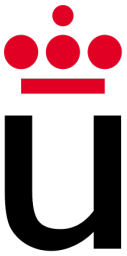


Universidad
Rey Juan Carlos

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE INFORMÁTICA

Máster Universitario en Visión Artificial

Trabajo Fin de Máster

Controlador visual basado en *Deep Learning* para la conducción autónoma en robots reales

Autor: Francisco Pérez Salgado

Tutor: José María Cañas Plaza

Curso académico 2019/2020

"Vida antes que muerte.
Fuerza antes que debilidad.
Viaje antes que destino."

— Brandon Sanderson, El camino de los reyes

Agradecimientos

No ha sido un camino de rosas, lo admito; pero no cambiaría ni un sólo nanosegundo del tiempo que he invertido en este proyecto por mil motivos. No voy a enumerarlos, sólo mencionaré el más importante: la satisfacción personal que trae consigo el la conclusión de este viaje. Muchos altos, muchos bajos, muchas horas frente al ordenador, otras tantas frente a mi pequeño compañero de viaje «Ruedín v2» viéndole sufrir y él viéndome a mi, y sobretodo, gente a la que dar las gracias.

En primer lugar, al tutor y coartífice de este trabajo José María Cañas. Igual que hace algunos años durante mi Proyecto de Fin de Carrera, ha sido el metrónomo que ha marcado el *tempo* de este proyecto, tan necesario para no perderse cuando las dificultades afloran. Gracias por tu tiempo y tu por infinita paciencia, José María.

Otro pilar fundamental de lo conseguido aquí es mi familia. Durante toda mi vida han estado a mi espalda, como colchón de seguridad, para no hacerme daño si me caía; esta vez tampoco han faltado. Cabe también una disculpa, por no haber tenido mucho tiempo para dedicarles durante el timepo dedicado a este trabajo. Nunca hay palabras para describir todo el amor, cariño y apoyo que ofrece la familia, pero ellos ya lo saben. Gracias.

A mi amiga Sandra. A pesar de que también eres familia desde hace ya bastantes años y todo lo dicho arriba te incluye, tengo una espinita clavada desde el PFC al no haberte incluido en los agradecimientos cuando estuviste ahí apoyando como la que más; así que este pequeño párrafo es para ti. Prácticamente empecé en el mundo de la robótica de la mano contigo y creo que no se puede pedir mejor inicio de viaje. Siempre estás ahí, siempre feliz y siempre, siempre ayudándome a crecer. Eres uno de mis motores. Mil gracias.

A mi amigo Nacho. Mi fuente de *memes* y de desahogos tanto técnicos como personales. No te voy a decir nada que no sepas, pero esto no se acaba aquí. Nos quedan mil proyectos que hacer... estás avisado. Esta batalla ha sido conjunta, que lo sepas. Gracias.

Por último, a todas las personas que forman parte de mi vida de una forma u otra, porque de todas se aprende en mayor o menor medida.

Resumen

En los últimos años, el campo de la Inteligencia Artificial (AI por sus siglas en inglés) ha explotado en muchas disciplinas, especialmente la Visión Artificial (VA). Uno de los problemas punteros que se está tratando de solucionar con los últimos avances en investigación es el de la conducción autónoma. El acto de conducir para un humano es algo que no requiere mucho esfuerzo mental, ya que somos capaces de tomar decisiones de forma casi instantánea ante cualquier situación que se nos presente mientras conducimos. Sin embargo, para las máquinas este proceso no es tan sencillo. En los últimos años se han propuesto soluciones a este problema empleando el aprendizaje automático, más concretamente las redes neuronales artificiales aplicadas a la VA. Este Trabajo de Fin de Máster trata de solucionar el problema de la conducción autónoma en robots reales utilizando un controlador visual basado en aprendizaje profundo (o *deep learning*); concretamente haciendo uso de un pequeño robot que sea capaz de navegar por una pista sin salirse valiéndose únicamente de la información sensorial proporcionada por una cámara de color.

Los modelos entrenados han de ser capaces de tomar decisiones de calidad de tal forma que la conducción sea lo más robusta posible. Para ello es imprescindible disponer de unos datos de entrenamiento de calidad, que sean representativos del problema y ayuden a las redes a generalizar ante cualquier entrada nunca vista por las mismas. El robot entrenado ha de ser capaz de navegar en diferentes circuitos bajo condiciones de iluminación diversas, por lo que se han creado conjuntos de datos específicos para la tarea a resolver.

Un requisito imprescindible es que la solución ha de utilizar técnicas de aprendizaje automático; más concretamente, el aprendizaje profundo. Para ello, se ha estudiado el estado actual de los modelos de *deep learning* para visión artificial más novedosos, así como el *hardware* específico para trasladar la solución a un robot real. Adicionalmente, se ha desarrollado una plataforma *software* llamada BehaviorStudio para la ejecución y validación de los algoritmos conseguidos para el problema de la conducción autónoma. Por último, se han llevado a cabo diferentes experimentos que demuestran la validez del trabajo realizado.

Las principales contribuciones de este trabajo son el desarrollo de un controlador visual basado en *deep learning* que permite a un robot real cuyo computador es una placa embebida, navegar por diferentes circuitos de forma robusta. Además, se contribuye con el desarrollo de una plataforma *software* destinada a la ejecución y evaluación de algoritmos de control robótico basados en redes neuronales de forma sencilla y flexible.

Índice general

Índice de figuras

Índice de tablas

1

Introducción

En este capítulo se presenta tanto el contexto como la motivación principal que ha impulsado todo el desarrollo de este trabajo. Además se resumen los objetivos del sistema desarrollado seguido de la metodología seguida para la realización del proyecto.

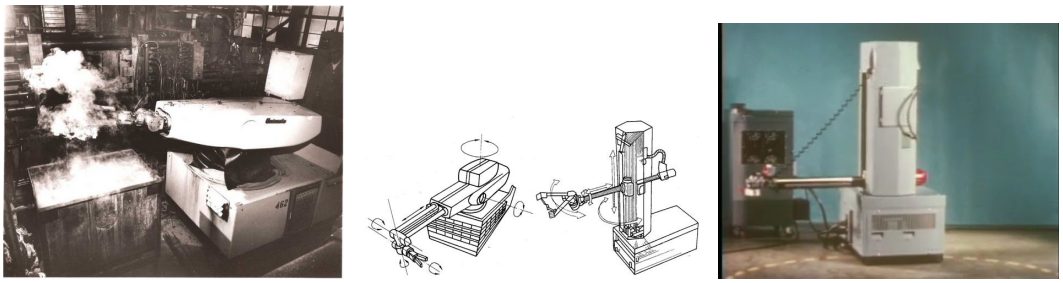
1.1. Robótica

La robótica es una disciplina que engloba tres aspectos: el diseño, construcción y programación de robots. La robótica se combina con muchas otras disciplinas diferentes como la informática, la electrónica, la mecánica, la inteligencia artificial, la ingeniería de control, etc. Citando a *Robot Institute of America*, se puede decir que: "*un robot es un dispositivo multifuncional reprogramable diseñado para manipular y/o transportar material a través de movimientos programados para la realización de tareas variadas.*"

Esencialmente, los robots se componen de sensores, actuadores y computadores, por lo que se puede considerar un robot como cualquier sistema informático o máquina que disponga de esos tres ingredientes. Los sensores son los encargados de obtener información del entorno que rodea al robot. Los actuadores son los encargados de interactuar con ese entorno. Los computadores, por su parte, son los encargados de recoger la información sensorial, analizarla y generar señales en función a esos datos para los actuadores. No obstante, estos tres ingredientes por sí mismos no son suficientes para que un robot pueda llevar a cabo un determinado comportamiento, por lo que hace falta la pieza que dota de «inteligencia» a los robtos, el *software*. Esta es la parte más importante de un robot, ya que es la encargada de procesar la información recibida por los sensores y traducirla a acciones que pueda llevar a cabo el robot a través de sus actuadores.

En el año 1961 la empresa *Unimate* diseña el primer robot programable y controlado digitalmente. Este robot se diseñó para llevar a cabo tareas potencialmente peligrosas para los humanos, en concreto levantar piezas metálicas calientes de una máquina de tintes y colocarlas posteriormente en un lugar determinado.

Este primer desarrollo marcó un hito que inició una carrera masiva de diversos tipos de robots para todo tipo de tareas. En los años sucesivos, debido al crecimiento del sector industrial, la robótica se centró principalmente en los entornos más industrializados. De este crecimiento comenzaron a aparecer robots que se ocupaban de la automatización de las tareas más peligrosas, aburridas e incluso tareas que requerían de gran precisión. Esto supuso que cada vez más, los humanos fueran centrándose en otro tipo de tareas a más alto



(a) **(b)** **(c)**

Figura 1.1: (a) Robot *Unimate*, (b) esquemas del *Versatran* y del *Unimate* y (c) Robot *Versatran*.

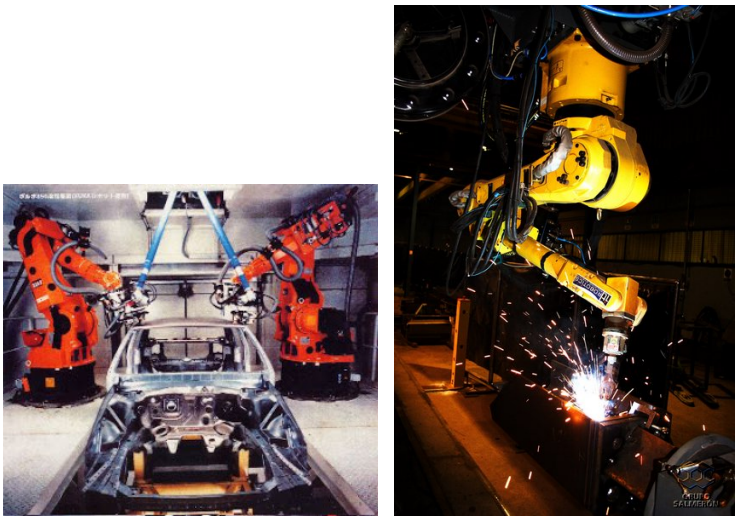
nivel como la investigación y desarrollo de robots más sofisticados. Un ejemplo importante de la época en el sector industrial fue el robot *Versatran* ((Figura 1.1c), desarrollado para transportar cargas dentro de la fábrica de ensamblaje de automóviles Ford, en 1962.

En 1969 aparecen los primeros robots soldadores tan utilizados hoy en día en cadenas de montaje. Este tipo de robots surge de la falta de precisión y velocidad de los trabajadores humanos a la hora de realizar soldaduras a los vehículos en las cadenas de producción. Estos robots supusieron mejoras importantes en las condiciones de los trabajadores, ya que el trabajo de soldadura era considerado como peligroso y dañino para las personas. Además, las cadenas de montaje también se vieron beneficiadas ya que estos robots, al ser muy precisos y veloces en las soldaduras, aumentaron en gran medida la productividad de las fábricas; tarea que un humano no podía llevar a cabo de forma tan rápida, segura y fiable.

A lo largo de los años y gracias a los avances en investigación, los robots han ido mejorando paulatinamente en cuanto a desempeño de tareas cada vez más complejas. Hoy en día existen robots capaces de llevar a cabo tareas complejas no sólo en el campo de la investigación, sino en el ámbito doméstico o industrial. Como se ha comentado, los robots son ideales para realizar tareas repetitivas, peligrosas o aburridas para las personas. Aparte de la ya mencionada tarea de soldadura, el ámbito industrial se ha visto beneficiado en las cadenas de producción y en tareas logísticas, por ejemplo, con los robots utilizados para el embalaje de magdalenas (Figura 1.7a), robots para levantar cargas pesadas de forma autónoma, transporte de herramientas, y multitud de propósitos diferentes. Uno de los ejemplos más claros en la automatización del sector industrial es *Amazon*. Esta empresa cuenta con una flota de robots autónomos que se encargan de transportar mercancías dentro de sus naves industriales [1]. para agilizar la logística de paquetes. Debido a la creciente demanda de transportes más rápidos con el auge de las compras por internet (*e-commerce*) es crucial que la logística sea lo más rápida, precisa y robusta posible, lo que *Amazon* consigue con el uso de robots completamente autónomos que puedan operar las 24 horas del día.

Del auge de la robótica no solamente se ha visto beneficiado el sector industrial. Otro foco en el que cada vez se centran más las empresas es la robótica en el ámbito doméstico. Actualmente, en este sector se comercializan diferentes tipos de robots que resultan muy útiles en las tareas del hogar. Un claro ejemplo son los robots aspiradores; empresas como

[1] `https://www.youtube.com/watch?v=tMpsMt7ETi8`



(a) **(b)**

Figura 1.2: (a) y (b) robots soldadores en plantas de producción industrial.

(a) **(b)** **(c)**

Figura 1.3: (a) Robot embalador de magdalenas, (b) *Roomba* de iRobot y (c) *Mi Robot Vacuum* de Xiaomi.

iRobot con el *Roomba* (Figura 1.7b) o Xiaomi con su *Mi Robot Vacuum* (Figura 1.3c) son pioneras en este sector.

Por último, cabe mencionar que en la actualidad, la conducción autónoma se ha puesto a la cabeza de los problemas a investigar por parte de las empresas y de la comunidad científica. La conducción autónoma trata de solucionar el problema de que un vehículo sea capaz de emular la capacidad de un humano de conducir tomando como información la proporcionada por diferentes sensores como cámaras (RGB, RGBD, monocromáticas, etc.), láser, radar, etc. El problema es complejo, ya que no sólo es necesario que el vehículo sea capaz de mantenerse dentro de la carretera, sino que además deberá cumplir con todas las normas de circulación pertinentes al igual que debe hacer un humano, además de ser capaz de tomar decisiones en multitud de escenarios y condiciones (baja visibilidad, cruce de peatones, accidentes, etc.) que se presentan durante la conducción. Es por esta complejidad que el problema aún no está resuelto.

