a los que la red fue entrenada o una luz ambiental baja. Por ello se decidió entrenar la red con los objetos utilizados en el entorno real. Para ello, es necesario una tarjeta gráfica potente. Por lo que se realizó una comparativa entre las tarjetas gráficas disponibles para realizar el entrenamiento en el menor tiempo posible. Dicha comparativa está representada en el Cuadro 4.3.

Placa	Arquitectura	C.C*	GPU Cores	RAM
NVIDIA Jetson Nano	Maxwell	5.3	128	4 GB
NVIDIA MX330	Pascal	6.1	384	2 GB
NVIDIA AGX Xavier	Volta	7.2	384 con 48 Tensor Cores	64 GB

C.C*: capacidad de cómputo.

Cuadro 4.3: Comparación entre las tarjetas gráficas disponibles para entrenar la red *YOLO V3 Tiny*.

4.3.1. Entrenamiento de la red *YOLO V3 Tiny* para detección de objetos propios

El objetivo es entrenar la red *YOLO V3 Tiny* con el objetivo de poder detectar objetos propios o mejorar la probabilidad de detección de objetos ya detectados por la red.

Para ello, es pertinente saber qué *dataset* se ha utilizado para generar los pesos actualmente usados. Atendiendo a la página oficial de *YOLO*¹⁴, el *dataset* utilizado es *COCO*. Por lo que será necesario obtener las imágenes y sus ficheros de etiquetado donde se asocian los objetos presentes en las imágenes. Existe un repositorio¹⁵ que permite la extracción, a partir del *dataset* mencionado, de las clases de objetos deseadas.

El formato del archivo de etiquetado de los objetos es muy relevante. El formato por defecto de *COCO* es el contenido en el Código 4.8¹⁶. Y el utilizado en *YOLO* es el contenido en el Código 4.9¹⁷. Por lo que es necesario realizar la conversión de un

¹⁴<https://pjreddie.com/darknet/yolo/>

¹⁵<https://github.com/KaranJagtiani/YOLO-Coco-Dataset-Custom-Classes-Extractor>

¹⁶<https://cocodataset.org/#format-data>

¹⁷<https://github.com/AlexeyAB/darknet#how-to-train-to-detect-your-custom-objects>

formato a otro. Este proceso también lo realiza el programa contenido en el repositorio anteriormente mencionado.

```

annotation{
    "id": int,
    "image_id": int,
    "category_id": int,
    "segmentation": RLE or [polygon],
    "area": float,
    "bbox": [x,y,width,height],
    "iscrowd": 0 or 1,
}

categories[{
    "id": int,
    "name": str,
    "supercategory": str,
}]

```

Código 4.8: Formato de etiquetado de objetos utilizado por *COCO*.

```
<object-class> <x_center> <y_center> <width> <height>
```

Código 4.9: Formato de etiquetado de objetos utilizado por *YOLO*.

Como se mencionó en el Capítulo 2, se ha utilizado la librería *LabelIMG* para realizar el etiquetado de los objetos y generar el archivo en formato compatible con *YOLO*. En la Figura 4.20 se representa el etiquetado de uno de los objetos reales con dicha librería.

El gráfico representado en la Figura 4.21, extraído de la siguiente fuente[Padilla et al., 2021], muestra el número de imágenes de cada tipo de objeto. Se puede observar que el la clase persona está presente en imágenes, seis veces más que cualquier otro tipo de objeto. Esto se debe a que existen multitud de personas con distinto aspecto y por ello es necesario una cantidad muy grande de imágenes para obtener una detección fiable. Esto es determinante a la hora del entrenamiento, ya que al ser tan elevado el número de imágenes, el tiempo de entrenamiento será considerablemente grande.