Existe un estándar: el J3016, ideado por la SAE (Sociedad de Ingenieros automotrices) que categoriza los niveles de conducción autónoma de un sistema en base a las capacidades del vehículo. En la Figura 1.4 se observa la taxonomía que hace referencia a los siguientes

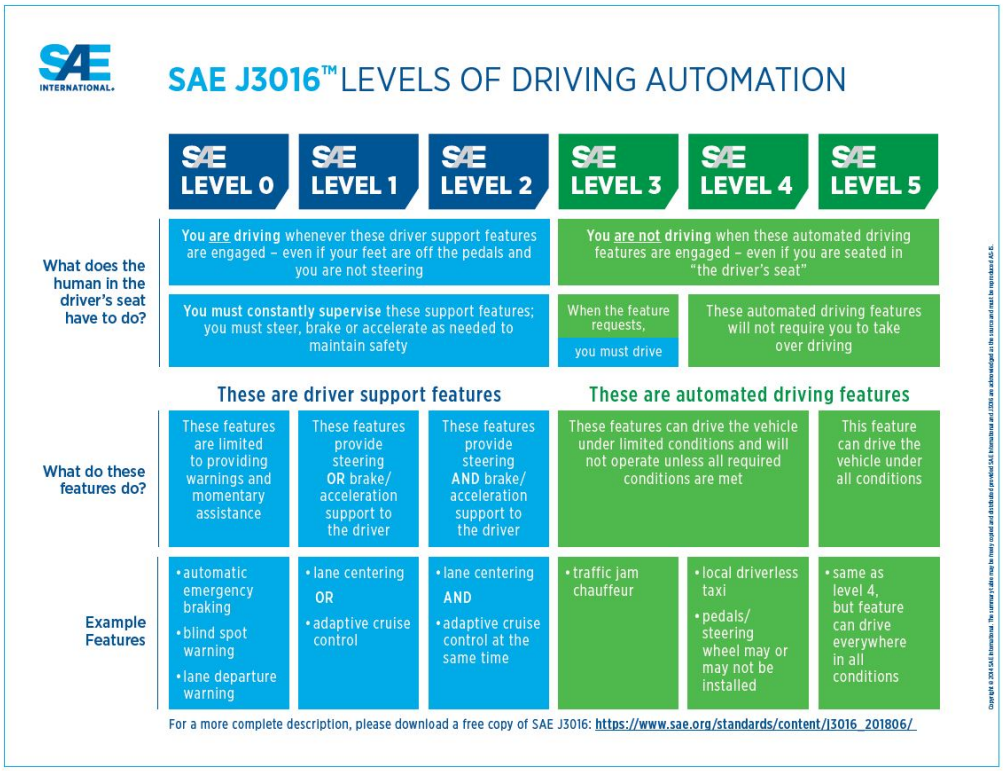


Figura 1.4: Niveles de conducción autónoma según el SAE

niveles:

1. Nivel 0. No hay automatización; conduce un humano.

2. Nivel 1. Asistencia al conductor. Algún sistema de apoyo como velocidad de crucero o auto aparcamiento. Sólo asiste al movimiento longitudinal o lateral.

3. Nivel 2. Automatización parcial. El sistema puede tomar el control del vehículo en determinadas circunstancias controladas. No obstante el humano ha de supervisar en todo momento.

4. Nivel 3. Automatización condicional. El sistema puede tomar decisiones en la conducción recibiendo información del entorno y calculando riesgos. El humano ha de intervenir cuando el sistema lo requiera.

5. Nivel 4. Autonomía. El sistema tiene el control del vehículo; es capaz de detectar objetos y eventos y actuar ante ellos.

6. Nivel 5. Autonomía completa. No se requiere supervisión del humano; el sistema es capaz de realizar una conducción del vehículo en cualquier situación.

Una de las empresas pionera en ofrecer soluciones a este problema de forma comercial es Tesla [2], la cual desde hace ya algunos años comercializa vehículos con la tecnología necesaria para circular por una vía de forma autónoma, a la que han bautizado como

[2] `https://www.tesla.com`

AutoPilot. No obstante, la empresa ofrece esta tecnología como un «asistente para la conducción» ya que, a pesar de que se ha demostrado que el coche es capaz de realizar la tarea, aún necesita la supervisión constante de un humano para evitar accidentes. Teniendo en cuenta la lista del SAE, se puede ver que la tecnología de Tesla está aún en el nivel 2, lo cual dista mucho aún de la conducción autónoma total.

El nivel más alto de esta lista lo tiene Google, con su proyecto Waymo. Teóricamente, Waymo está actualmente en el nivel 4 de la lista de SAE, aunque aún está en etapa de investigación. La tecnología de Waymo ha sido probada con éxito en diferentes entornos tanto de ciudad como de carretera de forma exitosa, no obstante, todas estas pruebas ha sido en entornos más o menos controlados. Este proyecto se explica con más detalle en la sección 1.3

En las siguientes secciones se explicará cómo conecta el problema de la conducción autónoma con la robótica, la visión artificial y el aprendizaje profundo, que son las tecnologías que enmarcan este proyecto.

1.2. *Deep Learning*

El *Deep Learning* o aprendizaje profundo es un campo dentro del aprendizaje automático (*Machine Learning*) que utiliza estructuras de redes neuronales organizadas en capas para conseguir aprender representaciones de los datos más significativas conforme se avanza en las capas de la red. La palabra *deep* hace referencia al número de capas que se utilizan en la red, por lo que se suele considerar que una red neuronal de más de 3 capas ya es una red neuronal profunda.

Una red neuronal es un tipo de estructura de cómputo que se caracteriza por emular el funcionamiento de un sistema nervioso humano, donde las diferentes neuronas conectadas entre sí trabajan juntas para solucionar un problema dentro de un dominio. La definición formal de una red neuronal es que es una función de aproximación de funciones universal, donde dado una serie de datos que entran a un sistema se pueda obtener una determinada salida deseada.

En la Figura 1.5 se ilustra una arquitectura de red neuronal básica, conocida como perceptrón multicapa (MLP o *MultiLayer Perceptron* por sus siglas en inglés). Esta red en particular consta de 3 capas: una capa de entrada, una capa oculta y una capa de salida. En la capa de entrada se representan los datos que ya han sido preparados para alimentar a la red en un formato determinado; la capa oculta se encarga, principalmente, de procesar la entrada de datos realizando los cálculos pertinentes para obtener una salida deseada utilizando los parámetros (pesos) de la misma; y en la capa de salida se reciben los datos procesados por la red neuronal. Todas las neuronas en una capa están conectadas con alguna (o todas) las neuronas de la capa siguiente, y a cada neurona se le asigna un número llamado peso que se utilizan como moduladores de la red. El proceso de aprendizaje consiste principalmente en ajustar esos pesos para que la red sea capaz de aprender a procesar los datos de entrada de manera que la salida sea igual o muy similar a la que se espera. Ese ajuste de pesos se realiza mediante ensayo-error de la siguiente manera: dada una entrada de datos $X = \{x_1, x_2, ..., x_n\}$, la red procesará dichos datos y generará una salida estimada $\hat{y}$ que será comparada con la salida esperada y produciendo un error

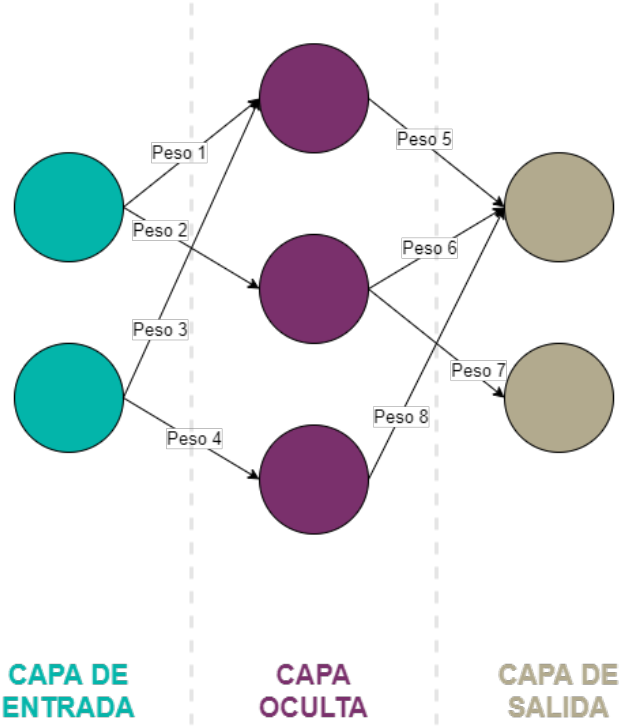


Figura 1.5: Esquema de red neuronal sencilla

(mediante alguna función como error cuadrático medio - MSE - o error absoluto medio - MAE -), ya que la salida de la red puede que no coincida exactamente con la salida esperada; el objetivo del aprendizaje de una red neuronal es que ese error se reduzca al máximo posible (idealmente el error será cero al final del entrenamiento, ya que la salida de la red y la salida esperada serán la misma, es decir, la red no ha cometido errores); para ello, cada vez que la red procesa una muestra de datos ésta actualiza sus pesos mediante una técnica llamada *back propagation*, que recibe su nombre por la forma en que el error cometido por la red se propaga hacia atrás actualizando los pesos de todas (o algunas) sus capas mediante cálculos de derivadas.

Las redes neuronales pueden tener multitud de formas diferentes, con distintas configuraciones en cuanto a número de capas, número de neuronas por capa, conexiones entre neuronas, etc. Actualmente, los modelos más grandes diseñados para tareas tanto de visión artificial como de procesamiento de lenguaje natural (NLP), que son dos de los campos donde el aprendizaje profundo ha tenido más impacto en la última década, pueden tener cientos de capas. No obstante, los primeros modelos neuronales eran mucho más pequeños debido a la limitación en la capacidad de cómputo.

En 2006 Geoffrey Hinton [43] propone un nuevo método de entrenamiento de las redes neuronales existentes hasta el momento, utilizando un entrenamiento por capas; de esta forma, cada capa de la red irá aprendiendo diferentes características de los datos de entrada empezando por las características más primitivas hasta las características más complejas extraídas de los datos según se avanza en las capas de la red. Este procedimiento se lleva a cabo hasta que se han entrenado todas las capas de la red. Este avance permitió el entrenamiento de redes mucho más profundas que las existentes en ese momento, popularizando el término *deep learning* o aprendizaje profundo en castellano. Con el tiempo se

ha demostrado que las redes neuronales más profundas otorgan resultados mucho mejores que otros modelos con menos capas o que las técnicas clásicas de *machine learning*.

En 2012, se inicia una revolución en el campo de la visión artificial con la llegada de AlexNet [1]. Alex Krizhevsky batió todos los récords de precisión en el problema de clasificación de imágenes en el torneo anual de *ImageNet* en 2012 al proponer un modelo basado en aprendizaje profundo con redes neuronales convolucionales acelerando el cómputo a través de GPU. Después de competir en el *ImageNet Large Scale Visual Recognition Challenge* [3], AlexNet saltó a la fama. Logró un error del 15,3 %. en la tarea de clasificación de imágenes, que fue un 10,8 % más bajo que el del segundo puesto. Este resultado fue gracias a la profundidad del modelo que era necesaria para su alto rendimiento y al uso de las redes neuronales convolucionales (CNN). En 2012 este tipo de procesamiento era muy caro desde el punto de vista computacional, pero se hizo factible gracias al uso de las GPU (*Graphic Processing Unit* o Unidades de procesamiento gráfico) durante el entrenamiento. Tras este hito, el campo de la visión artificial explotó y comenzaron a surgir multitud de modelos novedosos que expandieron los límites del aprendizaje profundo mejorando las técnicas de aprendizaje automático que existían hasta el momento. Esta revolución dio pie a la resolución de diferentes tipos de problemas usando la visión artificial en campos como la industria (p.e.: detección de imperfecciones en cadenas de montaje), la robótica (p.e.: navegación autónoma), la medicina (p.e.: detección de cáncer de mama), etc.

Uno de los proyectos de visión aritificial más ambiciosos de la última década ha sido GoogleBrain, que surgió también en 2012 de la mano de Jeff Dean y de Andrew Ng. Para este proyecto se desarrolló una red neuronal profunda que era capaz de detectar patrones en imágenes y vídeos de Youtube, llegando a ser capaz de reconocer gatos en vídeos de Youtube analizando las imágenes de cada fotograma. Dos años más tarde, la empresa DeepMind, dedicada al mundo de los videojuegos y el *deep learning*, es absorbida por Google. En 2016, DeepMind consiguió batir al mejor jugador del mundo (Lee Sedol) en el juego de mesa Go por 5 a 1, utilizando su algoritmo de *deep learning*: AlphaGo, mediante jugadas «creativas» y «nunca vistas» según jugadores de Go profesionales. No obstante, un año después crearon AlphaGo Zero, que es igual de potente que el AlphaGo original, pero con la característica de que es un sistema autodidacta, esto es, que ha aprendido por sí mismo a jugar al juego mediante aprendizaje por refuerzo jugando contra sí misma sin ningún tipo de información a priori. AlphaGo y AlphaGo Zero se basan en la visión para procesar las jugadas en cada turno, y tomar decisiones en función de la disposición de las fichas en el tablero.

En la acutalidad la IA se encuentra en plena ebullición. Tanto es así que una gran mayoría de empresas tecnológicas quieren implementar soluciones basadas en el *deep learning* para sus productos y/o servicios. Así, las grandes empresas tecnológicas como Google, Amazon, Facebook, Tesla, etc., ya ofrecen soluciones basadas en IA en diferentes campos como el NLP, la visión artificial o la robótica. Como ejemplos de los últimos dos años tenemos el algoritmo GPT-3 para generación automática de texto (creado por OpenAI); Google Home y Amazon Echo Dot (Figura 1.6) que son asitentes de voz inteligentes; Facebook utiliza el *deep learning* para orientar sus anuncios e identificar objetos en las imágenes subidas por sus usuarios; Google con su buscador utiliza *deep learning* con modelos basados en Transformers [10] para obtener mejores resultados en las búsquedas, incluso Amazon en sus almacenes con su flota de robots logísticos que utilizan *deep learning* para

[3] http://image-net.org/challenges/LSVRC/

Figura 1.6: Amazon Echo Dot y Google Home

moverse de forma eficiente por las naves, y también Tesla y Google con sus proyectos de conducción autónoma basados en IA Autopilot[4] y Waymo[5] respectivamente.

Una de las grandes compañías que han apostado por el *deep learning* es Facebook. Con Yann LeCun a la cabeza en el laboratorio de IA de Facebook, se desarrolla DeepFace [44] en 2014. DeepFace es un algoritmo basado en *deep learning* que es capaz de reconocer rostros en imágenes digitales con la misma precisión que un humano.

Una empresa de nacimiento reciente que es consecuencia de la explosión en la IA es OpenAI[6]. Esta empresa fundada por el dueño de empresas como Tesla y SpaceX, Elon Musk, se dedica a la investigación sobre diferentes campos de la IA sin ánimo de lucro. Uno de sus mayores logros desde su fundación es la creación de sus modelos de generación de texto en lenguaje natural: GPT-2 y su actual versión GPT-3. Estos dos modelos utilizan arquitecturas Transformers (que provocaron la explosión del campo del procesamiento del lenguaje natural o NLP en 2018), que son modelos de redes neuronales mucho más potentes que las LSTM permitiendo obtener información contextual con mucho mayor alcance temporal. Los algoritmos GPT-2 y GPT-3 tienen como objetivo la generación de texto en lenguaje natural simulando a un humano. Es tal la potencia de dichos modelos, que a pesar de que OpenAI es una empresa de *softwre* libre, no liberaron el modelo de GPT-2 por el peligro que suponía en la generación de noticias falsas o textos de odio de forma masiva. A día de hoy, y como ya sucedió en 2012 con el campo de la visión artificial, el NLP está en pleno auge; incluso Google implementa modelos basados en estas arquitecturas para su buscador.

[4] `https://www.tesla.com/autopilot`
[5] `https://waymo.com`
[6] `https://openai.com`

(a) **(b)**

Figura 1.7: (a) Cámara del *AutoPilot* del Tesla, (b) Cámaras del coche Waymo de Google.

1.3. *Deep Learning* aplicado a la robótica con visión

Uno de los campos de la ingeniería que mejor casa con el aprendizaje profundo es la robótica. Esto es así porque la «inteligencia» que se asocia a los robots se basa en su autonomía, es decir, la capacidad del robot de llevar a cabo comportamientos simples o complejos de forma totalmente autónoma. Históricamente estos comportamientos complejos se lograban mediante algoritmos sofisticados con mucha lógica de control, lo que llevaba a hacer que los algoritmos que generaban estos comportamientos fueran muy complejos y con una casuística muy extensa. Con la explosión del *deep learning* el campo de la robótica se ha visto enormemente beneficiado, ya que la algoritmia subyacente que conseguía la autonomía de los robots se ha simplificado en gran medida al añadir el factor de aprendizaje. Uno de los campos que más impacto ha tenido sobre la robótica aplicando el aprendizaje profundo ha sido la visión artificial. Actualmente la visión artificial aplicada a robots se puede ver en multitud de proyectos muy novedosos y en auge como la conducción autónoma con Tesla y Google a la cabeza (Figura 1.7); en el ámbito industrial con las cadenas de producción (robots soldadores, robots ensambladores, etc.); en el ámbito doméstico con las aspiradoras robóticas con cámara integrada que implementan algoritmos de SLAM visual (*Simultaneous Localization and Mapping* por sus siglas en inglés); y sobretodo en el ámbito de la investigación, donde hay multitud de estudios en curso sobre la visión artificial aplicada a la robótica.

Un ejemplo perfecto de la visión aplicada a la robótica en línea con este proyecto es el ya mencionado Waymo, de Google. El proyecto Waymo comenzó en 2009 y su objetivo principal es el de resolver la tarea de la conducción autónoma. Como se comentó en anteriores secciones, el nivel de autonomía máxima de la conducción autónoma es el nivel 5; Waymo ya ha conseguido llegar al nivel 4. Google se centra sobretodo en la conducción en entornos urbanos, donde, a priori, el problema es más complejo por el nivel de variables y situaciones imprevistas que pueden presentarse durante la conducción y a las que los conductores humanos se enfrentan todos los días. Este proyecto lo han bautizado como Waymo One[7]. Adicionalmente, Google tiene además un proyecto paralelo para conducción de vehículos de logística (furgonetas de reparto y camiones de transportes de mercancías) llamado Waymo Via[8], más orientados a profesionales del transporte logístico. En la figura 1.8 se pueden ver algunos vehículos utilizados en ambos proyectos.

[7] `https://waymo.com/waymo-one/`
[8] `https://waymo.com/waymo-via/`

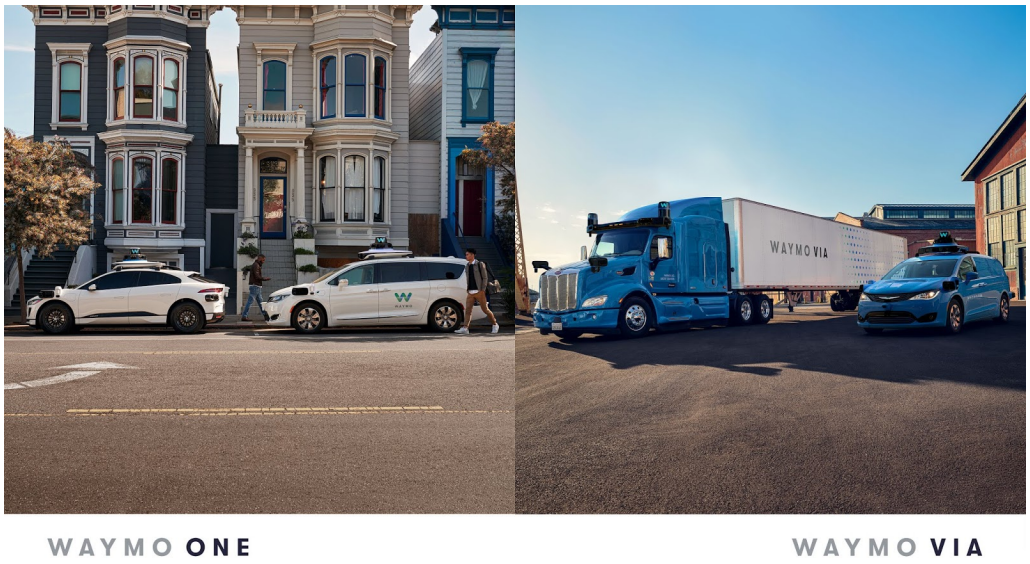


Figura 1.8: Proyectos en marcha de Google Waymo

Waymo One lleva 3 años operando en Phoenix (Arizona) de forma comercial como servicio de taxis autónomos. Las pruebas realizadas por Waymo One están limitadas a un área metropolitana de esta ciudad, restringiendo su zona de actuación a dichas áreas. Este servicio de taxis autónomos opera a través de una aplicación con la que los usuarios pueden contratar un servicio de taxi como cualquier otro. Todos los taxis de Waymo operan con modelos Chrysler Pacifica modificados con todos los sensores necesarios para la conducción autónoma; no obstante, todos los taxis están supervisados en todo momento por un conductor humano que estará listo para intervenir cuando la situación lo requiera.

Los vehículos autónomos de Waymo están diseñados para tener autonomía completa con sensores que otorgan visión en los 360 grados y láseres con alcance máximo de 300 metros de distancia. Entre los sensores integrados en los vehículos se encuentran: cámaras para la detección de carriles, láseres de corto alcance para detección de objetos cercanos, radar para la monitorización de objetos en movimiento (otros vehículos, peatones, etc.). Además integra interfaces en el interior del vehículo para que los supervisores humanos puedan interactuar con las funciones que ofrece el sistema. Para las pruebas, los ingenieros de Google crearon un simulador llamado Carcraft, que simula las diferentes condiciones de conducción con 25,000 coches virtuales autónomos en las calles de Texas, Mountain View y Phoenix.

A pesar de todos los avances conseguidos por Google, su sistema no es infalible, ya que se han registrado varios accidentes a lo largo de los años de vida del proyecto. Google sigue investigando en el problema de la conducción autónoma y ya ha conseguido llegar al nivel 4 del estándar de la SAE (Figura 1.4), sin ningún piloto al volante en algunos entornos controlados sin condiciones climáticas adversas, en carreteras con poca densidad de tráfico y sistemas de carreteras no complejos. Lo que indica que el problema aún no está resuelto del todo.

Por su parte, el proyecto Waymo Via sigue avanzando en investigación. Este proyecto comenzó su andadura en 2017 en las carreteras de California y Arizona. Se estima que para 2020 esas rutas se amplíen a Texas y Nuevo México. También se engloban en Waymo

Via los vehículos de reparto comerciales. Este año 2020 Google ha anunciado un programa piloto con la empresa de reparto UPS, donde los vehículos autónomos se encargarán de transportar la mercancía de las tiendas de UPS a una instalación logística de la misma empresa. Los camiones de transporte de mercancías y las furgonetas de reparto de Google incorporan la misma tecnología que los coches autónomos de Waymo One.

Por otra parte, en el ámbito de la investigación han surgido multitud de proyectos diferentes en la última década que relacionan estos tres campos: la robótica, la visión artificial y el aprendizaje profundo. Algunos de ellos se citan a continuación.

En [5] se estudió la capacidad de las redes neuronales para reconocer objetos textiles que cuelgan de un sólo punto, como si una pinza robótica los estuviera cogiendo. Se probó con diferentes prendas como pantalones, toallas y camisas de colores, formas y materiales diversos. Con un conjunto de 6 prendas los autores consiguieron una precisión del 100 % prediciendo además el punto de agarre de la prenda con un error de aproximadamente 5 cm. Como estudio comparativo, demostraron que la aproximación basada en redes neuronales profundas de visión mejoraban a algoritmos clásicos como SVMs.

En [6] se propone un estudio para solucionar el problema del agarre robótico. En este trabajo integraron la percepción visual y la percepción de contacto para realizar una clasificación háptica de los objetos (suave, rugoso, duro, pesado, etc.). Este sistema integrado en un robot podría predecir de antemano el tipo de objeto antes de entrar en contacto con él, para ajustar su agarre en función del objeto. Por ejemplo, un agarre de un objeto frágil como un huevo, necesita menos presión que un objeto más duro y pesado como una bola metálica.

En [7] mediante el uso de una arquitectura de redes neuronales a la que llaman *Visuo-Motor Deep Dynamic Neural Network* (VMDNN) que es una arquitectura basada en redes neuronales recurrentes (RNN), consiguen enseñar a un robot a reconocer gestos hechos por un humano, a «prestar atención» y a la clasificación de objetos y el ajuste del agarre. En este estudio se hizo uso de un robot que observaba a un colaborador humano, en un escenario en el que había dos objetos diferentes. La tarea del humano era señalar a uno de los dos objetos para que el robot enfocara su atención en el objeto señalado, lo reconociera y ajustara el agarre para ese objeto determinado. El sistema logró un agarre satisfactorio un 85 % de las veces en simulación, utilizando el simulador iCub.

Por último, en [8] entrenaron un modelo neuronal basado en redes neuronales recurrentes RNN para predecir las maniobras de tráfico en un vehículo conducido por un humano. El objetivo del trabajo era mejorar los sistemas anti colisión implantados en los coches autónomos, que típicamente no reaccionan a tiempo para evitar un accidente. Se entrenó este modelo en aproximadamente 1900 kilómetros de carreteras de alta y baja velocidad con 10 conductores diferentes. Los datos de entrenamiento incluían el vídeo desde la posición del conductor, el vídeo de la carretera frente al vehículo, la dinámica del vehículo, las coordenadas GPS y los mapas de las calles alrededor de la posición del vehículo.

1.4. Objetivos

Una vez presentado el contexto en el que se desarrolla este trabajo, a continuación se plantea y describe el problema abordado, así como la metodología aplicada al desarrollo *software* de las soluciones propuestas.

El objetivo principal de este trabajo es conseguir un algoritmo de control visual para conducción autónoma basado en *deep learning* y aplicar la solución a un robot pequeño en un entorno real. Además, se propone el desarrollo de una plataforma *software* para la ejecución de comportamientos neuronales orientado a la conducción autónoma bautizada como BehaviorStudio[9], que servirá como infraestructura para probar la solución aportada para el objetivo principal.

Para desarrollar este trabajo se han dividido ambos objetivos generales en subobjetivos más abordables para una mejor gestión del tiempo y una planificación más eficiente. Los puntos 3 y 4 están relacionados con el desarrollo de la plataforma BehaviorStudio, los puntos 5 y 6 con el desarrollo de la solución para la conducción autónoma, y los puntos 1, 2, 7 y 8 son comunes para ambos objetivos. Se detallan a continuación:

1. Estudiar el estado del arte de redes neuronales, visión artificial y conducción autónoma para obtener una visión panorámica del problema a resolver y ayudar en la toma de decisiones para de proyecto.

2. Estudiar el contexto previo a este proyecto mediante la lectura y la réplica del Trabajo de Fin de Máster de Vanessa Fernández [9], trabajo del que parte este mismo ampliando una de sus líneas futuras.

3. Diseño de la plataforma BehaviorStudio, generando todos los diagramas y documentación pertinente para el desarrollo del *software*. En esta etapa se generarán los diagramas UML del diseño, *mockups* de la aplicación y los requisitos de la misma.

4. Desarrollo de los diferentes componentes de la plataforma BehaviorStudio, generación de pruebas unitarias y confirmación del funcionamiento en entornos simulados mediante validación experimental.

5. Adquisición y montaje del robot escogido como plataforma *hardware* para la resolución del objetivo principal. Instalación de todas las herramientas necesarias para el desarrollo. Pruebas de ensamblaje, teleoperación, conectividad y movilidad del robot.

6. Desarrollo de los algoritmos de control visual basado en *deep learning* centrado en el uso de redes neuronales convolucionales para el problema de regresión utilizando redes recurrentes y aplicación de los mismos en el robot real.

7. Integración de la solución para la conducción autónoma en la plataforma BehaviorStudio y pruebas para comprobar el correcto funcionamiento de todas las partes en un entorno real.

[9] `https://jderobot.github.io/BehaviorStudio/`

8. Experimentación y evaluación de la solución conseguida sobre diferentes circuitos reales.

Además de los subobjetivos arriba listados, se querrá satisfacer una serie de requisitos para considerar válido el desarrollo del proyecto. Estos requisitos son:

- La plataforma BehaviorStudio deberá ofrecer un interfaz de usuario sencillo, intuitivo y usable. Además deberá ofrecer la posibilidad de mostrar la información sensorial en tiempo real.
- El robot deberá ser capaz de completar al menos una vuelta en cada circuito propuesto tanto en sentido horario como en sentido antihorario.
- La plataforma BehaviorStudio deberá funcionar indistintamente en entornos simulados y en reales.
- El proyecto al completo se desarrollará en Python utilizando las librerías de programación más novedosas como PyTorch y PyQt.
- Los algoritmos propuestos han de ser ágiles no tardando demasiado tiempo entre iteraciones para que el comportamiento de los robots sea reactivo y los movimientos más fluidos.
- Se utilizará el *middleware* robótico ROS por la compatibilidad entre entornos reales y simulados y porque facilita las comunicaciones entre los diferentes componentes de la aplicación.