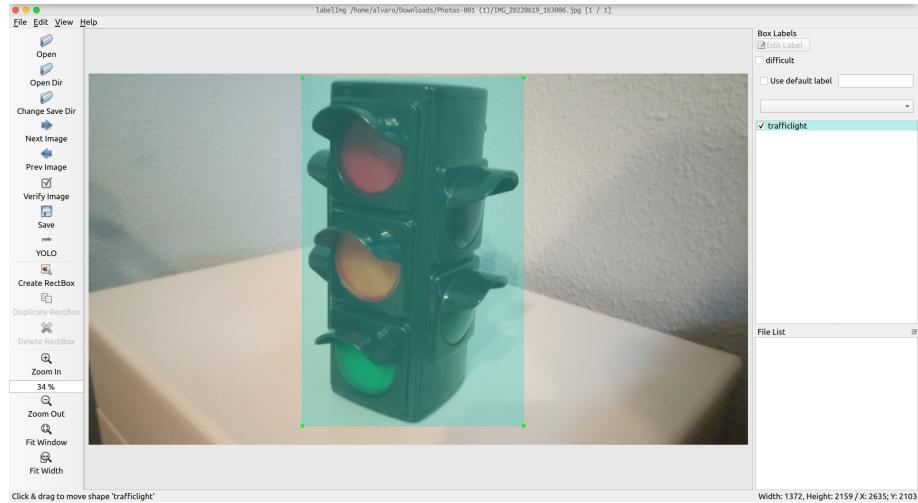


Figura 4.20: Etiquetado de objetos propios con la herramienta *LabelIMG*.

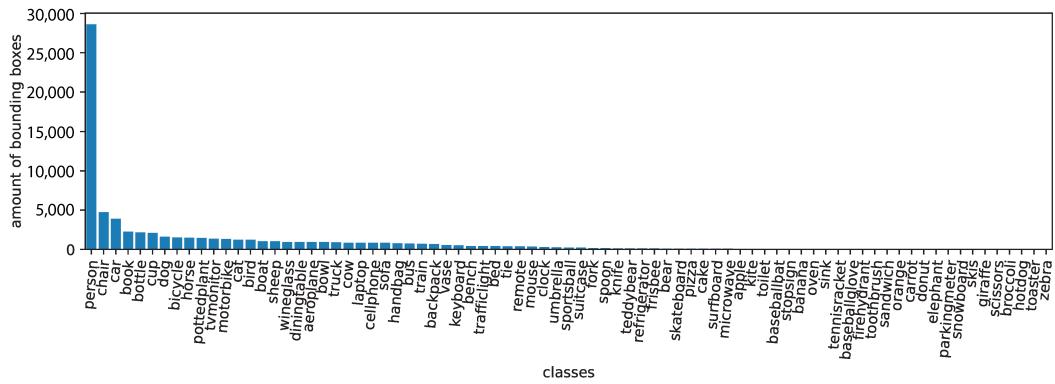


Figura 4.21: Distribución de las clases de objetos del *dataset* COCO.

Para proceder al entrenamiento, además del *dataset* y sus ficheros de etiquetado, es necesario utilizar el *framework* *Darknet*¹⁸ y hacer uso del comando *partial* para obtener los pesos preentrenados sobre los cuales, se entrenará la red con el *dataset* elegido.

También es necesario modificar una serie de archivos entre los que se encuentra el *cfg* que contiene los parámetros de la red. Las principales modificaciones a realizar son las siguientes:

- El número de *batch* deberá ser 64
- El número de *subdivisions* deberá ser 2

¹⁸<https://github.com/pjreddie/darknet>

- El número de *max_batches* deberá ser el número de clases multiplicado por 2000. Y este número no debe ser inferior al número total de imágenes
- El número de *steps* deberá ser fijado al 80 % y al 90 % de *max_batches*
- La anchura y altura de la imagen con la que entrena la red deberá ser como mínimo 416. En el caso de querer detectar objetos muy pequeños dicho parámetro deberá ser modificado a 608 o 832
- En cada parámetro *[yolo]* el número de clases de objetos a detectar deberá ser fijado y el número de *filters* deberá fijarse atendiendo a la siguiente fórmula: $(\text{classes} + 5) \times 3$

Se deberán crear los ficheros *train.txt* y *test.txt* donde deben estar las rutas absolutas a cada fichero de etiquetado. Y otro fichero llamado *obj.data* que contenga las rutas de los archivos anteriormente descritos, dicho contenido está presente en el Código 4.10.

```
classes = 9
train = data/train.txt
valid = data/test.txt
names = data/obj.names
backup = backup/
```

Código 4.10: Contenido del archivo *obj.data* con las rutas de los archivos necesarios.

Otro factor importante es el porcentaje de imágenes con el que no se entrena la red pero sí prueba para comprobar la validez de la misma. Estas imágenes componen el *dataset* de validación y debe estar compuesto por un 20 % de las imágenes totales del *dataset*.

Durante el entrenamiento se irá mostrando una gráfica que indica el progreso, representada en la Figura 4.22. En dicha gráfica se muestra el error de la red, se puede observar que el error final tras terminar el entrenamiento es de 0.663. Atendiendo a la guía¹⁹ utilizada para entrenar la red, el error deberá estar entre 0.05 en el caso de un *dataset* pequeño y simple a 3.0 en el caso de un *dataset* grande y difícil, como es el caso de la clase persona.