1.5. Metodología

Para la realización de este proyecto se ha optado por seguir un modelo de desarrollo en espiral (Figura 1.9) basado en prototipos, ya que los objetivos de este proyecto se ajustan más a este tipo de metodología. El modelo en espiral se basa en la idea fundamental de abordar las tareas iterativamente de forma que en cada iteración se aumenta la complejidad del proyecto, permitiendo la generación de prototipos funcionales al final de cada una. Debido a que se esperaban cambios en los requisitos constantemente típicos en proyectos de investigación, se ha optado por este modelo que permite libertad a la hora de reajustar funcionalidades específicas en cada ciclo, además de hacer el proyecto escalable tanto en funcionalidad como en complejidad.

El modelo en espiral se realiza por ciclos o iteraciones que corresponden a las diferentes fases del proyecto software. Cada ciclo se compone de cuatro fases principales, en las que se realizan diferentes tareas:

- **Determinar los objetivos**. Se definen las necesidades que debe cumplir el *software* a desarrollar en cada iteración, contemplando los objetivos finales. Esto hace que la complejidad del proyecto y el coste del ciclo avancen en función del tiempo.
- **Evaluar alternativas**. Se deben tener en cuenta las diferentes formas de llegar al objetivo proponiendo alternativas a los distintos elementos del proyecto. Además se

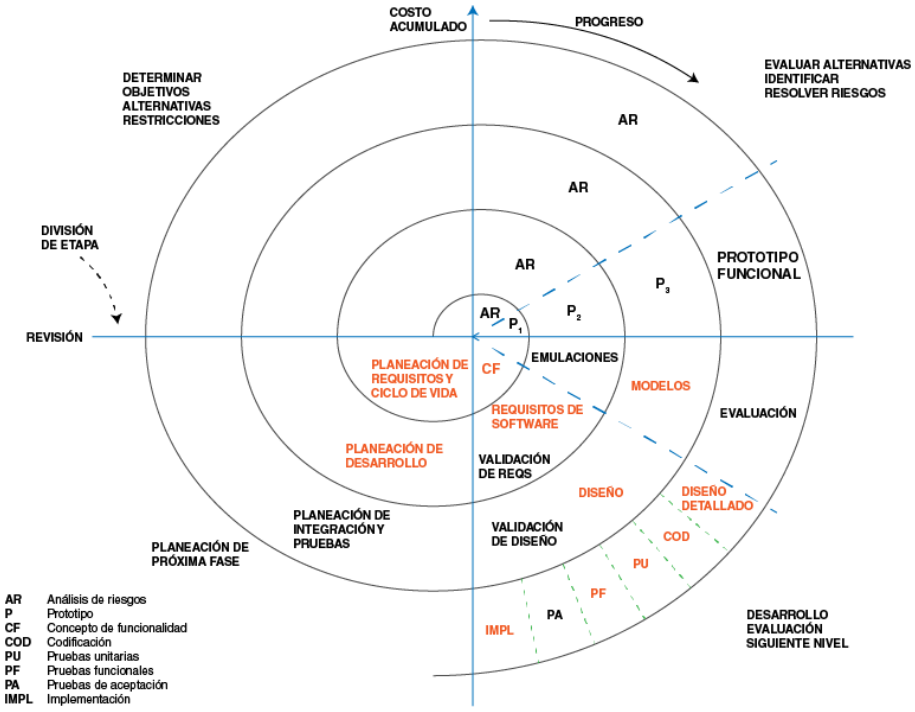


Figura 1.9: Metodología de desarrollo en espiral.

deben considerar los riesgos que puedan existir e intentar reducirlos al máximo.

- **Desarrollar y verificar**. Teniendo en cuenta las alternativas propuestas en la fase anterior, se debe elegir la mejor de ellas y desarrollarla para llegar a los objetivos propuestos, para que finalmente el proyecto de ese desarrollo sea verificado con las pruebas pertinentes.

- **Planificar**. Con los resultados de las pruebas realizadas en la fase anterior, se ha de planificar la siguiente iteración revisando los posibles errores cometidos durante el ciclo actual y se comienza con un nuevo ciclo.

Todo esto sumando a la posibilidad de llevar un control por hitos al final de cada iteración además de las reuniones semanales con el tutor realizadas durante todo el proceso de desarrollo, hacen de este modelo el ideal en nuestro proyecto, dado que cada reunión ofrece realimentación para lo hecho hasta el momento teniendo así continuamente el proyecto bajo control.

Todo el código fuente generado para este proyecto se divide en dos repositorios de Github. El código de entrenamiento de los modelos está en el repositorio de RoboticsLabURJC[10], y el código de la plataforma BehaviorStudio está en su propio repositorio[11]

[10] https://github.com/RoboticsLabURJC/2017-tfm-francisco-perez
[11] https://github.com/jderobot/behaviorstudio

2

Estado del arte

Las siguientes secciones describen algunos de las bases de datos más extendidas y usadas en la conducción autónoma, algunos modelos en redes neuronales exitosos en esta aplicación y algunos computadores especializados para el procesamiento de redes neuronales en tiempo real.

2.1. Bases de datos para conducción autónoma

La tarea de la conducción autónoma requiere que el vehículo sea capaz de tomar decisiones en todo momento ante cualquier escenario posible en las vías por las que circula. Para ello hace uso de la información proporcionada por los sensores a bordo. El sensor más común y que más información aporta a la tarea de la conducción son las cámaras, que es la base de las redes que se explicarán en la siguiente sección. Para controlar todos los posibles escenarios en los que un vehículo puede verse involucrado hace falta un conjunto de datos lo suficientemente grande y representativo de todas ellas, y es por esto que en los últimos años y debido al auge de esta tecnología han aparecido multitud de bases de datos que ayudan a solucionar este problema.

2.1.1. Comma.ai

Comma.ai puso a disposición de la comunidad un conjunto de datos en 2016 [24] destinado a probar modelos de conducción autónoma basados en visión artificial. Este conjunto de datos cuenta con imágenes de vídeo obtenidas a partir de la conducción de una persona a bordo de un coche por entornos reales. En concreto, se dispone de 11 videoclips diferentes con un tamaño de fotograma de 160x320 píxeles, lo que hace un total de 45GB de datos, que se traducen en 7.25 horas de datos de conducción en entornos reales. Todos los fotogramas están etiquetados con información de: velocidad, aceleración, ángulo de giro, coordenadas GPS y ángulos del giroscopio a bordo del vehículo. Todos estos datos están registrados y sincronizados mediante marcas temporales, para que no haya desajuste entre los datos capturados por la cámara y la información de las etiquetas.

Figura 2.1: Ejemplo de capturas del dataset NuScenes en diferentes situaciones climatológicas.

2.1.2. Udacity

Otro conjunto de datos popular para el problema de la conducción autónoma es el desarrollado por Udacity [25]. De forma similar al conjunto de Comma.ai, este *dataset* está formado por pequeños videoclips grabados bajo diferentes condiciones climatológicas: lluvia, sol, niebla. Cuando nació este dataset constaba únicamente de 40GB de datos, lo que suponía un problema debido a la gran cantidad de información necesaria para entrenar modelos que solucionen el problema de la conducción autónoma en el mundo real. Por eso, se decidió ampliar este conjunto añadiendo más datos hasta los 223GB. En total recopila 70 minutos de conducción en entornos reales bajo diversas condiciones. Esta variación en la muestra se traduce en un mejor rendimiento de los modelos entrenados para la conducción autónoma, además de proporcionar información más realista de lo que un sistema que implemente una solución de este tipo se encontraría en el mundo real. Al igual que en el conjunto de Comma.ai, este conjunto de datos está etiquetado con información de velocidad, aceleración, ángulos de dirección, freno, marcha de la palanca de cambios, latitud y longitud.

2.1.3. NuScenes

Este mismo año se ha presentado una nueva base de datos multimodal a gran escala para la conducción autónoma denominada NuScenes [27]. Esta base de datos se ha convertido en relevante al incluir información de toda una gama de sensores que incluye 5 radares, 1 lidar, 6 cámaras, IMU y GPS. NuTonomy scenes (NuScenes) tiene 7 veces más anotaciones y 100 veces más imágenes que el conjunto de datos de KITTI [32] que cubre 23 categorías incluyendo diferentes vehículos, tipos de peatones, dispositivos de movilidad y otros objetos. Este conjunto de datos recopila 1000 escenas de conducción grabadas en Boston y Singapur, las cuales son ciudades conocidas por su gran densidad de tráfico y situaciones complejas en la conducción. Estas escenas duran 20 segundos y están seleccionadas manualmente para mostrar diversas y diferentes maniobras de conducción, situaciones en la circulación y comportamientos inesperados. La mejora que aporta este conjunto de datos es que ofrece solución al desafío de que hay una falta de conjuntos de datos multimodales a gran escala. Esta multimodalidad es crítica, ya que ningún sensor es suficiente por sí solo y los tipos de sensores requieren cierta armonización, cosa de la que adolecen el resto de bases de datos mencionadas. En la Figura **??** se pueden ver algunos ejemplos de este conjunto de datos.

Además, se ha promovido el primer reto de detección de nuScenes que se lanzó en abril

de 2019. Los ganadores y los resultados de los desafíos se anunciaron en el taller sobre conducción autónoma (*nuScenes 3D detection challenge* como parte de *the Workshop on Autonomous Driving at CVPR 2019.*).

2.2. Redes neuronales para conducción autónoma

Para cualquier comportamiento autónomo en robots es necesario un algoritmo que tome decisiones. En el caso de este proyecto, esos algoritmos están basados en redes neuronales y aprendizaje profundo, por lo que en esta sección se describirán algunas redes neuronales aplicadas al problema de conducción autónoma de forma exitosa en los últimos años en la comunidad investigadora.

2.2.1. Redes neuronales convolucionales

El problema de conducción autónoma no es novedoso, y el empleo de aprendizaje de extremo a extremo para solventar dicho problema tampoco, ya que se lleva explorando desde finales de los 80. El proyecto *The Autonomous Land Vehicle in a Neural Network* (ALVINN) [3] fue desarrollado con el objetivo de atacar este problema desde la perspectiva de la visión artificial, al tratar de aprender ángulos de dirección a partir de la información sensorial proporcionada por un láser y un cámara, mediante una red neuronal con una sola capa oculta. Con el aprendizaje extremo a extremo como base, han surgido múltiples aproximaciones como [4] [11] [17], que se detallan a continuación.

La red PilotNet [11] [12] es un ejemplo de red extremo a extremo. Esta red fue creada por NVIDIA y está descrita en el trabajo [11]. Se trata de una red neuronal convolucional (CNN) que recibe una imagen frontal de la cámara del vehículo e infiere comandos de dirección. La medida de error se obtiene mediante la diferencia entre el comando inferido por la red neuronal y el comando deseado (supervisado) para una entrada en concreto, ajustando los pesos de la red en función de ese error mediante la técnica de *back propagation*.

En la Figura 2.2 se ilustra la arquitectura de este tipo de red neuronal. Se puede observar que consta de 9 capas, divididas en 5 capas convolucionales, 3 capas densamente conexas y una capa extra de normalización. La imagen de entrada es preprocesada cambiando su espacio de color a YUV, que será la que alimente a la red tanto para entrenamiento como para inferencia. El diseño de esta red neuronal fue fruto de la experimentación en la que variaban las configuraciones de la diferentes capas. Las dos primeras capas convolucionales usan un *stride* de 2x2 y un *kernel* de 5x5, mientras que las tres últimas capas no usan *stride* y su *kernel* es de 3x3. El diseño de esta red consideraba que la capa densamente conexa fuera el controlador final de la dirección del robot, pero esto no es posible al no disponer de control sobre qué capas de la red funcionan como extractores de características y qué capas funcionan como controladoras. Como en toda red neuronal convolucional, las características se van aprendiendo en el entrenamiento del modelo de forma automática.

El principal objetivo de [12] es tratar de explicar cómo aprende PilotNet y cómo la red realiza su toma de decisiones. Para ello, NVIDIA desarrolla un método para determinar los elementos de la imagen que más peso tienen en la decisión final de la red. Existe un informe detallado de dicho método en [18].

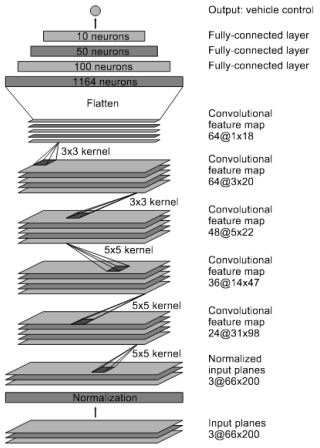


Figura 2.2: Arquitectura Pilotnet.

Figura 2.3: Ejemplos de objetos salientes para varias imágenes de entrada.

Definen objeto saliente como los elementos de la imagen que más influyen en la toma de decisiones de la red, por lo tanto, para encontrar dichos objetos, se centran en localizar las partes de la imagen donde los mapas de características tienen mejores activaciones. En [12] se explica todo el procedimiento y proponen un algoritmo para este fin, que se basa en hacer que las activaciones de los mapas de nivel superior se conviertan en máscaras para las activaciones de niveles inferiores. La Figura 2.3 muestra ejemplos de objetos salientes para tres imágenes de entrada. Se pueden observar las regiones de la imagen que contribuyen más a la salida de la red.

Todas estas técnicas hacen que PilotNet aprenda a reconocer objetos que aportan información útil de la carretera mejorando su toma de decisiones y haciendo que la red sea capaz de realizar una conducción autónoma segura manteniendo el coche en el carril bajo multitud de situaciones, tanto si las líneas que demarcan el carril están presentes como si no.

Derivada de la red explicada anteriormente (PilotNet), surge una nueva arquitectura denominada TinyPilotNet propuesta en [23]. Esta nueva red, que se puede observar en la Figura 2.4, está compuesta por 6 capas dispuestas como sigue: la capa de entrada que se

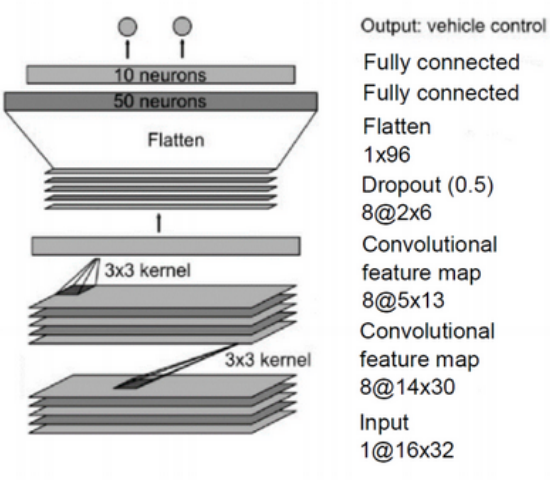


Figura 2.4: Arquitectura TinyPilotnet.

alimenta de imágenes de resolución 16x32 píxeles con un único canal de color en espacio HSV, las siguientes dos capas son dos capas convolucionales configuradas con un *kernel* de 3x3 y una capa de *dropout* al 50 % de probabilidad; la salida de estas capas produce un tensor que se convierte en un vector que alimentará las dos últimas capas densamente conexas con cabezal clasificador biclase, para predecir los valores de dirección y aceleración.

En [22] se propone un método para predecir el ángulo de giro de los vehículos: emulación de cámaras de eventos. Las cámaras de eventos son sensores que obtienen información de píxel de forma asíncrona a través de cambios en intensidad de la luz. Estos eventos se transforman en fotogramas mediante acumulación de píxeles en un intervalo constante de tiempo, lo que genera una imagen únicamente de eventos en 2D en un intervalo dado, que serán procesadas por una red neuronal profunda para asignar los ángulos de dirección. Las redes empleadas para el procesamiento de estas imágenes son redes con arquitectura residual, las ResNet; en concreto las redes ResNet18 y ResNet50. Se utilizan principalmente como extractores de características en problemas de regresión teniendo en cuenta únicamente las capas convolucionales. La codificación de las características obtenidas en la última capa convolucional en forma de vector, se lleva a cabo mediante una capa de *global average pooling* que retorna la media del canal de las características extraídas, para posteriormente agregar una capa densamente conexa, con activación ReLu y otra capa densamente conexa unidimensional para inferir el ángulo.

En [22] se infieren los ángulos emplenado tres tipos de entradas:

- Imágenes en escala de grises
- Diferencia de imágenes en escala de grises
- Imágenes creadas por acumulación de eventos.

Se analiza el rendimiento de los modelos midiendo el tiempo de integración empleado para la generación de imágenes de eventos. A mayor cantidad de tiempo, mayor cantidad de eventos se capturan en los contornos de los objetos. Empíricamente se ha comprobado que un lapso de tiempo de $50ms$ para la generación de la imágenes de eventos, el rendimiento de

la red mejora, ya que tiempos más grandes degradan el rendimiento de la red al obtenerse imágenes más grandes. Como desventaja, si se emplean imágenes en escala de grises, a altas velocidades las imágenes se difuminan y se vuelven muy ruidosas.

En el trabajo [21] se lleva a cabo un estudio donde se analiza el rendimiento de una red neuronal extremo a extremo para la conducción autónoma en base a las imágenes capturadas por el vehículo. Además se realiza un estudio comparativo entre las redes de clasificación y regresión, y el impacto que tienen las dependencias temporales de imágenes consecutivas. Parten de una variación de las arquitecturas PilotNet, VGG o Alexnet realizando modificaciones sobre ellas. Uno de los estudios de este trabajo es el impacto del grado de granularidad del número de clases de salida en el rendimiento del sistema. Además se evalúan los métodos que permiten la inclusión de la temporalidad en las imágenes: qué impacto tiene la inclusión de entradas consecutivas en el sistema a través del empleo de arquitecturas extremo a extremo con capas recurrentes.

El método *stacked frames* consiste en el apilado de imágenes temporalmente consecutivas para la creación de una imagen apilada. Sea I_t un fotograma en el instante t, este método apila además los fotogramas I_{t-1}, $I_{t-2}, etc.$ que servirán como entrada a la red. El formato de los vectores resultantes de este apilado crece en en la dimensión del canal (en profundidad), así, una imagen I_T con dimensiones $224x224x3$ con las imágenes I_{t-1} e I_{t-2} apiladas, tendría unas dimensiones de $224x224x9$. Con la aplicación de este método se demuestra que el rendimiento de la red mejora. La conclusión obtenida es que esta mejora puede deberse a que la red puede tomar una decisión basada en la información promedio de múltiples fotogramas apilados. Si se utiliza una sola imagen, se carece de información contextual de lo que el vehículo ha percibido anteriormente, por lo que las predicciones pueden variar en gran medida. Sin embargo, con éste método, la información contextual puede servir para contener predicciones extremas al poder cancelarse la información entre sí. No obstante, esto tiene un límite, ya que apilar demasiadas imágenes puede provocar que la red pierda poder de generalización deteriorando su rendiminto; por ello se apilan únicamente tres fotogramas.

2.2.2. Redes neuronales recurrentes

Las redes neuronales recurrentes (RNN) presentan una técnica novedosa en cuanto al modelado de las relaciones temporales de los datos de entrada. Para ello emplean lo que denominan células de memoria. No obstante estas células de memoria están limitadas a relaciones temporales de corto alcance, por lo que con la inclusión de las *Long Short-Term Memory* más comúnmente denominadas LSTM, que modelan relaciones temporales de largo alcance, se solventa este inconveniente.

Estas redes son especialmente útiles cuando los datos tienen el componente de la temporalidad, es decir se trata de secuencias de datos relacionadas en el tiempo. En múltiples estudios se ha hecho uso de la capacidad de estas redes para aprovechar la información secuencial para investigaciones sobre conducción autónoma.

En [15] se emplean este tipo de arquitecturas recurrentes, más concretamente las LSTM. Fruto de este estudio, se obtiene un controlador reactivo basado en aprendizaje profundo mediante una arquitectura de red neuronal muy sencilla que requiere pocas imágenes para realizar un entrenamiento efectivo. Esta arquitectura es bautizada como

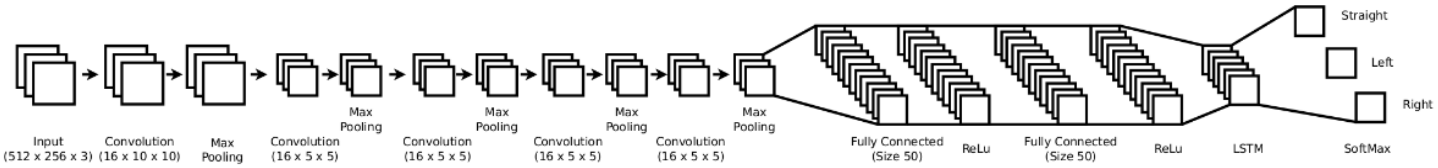


Figura 2.5: Estructura de red ControlNet.

ControlNet, y supera a otras redes más complejas tanto en su aplicación en entornos interiores y exteriores haciendo uso de diferentes plataformas robóticas. Esta red se alimenta de imágenes en color (RGB) para realizar inferencia sobre ellas y obtener comandos de control. La arquitectura de esta red, que se puede observar en la Figura 2.5, implementa capas convolucionales (para extraer información sobre características de las imágenes de entrada) con sus capas de *pooling* y capas densamente conexas (que actúan como clasificadores) alternas. Además, como se ha explicado, implementa una capa LSTM que incorpora información temporal, permitiendo que el vehículo pueda continuar con la misma dirección sobre fotogramas consecutivos similares.

En [13] se crea la denominada *Convolutional Long Short-Term Memory Recurrent Neural Network*, más conocida conocido como C-LSTM que se muestra en la Figura 2.6. Este tipo de red neuronal se entrena extremo a extremo con el objetivo de que aprenda las dependencias temporales y visuales involucradas en la tarea de la conducción autónoma. Este sistema está compuesto de dos tipos de redes: las CNN y las LSTM y se alimenta de las imágenes capturadas por la cámara a bordo del vehículo para realizar inferencia del ángulo de giro en base a la información sensorial. Las características de la imagen se obtienen a través de la CNN fotograma a fotograma para aprender las dependencias visuales. Estas características son entonces procesadas por la LSTM para aprender las dependencias temporales. Al final de esta arquitectura existe un cabezal clasificador que predice el ángulo de giro del vehículo. En este sistema se aplica el concepto de *transfer learning*, cuya idea fundamental se basa en realizar un entrenamiento adicional sobre una red pre-entrenada sobre un dominio concreto totalmente distinto del que fue entrenado. En este caso, la CNN está pre-entrenada en el conjunto de datos de Imagenet [14], que es un conjunto de datos de 1,000,000 imágenes anotadas de objetos cotidianos agrupados en 1,000 clases diferentes. Los pesos entrenados de esa CNN sobre ImageNet son transferidos a otra red de dominio específico para imágenes de conducción, donde se reentrena partiendo de los pesos ya aprendidos en el primer entrenamiento. Posteriormente, la LSTM aprende las dependencias temporales de los vectores de características obtenidos por la CNN para tomar una decisión.

Un trabajo similar se propone en [16]. En este caso se propone un modelo de *deep learning* que a partir de imágenes captadas por una cámara a bordo del vehiculo, se infieren ángulos de dirección dividiéndolo en dos subredes: una subred extractora de características y otra subred de predicción de dirección.

La subred extractora de características genera un vector de longitud fija (128 características) que modela la información sensorial y el estado del vehículo. Para esto, emplea una capa denominada *spatio-temporal convolution* o ST-Conv y otra capa LSTM denominada ConvLSTM que modifican las dimensiones espacio-temporales de las imágenes de entrada. Posteriormente, se emplea una capa densamente conexa para obtener el vector de

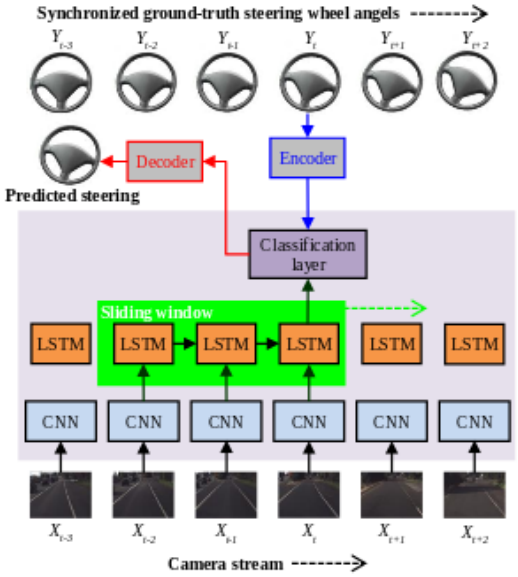


Figura 2.6: Arquitectura C-LSTM.

128 características mencionado. Este vector alimenta a la siguiente subred de predicción de dirección.

La subred de dirección trata de concatenar la información de 3 fuentes: el vector de la subred extractora de características, la información del estado del vehículo y las acciones de dirección posibles. Para esto se incluye un paso de recurrencia entre la capa de salida y antes y después de las capas «concatenadoras» llamadas *concat*. Se tiene que:

- La capa *concat* que precede a la capa LSTM agrega 3 características más al vector de 128 inicial: la velocidad, el ángulo de la rueda y el par de torsión.

- La capa *concat* que va después de la capa LSTM está compuesta de un vector de características por un vector de 195 características: 128 de la primera subred + 64 de la LSTM + 3 de la capa *concat* anterior.

Por su parte, en [17] se propone un trabajo similar a los anteriores al introducir un modelo de atención visual para el entrenamiento extremo a extremo de una red neuronal convolucional. El modelo de atención tiene en cuenta las zonas de la imagen que más afectan a la inferencia potencialmente. Este modelo predice órdenes de dirección a través de píxeles crudos, teniendo como salida el radio de giro inverso $\hat{u}_t$, que se relaciona con el ángulo de dirección utilizando la geometría de Ackermann.

Este modelo en concreto, al igual que los anteriores, se ayuda de las CNN para la extracción de un vector de características codificadas, a las que llaman características convolucionales cubo x_t, a partir de las imágenes de entrada captadas por la cámara del vehículo. Cada uno de esos vectores de características puede contener información descriptiva de los objetos de la imagen (a alto nivel) que permite una atención selectiva de diferentes zonas de la imagen. Para llevar a cabo este trabajo, hacen uso de la red PilotNet (explicada al principio de esta sección) para conseguir un modelo de conducción. La particularidad de este modelo es que la información espacial de la imagen es muy importante, por lo que omiten las capas de *pooling* de la red para evitar la pérdida de esa

información. Con esto, se genera un cubo x_t tridimensional de características que pasa a través de las capas LSTM que estiman el $\hat{u}_t$ (radio de giro inverso). Como elemento final de este método se incluye un decodificador que es alimentado con un mapa de atención visual y ofrece como salida las detecciones de saliencias visuales locales.

2.3. *Hardware* acelerador de cómputo neuronal

Hasta ahora se han visto dos elementos esenciales del problema de la conducción autónoma mediante redes neuronales: las propias redes neuronales y los conjuntos de datos con los que se entrenan. Debido al carácter de los datos a procesar, donde se tiene información sensorial proveniente de cámaras (de color, de profundidad, de lineales, etc.) que ofrece una enorme cantidad de datos en cada imagen captada, es indispensable una alto poder computacional para procesar todos esos datos de forma eficiente y en tiempo real.

La conducción autónoma es una tarea en la cual el tiempo real es un aspecto crítico, ya que se trata un problema cuya solución ha abordarse desde una perspectiva de control híbrida (planificación + reacción) donde se requiere que la toma de decisiones por parte del algoritmo sea muy rápida para poder reaccionar de forma instantánea ante la gran cantidad de escenarios imprevistos que se pueden suceder durante la conducción (peatones que cruzan sin mirar, accidentes de tráfico, animales que cruzan la vía, etc.). No obstante, no todos los vehículos pueden estar dotados de computadores de grandes dimensiones como podría ser un computador de escritorio debido a las limitaciones tanto de alimentación como de espacio. Por este motivo, algunas empresas como Google o NVIDIA han creado *hardware específico* para solventar este problema y ya no solo para aplicarlo a la conducción autónoma, sino también a cualquier problema domótico o, como tecnología en auge, IoT (*Internet of Things*).

En esta sección se describirán diferentes dispositivos diseñados para ser suficientemente potentes al ejecutar una red neuronal pesada y ocupar el mínimo espacio físico posible con el objetivo de empotrarlo en algún sistema.

2.3.1. NVIDIA Jetson TK1

En 2014, NVIDIA presentaba su primera placa para sistemas embebidos de alto rendimiento y bajo consumo, denominada Jetson TK1 [34]. Esta placa de dimensiones muy reducidas (127mm x 127mm) cuenta con un procesador ARM de 32-bits con 2 núcleos y una GPU con arquitectura Kepler que implementa 192 núcleos CUDA, ambos incluidos en el SoC (*System-on-Chip*) Tegra K1 [38], además de una memoria RAM de 2 GB con tecnología DDR3. La presentación del procesador Tegra K1 fue un todo un hito, ya que fue el primer procesador móvil que igualaba en rendimiento a sistemas de escritorio con un bajísimo consumo de energía de 12,5 W a máxima carga.