¹⁹<https://github.com/AlexeyAB/darknet#how-to-train-to-detect-your-custom-objects>

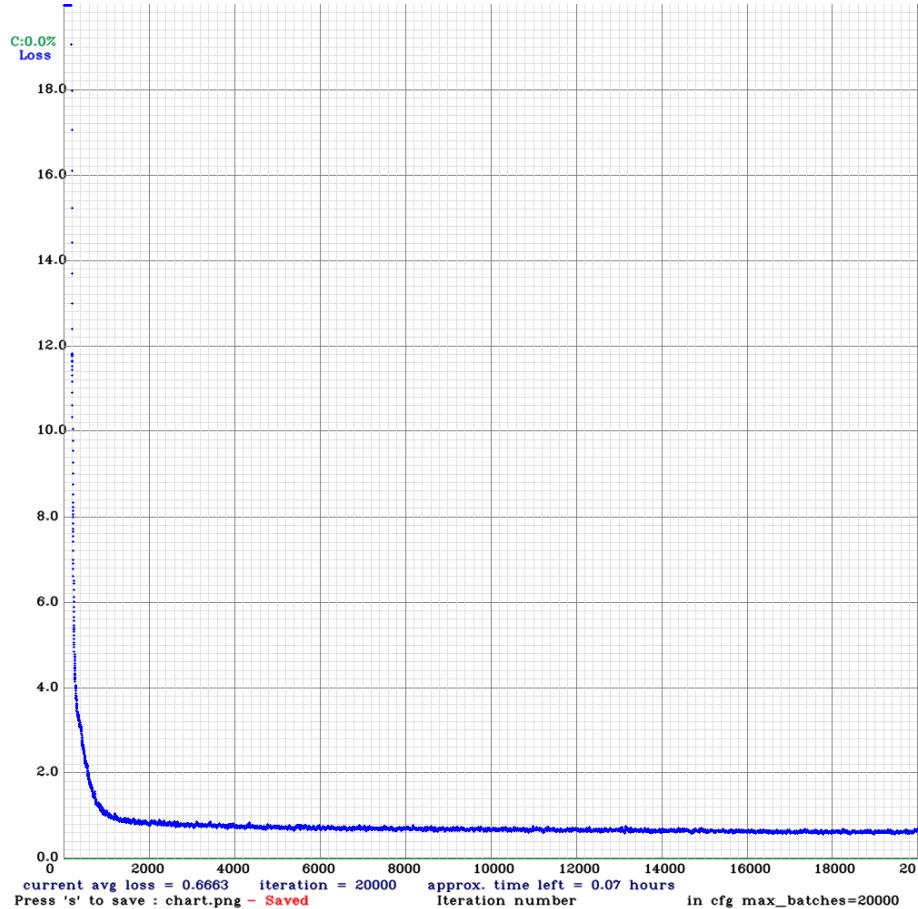


Figura 4.22: Gráfica de entrenamiento mostrando el error de la red *YOLO V3 Tiny*.

4.3.2. Controlador diseñado para el seguimiento de carril a partir de la salida de la red neuronal

Una vez obtenida la salida de la red neuronal que indica el centro del carril, es necesario elaborar un controlador que comande velocidades lineales y angulares al robot. Para ello se ha realizado un controlador proporcional (P) con una serie de constantes para lograr un comportamiento adecuado, dicho controlador está disponible en el Código 4.11.

4.3.3. Transformación del modelo de *PyTorch* a *TensorRT*

Como se expuso en el Capítulo 3, existe una librería llamada *PyTorch to TensorRT* capaz de convertir modelos de *PyTorch* a modelos optimizados aprovechando los tensores de las gráficas dedicadas, si bien la placa *NVIDIA Jetson Nano* no dispone de una GPU con tensores²⁰, sí permite realizar inferencia utilizando operaciones con *FP16*

²⁰<https://connecttech.com/jetson/jetson-module-comparison/>

```

def setSteering(self, steering):
    if steering > self.FORWARD_RANGE:
        rightSpeed = self.STEERING_OFFSET + abs(steering *
            self.STEERING_GAIN)
        if rightSpeed < self.MIN_SPEED:
            rightSpeed = self.MIN_SPEED
        leftSpeed = self.TURNING_SPEED
        self.goRight()
    elif steering < -self.FORWARD_RANGE:
        leftSpeed = self.STEERING_OFFSET + abs(steering *
            self.STEERING_GAIN)
        if leftSpeed < self.MIN_SPEED:
            leftSpeed = self.MIN_SPEED
        rightSpeed = self.TURNING_SPEED
        self.goLeft()
    else:
        self.goForward()
        leftSpeed = self.throttleMotor
        rightSpeed = self.throttleMotor

    self.setSpeed(leftSpeed, rightSpeed)