Se han realizado algunos bancos de pruebas de esta placa probando su rendimiento con la ejecución de algunas redes neuronales modernas desarrolladas para sistemas embebidos, como *mobilenet*, *googlenet* o *shufflenet* entre otras. Como referencia, en la Figura 2.7 se tiene este mismo banco de pruebas medido sobre una tarjeta gráfica de alta gama para

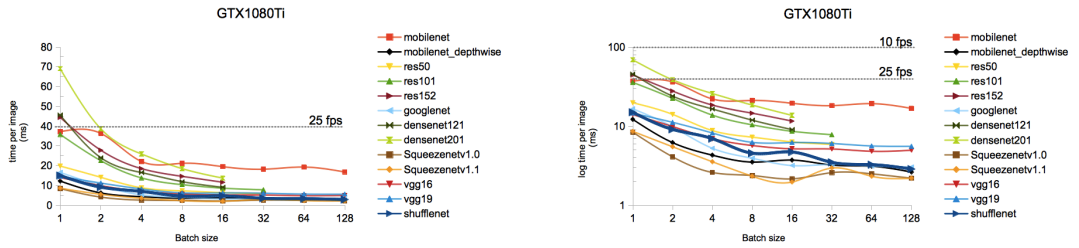


Figura 2.7: Banco de pruebas de referencia de la NVIDIA GTX 1080 de escritorio. (Izquierda) tiempo de procesamiento lineal. (Derecha) tiempo de procesamiento en escala logarítmica.

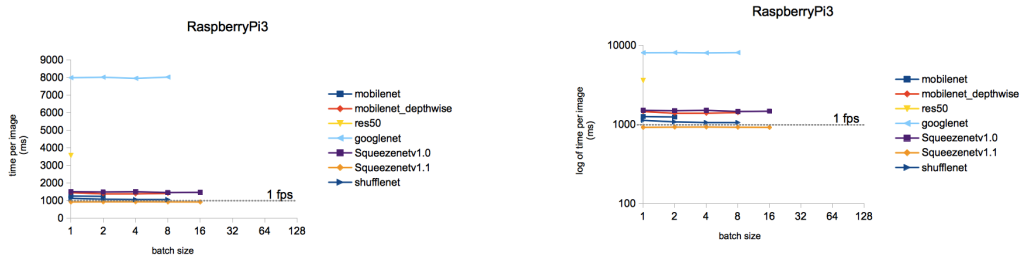


Figura 2.8: Banco de pruebas de referencia de una Raspberry Pi3. (Izquierda) tiempo de procesamiento lineal. (Derecha) tiempo de procesamiento en escala logarítmica.

sistemas de escritorio, la NVIDIA GTX 1080Ti como referencia techo, y en la Figura 2.8 la referencia suelo de este mismo banco de pruebas llevado a una RaspberryPi modelo 3.

Los resultados se miden como la media de tiempo (en *ms*) que tarda el SoC TK1 en procesar una imagen frente a un tamaño de lote determinado de la red neuronal. Como se puede observar en la Figura 2.9 (izquierda), la red más lenta es la *mobilenet* con un tiempo de procesamiento por imagen de casi medio segundo (500 *ms*), mientras que la red más rápida es la *shufflenet v1.1* con un tiempo de procesamiento por debajo de los 100 *ms*, lo que aporta una tasa de fotogramas por segundo relevante. En la Figura 2.9 (derecha) se puede apreciar con mayor claridad el rendimiento de esta placa con diferentes redes neuronales.

No obstante, si se compara esta placa con una RaspberryPi 3, los resultados son significativos, ya que su rendimiento es más del doble con respecto a la Raspberry. Y si se compara por arriba con la GTX 1080, no está muy lejos de los mínimos de esta, por lo que el poder computacional con respecto a la energía consumida es notable. Todo esto se puede ver reflejado en la Figura 2.10

2.3.2. NVIDIA Jetson TX1

En 2015, NVIDIA dio un paso adelante con su nueva placa para sistemas empotrados: la Jetson TX1 [35]. Esta placa de dimensiones aún más reducidas que su predecesora (50mm x 87mm) cuenta con un procesador ARM de 64-bits con 4 núcleos y una GPU con

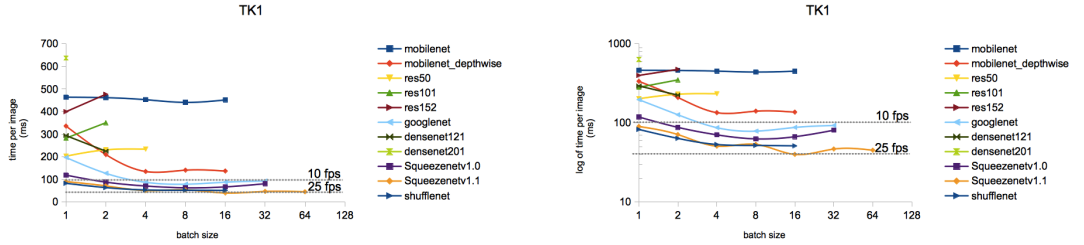


Figura 2.9: banco de pruebas para la NVIDIA Jetson TK1. (Izquierda) tiempo de procesamiento lineal. (Derecha) tiempo de procesamiento en escala logarítmica.

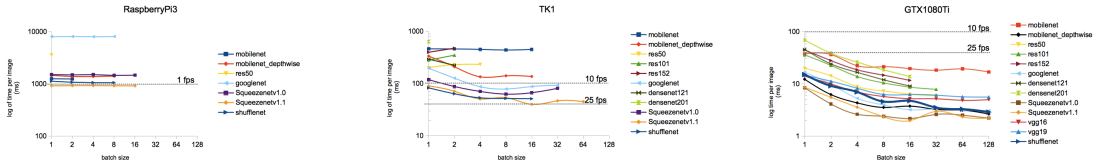


Figura 2.10: Comparativa entre RaspberryPi 3 (izquierda), Jetson TK1 (centro) y GTX 1080Ti (derecha).

arquitectura Maxwell, ambos incluídos en el SoC (*System-on-Chip*) Tegra X1, además de una memoria RAM de 4 GB con tecnología LPDDR4. Este módulo supuso un salto en capacidad de cómputo ya que el procesador dobla en poder al de la Jetson TK1 tanto en CPU como en GPU. Este salto en potencia también se ve reflejado en el consumo, ya que a máxima carga, esta placa consume 15 W frente a los 12.5 W de la TK1.

En la Figura 2.11 se reflejan los resultados de los bancos de pruebas aplicados a esta placa. Comparando con los resultados de la placa TK1 (Figura 2.9), a simple vista se puede percibir que el salto en rendimiento y potencia es evidente. Mientras que la Jetson TK1, no es capaz de ofrecer tasas de refresco mayores que 25 fotogramas por segundo, la Jetson TX1 supera ese umbral en más de la mitad de las redes probadas. Y las redes más lentas como la *densenet201* obtienen una mejora de rendimiento significativa en la placa TX1. Comparada con la GTX 1080Ti (Figura 2.7), se puede ver que el rendimiento de la TX1 se acerca mucho a los resultados obtenidos por la 1080Ti. Hay que tener en cuenta que la placa Jetson TX1 cuenta con una tecnología propietaria de NVIDIA denominada Maxwell, cuyo chip implementa 256 núcleos CUDA; mientras que la GTX 1080Ti cuenta con una tecnología más avanzada, también propietaria de NVIDIA, denominada Pascal y cuyo chip implementa 3584 núcleos CUDA.

2.3.3. NVIDIA Jetson Nano

En 2019 NVIDIA presentó sus nuevas placas de precio y prestaciones reducidas denominadas Jetson Nano [36]. Estas nueva placas siguen la línea de sus hermanas con un tamaño reducido (80mm x 100mm) pero con precio y prestaciones más bajos, aunque muy potentes y optimizadas para inteligencia artificial. Cuenta con un procesador ARM de 64-bits con 4 núcleos y una GPU basada en tecnología Maxwell (como sus hermana Jetson TX1) con la mitad de núcleos CUDA (128). Este módulo tiene dos modos de consumo:

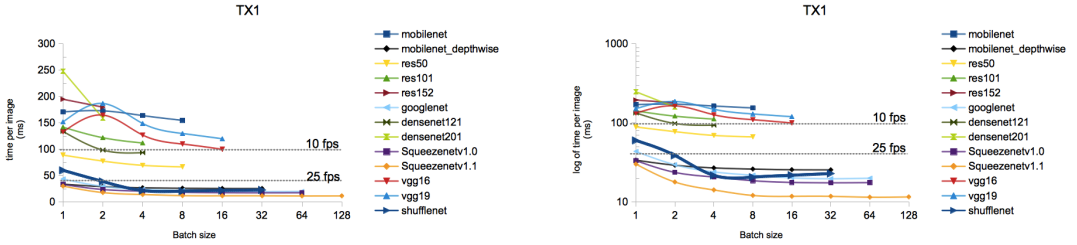


Figura 2.11: banco de pruebas para la NVIDIA Jetson TX1. (Izquierda) tiempo de procesamiento lineal. (Derecha) tiempo de procesamiento en escala logarítmica.

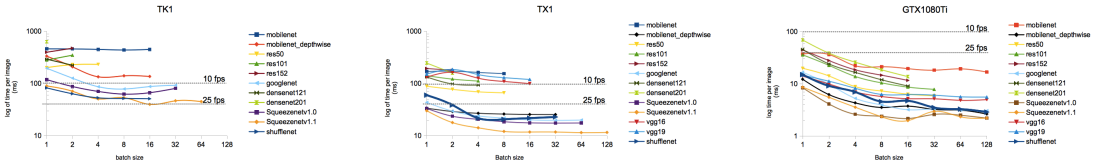


Figura 2.12: Comparativa entre Jetson TK1 (izquierda), Jetson TX1 (centro) y GTX 1080Ti (derecha). Escala logarítmica.

modo normal (5W) y carga máxima (10W), lo que lo hace el más liviano de toda la gama Jetson, además tiene un precio reducido con respecto al resto de placas de la familia. Cabe destacar que el modo de consumo normal desactiva parte del hardware de la placa, ya que apaga 2 núcleos del procesador para ahorrar energía. No obstante, a máxima potencia, ofrece el máximo rendimiento hardware.

En la Figura 2.13 se tiene el resultado de los tests en la Jetson Nano. Comparado con su hermana la Jetson TX1 (Figura 2.11) se puede apreciar que el rendimiento es ligeramente más bajo debido a que tiene la mitad de núcleos en su CPU y su frecuencia de trabajo es más baja. No obstante, si se compara con la Jetson TK1 (Figura 2.14 (izquierda)), se puede comprobar que la supera en rendimiento en todas las redes probadas. Hay que tener en cuenta, en este caso concreto, el consumo de ambas placas y el número de núcleos de GPU de ambas. Mientras que la Jetson TK1 cuenta con 192 núcleos CUDA en su GPU, la Jetson Nano cuenta con 128 núcleos. Además, el consumo de la Jetson TK1 es más del doble que la Jetson Nano. Con todo eso, esta placa es superior a la TK1 en cuanto a ratio rendimiento/consumo. Para la misma red *mobilenet*, la placa Nano bate en rendimiento a la TK1 acercándose a una tasa de fotogramas por segundo de 10, mientras que la TK1 permanece constante muy alejada de esos valores.

2.3.4. Google TPU *Tensor Processing Unit*

Google ha puesto a la venta su propio *hardware* acelerador de cómputo, que hasta hace pocos meses no ofrecía. En su lugar, ofrecía servicios de computación en la nube como *Google Colab*, en el que de forma gratuita se tiene acceso a un servidor para realizar cálculos en la nube con esta tecnología TPU que acelera en gran medida los tiempos de aprendizaje de las redes neuronales. La Figura 2.15 muestra una unidad de TPU de Google.

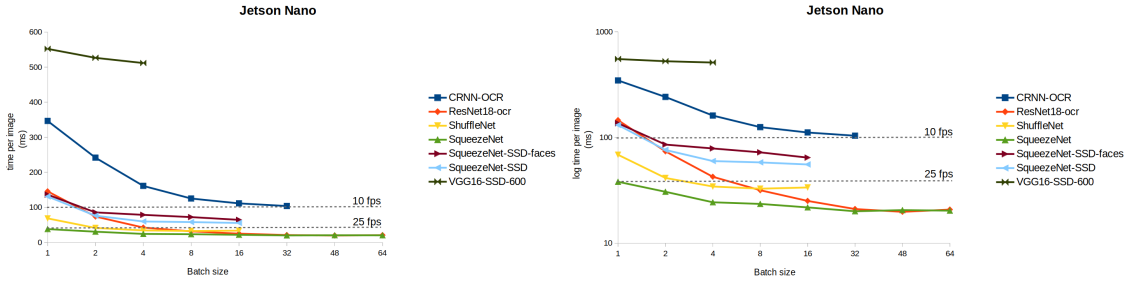


Figura 2.13: banco de pruebas para la NVIDIA Jetson Nano. (Izquierda) tiempo de procesamiento lineal. (Derecha) tiempo de procesamiento en escala logarítmica.

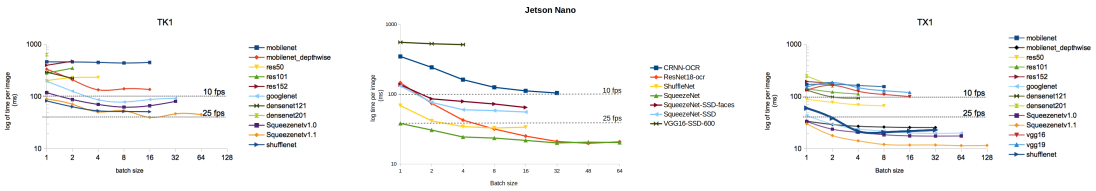


Figura 2.14: Comparativa entre Jetson TK1 (izquierda), Jetson Nano (centro) y Jetson TX1 (derecha). Escala logarítmica.

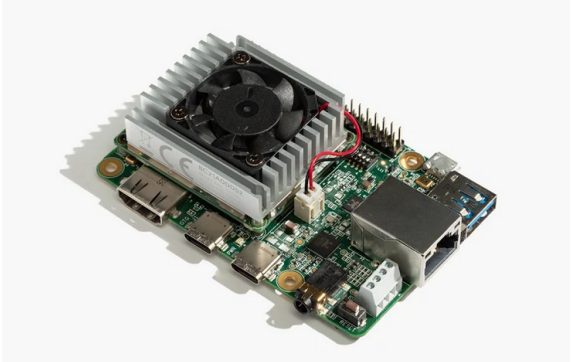


Figura 2.15: TPU de Google.

En 2018, Google presentó su propio *hardware* acelerador de cómputo especializado en cálculo matricial y al que denominó TPU [41]. Estas siglas, similares a las de GPU (*Graphic Porcessing Unit*), vienen de *Tensor Processing Unit* o Unidad de procesamiento de tensores. Un tensor no es más que una matriz multidimensional. La ventaja de las TPU es que pueden realizar cálculos en paralelo a una gran velocidad y, dado su carácter optimizado para el cálculo matricial (álgebra lineal), son ideales para la ejecución algoritmos de aprendizaje automático para visión artificial al ser la información que procesan estas redes matrices (imágenes).

No obstante, esta tecnología está pensada para ser utilizada en determinados ámbitos de trabajo concretos, de tal forma que en cómputo de carácter más general es más recomendable utilizar CPU o GPU.

- CPU. Recomendado para realizar prototipos rápidos que requieran mucha flexibilidad. Entrenar modelos sencillos, pequeños y limitados.
- GPU. Modelos con un número notable de operaciones y con tamaños de lote grandes.
- TPU. Modelos de cómputo de matrices, con amplio tiempo de entrenamiento. Modelos muy grandes.

3

Infraestructura

En este capítulo se explican las diferentes herramientas software utilizadas para el desarrollo del proyecto, para situal el marco tecnológico del mismo. Se detallan los diferentes componentes *hardware* de los que se ha hecho uso en este proyecto, así como las herramientas de *software* utilizadas. La decisión de la infraestructura a usar estuvo estrechamente relacionada con el estudio realizado del estado del arte de la sección anterior. También se enumeran y explican superficialmente todas las herramientas *software* de las que se ha hecho uso durante el desarrollo del proyecto. Cada sección describirá brevemente el componente en cuestión y el uso que se le ha dado en el proyecto. Se ha utilizado exclusivamente *software* de código abierto.

3.1. NVIDIA Jetson Nano

El objetivo de este proyecto es que un robot doméstico pueda ser capaz de navegar autónomamente por un circuito real sin salirse. Para ello es necesario el uso de hardware embebido (tamaño pequeño y gran capacidad de cómputo) sobre el que se ejecute la aplicación. Por ello, este proyecto se desarrolla usando un SoM dedicado: la placa Jetson Nano de NVIDIA. Este sistema cuenta con una GPU de alto rendimiento y motores de optimización de bajo nivel que reducen enormemente el tiempo necesario para realizar las operaciones requeridas para las aplicaciones de aprendizaje profundo, como las convoluciones de los tensores.

El bajo consumo de esta placa (10W a plena potencia, 5W a potencia normal), junto con su pequeño tamaño, la hace apta para ser incorporada en un robot portátil equipado con una batería. Dado que los modelos y los conjuntos de datos que utilizamos para este proyecto son relativamente pequeños, el tamaño del almacenamiento no es un problema. Esta placa cuenta con un procesador ARM de 64 bits, y puede alojar un sistema Linux completamente funcional a través de las imágenes JetPack disponibles (sección 3.3 para más información). Al estar equipada con dos antenas Wi-fi conectadas a un módulo dedicado para comunicaciones, se puede interactuar con el sistema a través de conexiones SSH. Con respecto a la RAM disponible en la placa, está limitada a 4 GB compartida por la GPU y la CPU, lo que puede poner en peligro la ejecución del software desplegado y las redes neuronales que deben ser controladas en todo momento para ahorrar la máxima cantidad de RAM posible. En la práctica, se han tenido que desactivar algunos servicios del sistema para que la placa fuera más desahogada en cómputo. La placa Jetson Nano puede ser visualizada en la Figura 3.1

Figura 3.1: NVIDIA Jetson Nano

Figura 3.2: JetBot kit de Waveshare

3.2. Robot JetBot

La placa de cómputo por sí sola no es suficiente para el desarrollo de este proyecto. Necesitamos un soporte que pueda ser controlado por el cerebro de la aplicación para la resolución del problema; en otras palabras, un robot.

Hay variedad de opciones disponibles en el mercado, con kits robóticos que incluyen todo el hardware necesario para ensamblar un robot a partir de las piezas disponibles en el kit, así como la posibilidad de imprimir tus propios componentes y construir un robot desde cero utilizando una impresora 3D para ello. En este caso, se ha optado por utilizar un kit de alta calidad, que incluya todas las piezas, sensores y actuadores necesarios, además de hardware auxiliar, para facilitar la tarea y ahorrar tiempo en ensayo/error de diseño y ensamblaje. El kit escogido ha sido el proporcionado por la empresa Waveshare, que se puede ver en la Figura 3.3. Esta empresa es una tienda de componentes electrónicos, servicios y soluciones. Entre sus productos se encuentra el kit robótico denominado *JetBot AI Kit*[1] que incluye todas las piezas necesarias para la construcción de un robot pequeño, perfecto para el propósito de este proyecto.

Se ha escogido este modelo en concreto por que está diseñado para montar una placa Jetson Nano (es el kit recomendado por NVIDIA), que es la que se ha elegido para este proyecto, además de por su calidad de materiales. El chasis está fabricado en una aleación de aluminio con acabado cepillado, incluye llantas de plástico prensado y neumáticos de goma con dibujo para un mejor agarre. El kit incluye un controlador para los motores, con *sockets* para baterías que hace que el cableado sea trivial, no teniendo que lidiar con PCBs ni cables sueltos. Además, ofrece una cámara con un soporte en ángulo, un módulo wi-fi con dos antenas, un par de motores, ventilación, un joystick inalámbrico para la teleoperación, y herramientas para llevar a cabo el ensamblado del robot. Como único punto negativo, cabe destacar que la calidad de los motores no es la mejor, ya que carecen de par suficiente para un ajuste fino en velocidades bajas.

A continuación se detallan los componentes principales que componen este kit.

3.2.1. Cámara

El sensor que incluye la cámara de este kit es un IMX219-160. Este sensor es muy similar en características al de la cámara de la RaspberryPi (la PiCam). La Tabla 3.1

[1] https://www.waveshare.com/jetbot-ai-kit.htm

condensa las especificaciones técnicas de esta cámara.

Tamaño	25 mm (W) x 24 mm (L)
Sensor	Sony IMX219
Resolución	3280 (H) x 2464 (V)
FOV	160º
Distancia focal	3.15 mm
Apertura	2.35

Tabla 3.1: Especificaciones de la cámara

Como se observa en la Tabla 3.1 esta cámara dispone de una resolución de 3280x2464 píxeles (8 megapíxeles) con un FOV (*Field of view*) de 160º debido a su gran angular. Puede trabajar a alta tasa de imágenes por segundo en función de la resolución a la que se trabaje, en concreto: 3280 x 2464 a 21 fps, 3280 x 1848 a 28 fps, 1920 x 1080 a 30 fps, 1280 x 720 a 60 fps, 1280 x 720 a 60 fps.

3.2.2. Motores

El kit cuenta con un par de motores como actuadores principales del robot. En este caso concreto, estos motores DC son genéricos y no de demasiada calidad. Estos motores han sido una fuente de problemas durante todo el desarrollo del proyecto, ya que su falta de par, ha hecho casi imposible un control fino del robot a bajas velocidades.

3.2.3. Placa controladora

Este componente, que se puede observar en la Figura 3.3 es el núcleo de este kit, ya que incorpora varios chips que ofrecen diferente funcionalidad, además de eliminar el trabajo de cableado casi al completo. La placa puede ser alimentada de dos formas: mediante baterías del tipo 18650 o mediante un adaptador de corriente de 12V. El zócalo de las baterías permite la carga de las mismas mientras la placa esté conectada a la corriente. Esta placa alimenta también a la Jetson Nano, mediante un regulador de voltaje APW7313, que provee a la Jetson de 5V estables para su funcionamiento.

La función principal de esta placa es controlar los motores del robot, a través del protocolo I2C. Cuenta con conectores del tipo JST para ambos motores, que van directamente conectados a la placa. El chip controlador es un TB6612FNG dual H-bridge diseñado para trabajar con pequeños motores DC. El rango de voltajes soportado por este chip oscila entre los 4.5V y los 13.5V.

La placa cuenta además con una pantalla OLED de 0.91 pulgadas de una resolución de 128x32 píxeles, que muestra información sobre el robot como la dirección IP del robot, memoria RAM, tiempo de vida de batería, energía, etc. Esta pequeña pantalla resulta muy útil para comprobar de forma rápida el estado del robot sin necesidad de acceder a una terminal dentro del mismo. Para monitorizar la batería en tiempo real se utiliza el chip ADS1115 AD.

Figura 3.3: Placa controladora del JetBot

3.2.4. Módulo Wi-Fi

Otro componente principal de este kit es el módulo Wi-Fi, en este caso un módulo Intel AC-8265 de doble banda: 2.5 GHz y 5 GHz. En la Tabla 3.2 se resumen las especificaciones.

Bandas	2.4 GHz, 5 GHz
Velocidad máxima	867 Mbps
Certificado	802.11 ac
Bluetooth	Sí

Tabla 3.2: Especificaciones del módulo Wi-Fi

La inclusión de bluetooth del módulo lo hace especialmente útil para conectar periféricos como teclados, ratones o incluso joysticks para la teleoperación del robot. Adicionalmente, el kit de Waveshare cuenta con 2 antenas para ampliar el alcance de la señal, útil en exteriores o en entornos muy grandes donde el robot pueda estar alejado del computador.

3.2.5. Joystick

Como último aporte, este kit cuenta con un joystick inalámbrico para poder teleoperar el robot. Dado que el módulo inalámbrico que incluye el kit soporta conexiones bluetooth, el joystick se puede conectar al mismo mediante un *dongle* que viene incluido. Esta herramienta ha resultado útil en los primeros compases del proyecto para probar la movilidad del robot de forma sencilla. En la Figura 3.4 se puede ver este joystick.

3.2.6. Pistas de Lego

Como último elemento de este proyecto se han utilizado pistas de Lego que han hecho las veces de circuito sobre el que el robot debía navegar (Figura 3.5). Debido al carácter modular de Lego, el intercambiar la forma de las pistas es tarea trivial. Es por ello que

Figura 3.4: Joystick de JetBot

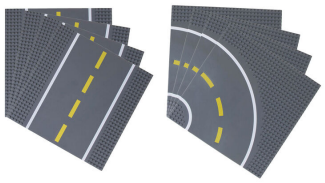


Figura 3.5: Módulos de lego

se decidió utilizar este elemento como pista para poder probar la solución del proyecto en diferentes formatos de circuito.

3.3. NVIDIA JetPack

NVIDIA ofrece una gama de placas de alto rendimiento para sistemas empotrados, debido tanto a los avances en IA como en la necesidad cada vez más creciente de minimizar el tamaño de los computadores en diversos terrenos como, por ejemplo, el de la conducción autónoma. La NVIDIA Jetson Nano[2] es una de estas placas ya que está diseñada para trabajar como un computador de alto rendimiento en sistemas empotrados. En la sección 2.3.3, se explica con relativo detalle las características de esta placa. Toda la gama, Jetson Nano, TX1, TX2, Xavier series, etc, siguen unas pautas de diseño muy optimizadas. La Jetson Nano en concreto implementa una versión adaptada de Ubuntu Linux, llamada NVIDIA JetPack que es desarrollada y mantenida por NVIDIA, y está disponible para ser descargada e instalada como el *firmware* de la placa.

Para el sistema desarrollado la versión utilizada es JetPack 4.3; más precisamente, la imagen utilizada es una rama del JetPack 4.3 optimizada para el robot Jetbot (con la placa Jetson Nano). Esta implementación incluye interfaces de bajo nivel para implementar operaciones de computación en paralelo (usando CUDA), y varios SDKs de optimización (kits de desarrollo de software), como TensorRT, junto con interfaces para operar con los sensores y actuadores del robot. Además, contiene toda la infraestructura necesaria a nivel de software como CUDA, Pytorch, Python, etc. que mejor compatibilidad tienen con la versión de JetPack que se utilice. Este motor *software* es de especial interés para este desarrollo, ya que permite aumentar enormemente la velocidad de inferencia sin perder precisión, gracias a sus optimizaciones.

NVIDIA JetPack se ha usado como SO (sistema operativo) base para la placa controladora Nano. Haber utilizado una distribución de Ubuntu (aunque fuera liviana) habría acarreado problemas de gestión de recursos como la CPU o la memoria a la hora de hacer inferencia en tiempo real, como se ha comprobado en la práctica.

3.4. *Middleware* robótico

Este proyecto se desarrolla haciendo uso del *middleware* ROS (*Robot Operating System*). Dada la necesidad de comunicación entre el robot y la plataforma BehaviorStudio,

[2]https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit

esta herramienta facilita la interconexión entre el *hardware* y el *software*. ROS[3] es un meta-sistema operativo de código abierto, que está mantenido por la *Open Source Robotics Foundation* (OSRF). Al ser de código abierto, la comunidad ha desarrollado multitud de herramientas para ROS, que se integran y sincronizan continuamente con las versiones nuevas que lanzan.

ROS proporciona un entorno distribuido compuesto por nodos, que son programas que se ejecutan de forma independiente tanto en local como en distribuido para realizar tareas individuales. Estos nodos se comunican entre sí de dos formas distintas: modo cliente-servidor entre nodos (modo síncrono) o meidante los denominados *topics* (modo asíncrono). Los *topics* de ROS se basan en comunicación estándar TCP/UDP mediante *sockets*, y están ideados para comunicaciones unidireccionales y en *streaming*, donde cada nodo puede ser publicador o subscriptor. En modo publicador, el nodo está escribiendo datos en un *topic* determinado que algún subscriptor estará consumiendo. En modo subscriptor, el nodo estará consumiendo datos de un *topic* que estará siendo actualizado por un publicador. Los datos que se leen/escriben en cada *topic* siguen un determinado formato a través de mensajes de diversos tipos, según el propósito de la comunicación: mensajes de geometría, velocidad, posición, etc.