```

Código 4.11: Controlador P realizado para el seguimiento de carril.

lo que aumenta el rendimiento en gran medida. Esta conversión se realiza de forma muy sencilla llamando a la función `torch2trt()` tal y como representa el Código 4.12.

```

data = torch.zeros((1, 3, 224, 224)).cuda().half()
model_trt = torch2trt(model, [data], fp16_mode=True)
torch.save(model_trt.state_dict(), 'model_tensor.pth')

```

Código 4.12: Conversión del modelo para realizar optimización y aumentar el rendimiento.

En la Figura 4.23 están representadas capturas de la ejecución en el entorno real desde la perspectiva de la visión del robot.



Figura 4.23: Visión del robot durante la ejecución en el entorno real.

4.4. Diseño del software a partir de un diagrama de clases

El diseño del software se ha realizado a partir de un diagrama de clases, utilizando la herramienta *UMLET*²¹, que permite estructurar crear clases, dependencias entre ellas y los atributos y funciones que contiene. El presente diseño, representado en la Figura 4.24, contiene una clase principal llamada *AutonomousVehicle* que tiene instancias de la clase *LaneFollower* que implementa el seguimiento de carril, de la clase *JetRacer* que implementa el controlador del vehículo y la comunicación con los motores del mismo. También contiene una instancia de la clase *ObjectDetector* que implementa la detección de objetos y, a su vez, la clase *ObjectDetector* hace uso del flujo de vídeo proveniente de un *topic* de *ROS*.

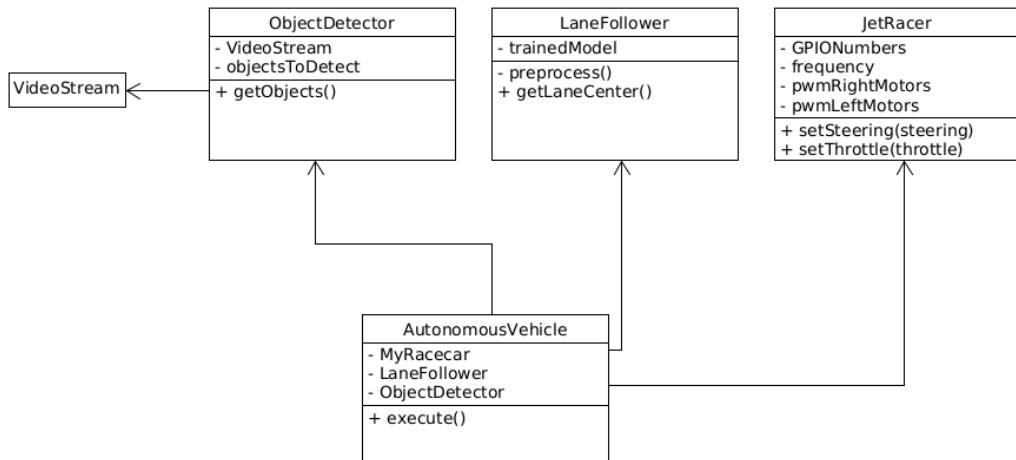


Figura 4.24: Diagrama de clases.

²¹<https://www.umlet.com/>

Capítulo 5

Conclusiones

En este capítulo se realiza una recapitulación de los problemas resueltos y las soluciones utilizadas, así como los experimentos realizados para validar los resultados. Por último, se citan una serie de posibles usos alternativos del software utilizado.

5.1. Conclusiones

El objetivo principal era implementar un coche autónomo bajo una plataforma de bajo coste y reducido tamaño capaz de circular por un circuito o carretera en un entorno dinámico interactuando con objetos propios de una ciudad, como semáforos, señales de stop o peatones. Dicho objetivo debía ser llevado a cabo en dos entornos distintos; en un entorno simulado utilizando el simulador *Gazebo*¹ y en un entorno real, ensamblando un robot donde se utiliza como cerebro, una placa *NVIDIA Jetson Nano*² y como sensor principal, una cámara USB.

Para cumplir dichos objetivos, se han utilizado dos redes neuronales. Por un lado, para realizar el seguimiento de carril, se ha hecho uso de la librería *JetRacer*³. Dicha librería implementa una red residual, concretamente la red preentrenada *ResNet-18*, que mediante un entrenamiento previo a partir de un *dataset* propio, proporciona como salida de la red, el centro del carril o carretera. Dicha salida se utiliza como entrada en un controlador implementado para poder seguir recto o girar en caso de enfrentarse a una curva.

Por otro lado, se ha utilizado la red *YOLO V3 Tiny* mediante el *framework* *Darknet*⁴ y su implementación en *ROS*, *Darknet ROS* para la detección de objetos. En el caso del entorno real, la red mencionada ha sido entrenada a través de un

¹<https://github.com/gazebosim/gz-sim>

²<https://developer.nvidia.com/embedded/jetson-nano>

³<https://github.com/NVIDIA-AI-IOT/jetracer>

⁴<https://pjreddie.com/darknet/>

dataset propio con los objetos reales, con el objetivo de ofrecer una detección de mayor fiabilidad.

Todo ello ha sido combinado en dos paquetes *ROS*, diferenciando entorno simulado y real.

La principal limitación del sistema es la potencia de la placa utilizada. Al necesitar dos redes neuronales y para obtener un rendimiento aceptable, es necesario trabajar con una resolución considerablemente baja. Por otra parte, al disponer de una sola cámara esta debe tener un ángulo de inclinación hacia abajo para poder visualizar correctamente el circuito, lo que provoca un campo de visión reducido que limita la detección de objetos de gran altura.

5.2. Líneas futuras

Tal y como se expuso en el Capítulo 1, existen multitud de casos donde se utilizan vehículos autónomos, y no únicamente en el ámbito de la conducción. Por lo que la solución implementada podría ser aplicable a otros usos.

Por ejemplo, en el ámbito de la inspección, la red neuronal de seguimiento de carril podría ser utilizada para navegar a lo largo de un túnel, como el de la Figura 5.1, en el que buscan artefactos explosivos a través de la red de detección de objetos.

Otro posible uso sería la exploración de bosques o montañas para prevenir incendios de modo que el robot debería seguir la senda o camino en busca de fuego o signos de peligro haciendo uso de la detección de objetos para poder alertar correctamente.



Figura 5.1: Posibles usos alternativos del software implementado.

Todo ello son ámbitos de aplicación que no hacen más que proporcionar una intuición acerca del potencial de las redes neuronales, que mediante un entrenamiento a través de un conjunto grande y completo de datos, permiten adaptarse a multitud de situaciones.

Bibliografía

- [Ayoub and Schneider-Kamp, 2021] Ayoub, N. and Schneider-Kamp, P. (2021). Real-time on-board deep learning fault detection for autonomous uav inspections. *Electronics*, 10:1091.
- [Bochkovskiy et al., 2020] Bochkovskiy, A., Wang, C.-Y., and Liao, H.-Y. M. (2020). Yolov4: Optimal speed and accuracy of object detection.
- [Dal, 2021] Dal, A. (2021). Distance(webcam) estimation with single-camera opencv-python.
- [Fukushima, 1979] Fukushima, K. (1979). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *NHK Broadcasting Science Research Laboratories*.
- [J.Pomeyrol, 2019] J.Pomeyrol (2019). Canonical concreta el soporte de 32-bit para ubuntu 20.04 lts. Muy Linux.
- [Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization.
- [Padilla et al., 2021] Padilla, R., Passos, W. L., Dias, T. L. B., Netto, S. L., and da Silva, E. A. B. (2021). A comparative analysis of object detection metrics with a companion open-source toolkit. 10(3).
- [Raiciu, 2009] Raiciu, T. (2009). How night vision works. *autoevolution.com*.
- [Redmon and Farhadi, 2018] Redmon, J. and Farhadi, A. (2018). Yolov3: An incremental improvement. *arXiv*.
- [Ros, 2021] Ros, I. (2021). Tesla monta cpus zen de amd y gpus rdna 2 en sus coches.
- [SAE, 2018] SAE, I. (2018). Sae j3016 levels of driving automation. *SAE J3016*.
- [Today, 2021] Today, C. (2021). Why tesla using pytorch for autopilot. *Corporates Today*.

- [von Szadkowski and Reichel, 2020] von Szadkowski, K. and Reichel, S. (2020). Phobos: A tool for creating complex robot models. *Journal of Open Source Software*, 5(45):1326.