Una característica interesante de ROS es que ofrece un sistema de almacenamiento de *topics* llamado ROSBags. Los ROSBags almacenan toda la información de los mensajes emitidos en el momento de la grabación en un solo archivo. Este archivo puede ser posteriormente reproducido para recuperar los mensajes de los *topics* generados durante la grabación en el mismo orden en que fueron emitidos. Esto resulta útil especialmente para la grabación de conjuntos de datos, como imágenes de cámara, comandos de velocidad, de posición, etc., permitiendo al usuario realizar pruebas sobre el mismo conjunto de datos con diferentes modelos.

La versión de ROS utilizada para el desarrollo de este proyecto fue *Melodic Morenia*, que era la LTS cuando el proyecto comenzó. Sin embargo, se está migrando a ROS *Noetic Ninjemys* dado que es la actual LTS.

3.5. Gazebo

Gazebo [4] es un simulador 3D que ofrece un entorno para desarrollar y probar sistemas multirrobot de manera sencilla. Es capaz de simular de forma realista muchos tipos de sensores y actuadores, entre los que destacan cámaras, láseres, sónares, GPS, etc, todo ello gracias, también, al motor de físicas que utiliza: Bullet, y a OpenGL (*Open Graphics Library*). Además da soporte a multitud de robots, tanto predefinidos como definidos por el usuario. Gazebo es un *software* de código libre, ya que está licenciado bajo la licencia de Apache 2.0 [5]

Gazebo sigue a la cabeza de los simuladores de facto en la comunidad robótica del *software* libre debido a su completitud y precisión. En la actualidad ya hay simuladores

[3] https://www.ros.org
[4] http://gazebosim.org
[5] http://www.apache.org/licenses/LICENSE-2.0.html

dedicados a la robótica como el AirSim de Microsoft [6] o la plataforma ISAAC de NVIDIA [7]; no obstante Gazebo tiene integración nativa con ROS por lo que se ha escogido este simulador para este proyecto.

El hecho de que sea un simulador multirrobot dota de una gran utilidad a Gazebo, ya que se pueden simular infinidad de situaciones en las que uno o varios robots interactúen y estudiar su comportamiento de forma segura y controlada.

El simulador Gazebo se ha utilizado para probar la plataforma BehaviorStudio para robots simulados.

3.6. Biblioteca OpenCV

Para el procesamiento de imágenes en general, OpenCV[8] (*Open Source Computer Vision*) es una biblioteca multilenguaje (Python, Java, C++) de código abierto (escrita nativamente en C++) más usada para propósitos de Visión por Computadora. Entre los métodos clásicos que reúne se pueden encontrar varias funciones adecuadas para el reconocimiento facial, la costura de imágenes, el seguimiento de los ojos, el filtrado de colores, etc.

Esta librería ofrece gran cantidad de algoritmos para manejar estructuras de datos, realizar modificaciones sobre las imágenes, detección de formas, etc. Debido a su gran eficiencia dada su integración con librerías de procesamiento de gráficos con GPU (NVIDIA CUDA y OpenGL), se ha convertido en la librería de visión artificial más usada por la comunidad tanto científica como por empresas; es decir, es el estándar *de facto* en cuanto a librerías de procesameinto de imágenes.

Esta librería se ha utilizado como librería auxiliar y para realizar diferentes pruebas, como, por ejemplo, la implementación de una solución no basada en el aprendizaje a máquina para el robot simulado. Se ha utilizado la versión 4.4 de la biblioteca.

3.7. *Middleware* neuronal

Pytorch[9] es una librería de cómputo numérico de alto rendimiento centrada en el cálculo en paralelo utilizando tensores como estructuras de datos estrella. Pytorch es una librería impulsada por Facebook, de nacimiento reciente, en la que su principal ventaja frente a otros marcos como Tensorflow o Caffe es que utiliza grafos dinámicos en lugar de estáticos, mediante una técnica, que los autores denominan *reverse-mode auto-differentiation*, que les permite cambiar el comportamiento de las redes neuronales de forma arbitraria sin coste adicional. Otra característica que hace muy potente a esta librería es que está preparada de forma nativa para trabajar con GPU (*Graphic Processing Unit* o con CPU

[6] https://github.com/microsoft/AirSim
[7] https://developer.nvidia.com/isaac-sim
[8] https://opencv.org
[9] https://pytorch.org

(*Central Processing Unit*) a placer, permitiendo además el cálculo distribuido de manera sencilla a través de métodos del API de la librería.

Pytorch tiene integración total con Python, lo que permite utilizar librerías de ciencias de datos como Numpy, Scipy, Scikit-learn, etc fácilmente. Un tensor en Pytorch es conceptualmente idéntico a uno de Numpy, lo que hace que ambas librerías se integren de forma nativa pudiendo aprovechar la potencia conjunta de las mismas, a la vez que simplifica la codificación de soluciones basadas en aprendizaje automático.

Debido a su gran simplicidad, la curva de entrada sencilla y la extensa documentación repleta de ejemplos de diversa índole, además de la integración con las placas Jetson de NVIDIA, se ha elegido este marco de trabajo como el principal para trabajar con las redes neuronales que se han desarrollado en este proyecto.

3.8. Lenguaje Python y bibliotecas específicas

Este trabajo ha sido desarrollado usando el lenguaje de programación Python[10]. Debido a algunos problemas de compatibilidad con la versión *Melodic Morenia* de ROS 1, se ha optado por la utilización de la versión Python 2.7, aunque ha llegado a su fin de la vida útil. Esto se debe a que la implementación de ROS *Melodic* está basada en Python 2.7 y no es compatible con algunas versiones más recientes de Python sin algunos ajustes previos. Sin embargo, BehaviorStudio es totalmente compatible con las nuevas versiones de Python (Python 3.6+) y la última versión de ROS1: ROS *Noetic Ninjemys*, debido a que es agnóstico con respecto a la plataforma utilizada.

Python se ha utilizado en este proyecto para el desarrollo íntegro de BehaviorStudio, así como todo el código auxiliar de entrenamiento del modelo y preprocesamiento de los datos.

3.8.1. Numpy

La librería más potente para cálculo numérico y manejo de estructuras de datos complejas como los tensores es Numpy[11] (Numeric python). Esta librería está diseñada para trabajar con Python, aunque está desarrollada en C++ para optimizar la eficiencia de la librería. Al igual que ocurre con OpenCV para el procesamiento de imágenes, Numpy se ha convertido en el estándar *de facto* para el cálculo numérico. Con la irrupción de las redes neuronales, dado que estas trabajan principalmente con tensores, se ha popularizado aún más el uso de esta librería. El punto fuerte de esta librería es la optimización de todas sus operaciones, haciendo que las implementaciones manuales de la mayoría de ellas sea altamente ineficiente en comparación.

Todas estas capacidades hacen que Numpy sea una librería excelente para el manejo de estructuras de datos a bajo nivel, permitiendo operar sobre dichas estructuras de forma intuitiva y proporcionando métodos para facilitar las operaciones con ellas. A todo esto se

[10] https://www.python.org
[11] https://numpy.org

suma que Numpy es un proyecto de código abierto, por lo que está en constante desarrollo y mejora gracias a la comunidad.

En este desarrollo se ha utilizado Numpy como librería auxiliar para trabajar con los *datasets* y con los tensores resultantes de las redes neuronales utilizadas.

3.8.2. PyQt

Qt es un conjunto de bibliotecas gráficas multiplataforma para desarrollar interfaces gráficas de usuario (GUIs). En nuestro proyecto se ha hecho uso de la librería Qt5 en su implementación en Python, que es la más estable hasta el momento. Debido a su carácter multiplataforma y multilenguaje, Qt dispone de diferentes APIs (*Application Programming Interfaces*) en función del lenguaje de desarrollo que se utilice. Así por ejemplo disponemos de la versión para C++ denominada Qt, la versión para Java denominada QTJambi o la versión para Python llamada PyQt, entre otras. Dispone tanto de versión gratuita como versión de pago destinada a soluciones empresariales.

Como la mayoría de bibliotecas gráficas, Qt ofrece una gran cantidad de elementos de interfaz tales como: botones, deslizadores verticales y laterales, cuadros de texto, separadores y un largo etc. Adicionalmente, ofrece una interfaz de comunicaciones entre elementos basada en eventos, lo que facilita el paso de información y/o de disparadores para los elementos dinámicos de la interfaz de usuario. Qt cuenta además con un motor de animaciones y una librería 3D que se integra perfectamente en el código regular de la librería.

Esta librería se ha utilizado para el desarrollo de la parte visual de BehaviorStudio. Toda la interfaz gráfica se ha codificado utilizando exclusivamente PyQt5 y PyQt3D, sin la ayuda de IDEs como QtCreator.

3.8.3. Otras bibliotecas

Adicionalmente a las mencionadas, se han utilizado también otras librerías de manera circustancial que se mencionan a continuación:

- **Pandas**[12]. Es un paquete Python de código libre que proporciona estructuras de datos rápidas y flexibles diseñadas para hacer que el análisis / manipulación de datos. Se ha utilizado en este proyecto para la manupulación de los *datasets* con los que se han entrenado las redes neuronales utilizadas.

- **Matplotlib**[13]. Es una biblioteca de graficado en 2D de Python que produce figuras de calidad en una variedad de formatos impresos y entornos interactivos. Se ha utilizado en este proyecto para obtener las gráficas resultantes del entrenamiento de las redes neuronales.

[12] https://pandas.pydata.org
[13] https://matplotlib.org

- **Npyscreen**[14]. Es una biblioteca de widgets de python y un marco de aplicación para programar aplicaciones de terminal o de consola. Está construida sobre ncurses, que es parte de la biblioteca estándar. Se ha utilizado en este proyecto para construir una interfaz de usuario basada en terminal.

[14]https://npyscreen.readthedocs.io/introduction.html

4

BehaviorStudio

Una vez definidas todas las herramientas que se van a utilizar para el desarrollo de este trabajo, se dedicará este capítulo para cubrir de forma extensa y precisa uno de los objetivos fundamentales del mismo: la plataforma de *machine learning* BehaviorStudio[1]. Para ello, se dará primero una visión general de la arquitectura de la aplicación, para acto seguido describir todos los detalles de la implementación llevada a cabo, así como los problemas que han surgido y las soluciones aportadas.

Este capítulo está dividido en cinco secciones principales que abordarán una pequeña introducción de la aplicación, el diseño de la arquitectura, sus principales componentes y el modo de ejecución distribuido. Se concluye con una validación experimental en un entorno de simulación de lo desarrollado.

4.1. Introducción

BehaviorStudio es una plataforma de *machine learning* de comportamientos basados en redes neuronales que permite conectar distintos *software* inteligentes a un robot tanto real como simulado. La plataforma está ideada principalmente para la conducción autónoma a través del control visual utilizando como sensores principales las cámaras a bordo del robot. No obstante, está también preparada para el soporte de diferentes robots como: drones, robots industriales, domésticos, etc, y para diferentes tipos de sensores y actuadores. El principal objetivo de esta plataforma es proporcionar una infraestructura accesible y flexible que permita ejecutar diferentes comportamientos complejos en coches autónomos implementados con diferentes técnicas como: *deep learning* (redes de clasificación, regresión, recurrentes, LSTMs, ...), *reinforcement learning* e incluso programación clásica utilizando librerías de visión como OpenCV.

BehaviorStudio está íntegramente desarrollada en Python dado que es uno de los lenguajes más utilizados por la comunidad en el desarrollo de aplicaciones de aprendizaje automático, al disponer de gran cantidad de librerías y herramientas creadas por la comunidad científica, tales como Numpy, Pytorch, OpenCV, etc. BehaviorStudio surge de la inspiración de otra plataforma similar en características denominada *DetectionStudio*[2] desarrollada por la asociación JdeRobot[3].

[1] https://jderobot.github.io/BehaviorStudio/
[2] https://jderobot.github.io/DetectionStudio/
[3] https://jderobot.github.io

BehaviorStudio proporciona la siguiente funcionalidad:

- **Carga de cerebros**. Se pueden cargar diferentes algoritmos de comportamiento para su evaluación en tiempo de ejecución.
- **Modificación «en caliente»**. BehaviorStudio permite la modificación del código del cerebro en ejecución sin necesidad de parar el programa, realizando carga dinámica de los algoritmos que gobiernan el robot.
- **Ejecución en diferentes plataformas**. La plataforma soporta entornos de ejecución locales (simulación en Gazebo) y remotos (robots reales).
- **Visualización**. Se incorpora la visualización de diferentes sensores como la cámara RGB o el láser.
- **Modo de ejecución distribuida**. Con BehaviorStudio se permite ejecutar la lógica y la visualización en diferentes computadores (modo *headless*), para permitir así que la capa de lógica corra empotrada en los robots, mientras que la capa de visualización se ejecute en un computador externo.
- **Creación de *datasets***. Se pueden grabar conjuntos de datos utilizando *ROSBags* asociados a *topics* de ROS.

La implementación de esta plataforma es flexible, lo que permite incorporar nueva funcionalidad como visualización de otros tipos de sensores de forma sencilla, o de nuevos marcos de trabajo como Tensorflow, Caffe, etc. Actualmente soporta Keras y Pytorch.

4.2. Arquitectura *software*

El diseño de BehaviorStudio se puede ver en la Figura 4.1. Como se puede observar, al tratarse de una aplicación que incorpora visualización, se ha implementado una arquitectura Modelo-Vista-Controlador (MVC). Se ha elegido esta arquitectura porque distingue muy bien la parte de lógica y la parte de visualización, disminuyendo el acoplamiento de componentes y facilitando la cohesión de los mismos. La parte inferior de esa infografía, bajo la etiqueta "*Robotic Platform*", ilustra la capacidad que tiene BehaviorStudio de conectase con robots reales o simulados indistintamente, cambiando únicamente la configuración de la aplicación. Esto se consigue gracias a la estandarización en las comunicaciones que ofrece ROS en forma de *topics* tanto para recibir la información sensorial de los sensores del robot, como para comandar órdenes a sus actuadores.

Los principales componentes de esta aplicación son los que se aprecian en la Figura 4.1. Por simplicidad, este diagrama sólo muestra los componentes principales de BehaviorStudio, obviando el módulo de grabación de conjuntos de datos y el módulo *driver*, que se encargan de generar un fichero ROSBag a partir de los *topics* que el usuario seleccione en el interfaz de usuario (se explica más detalladamente en la sección 4.3.5.1) y de gestionar la ejecución de la aplicación (punto de entrada) respectivamente.

Se puede apreciar que los componentes robot, piloto y el cerebro forman parte del Modelo en el MVC; a su vez se observa el módulo controlador perteneciente a la capa

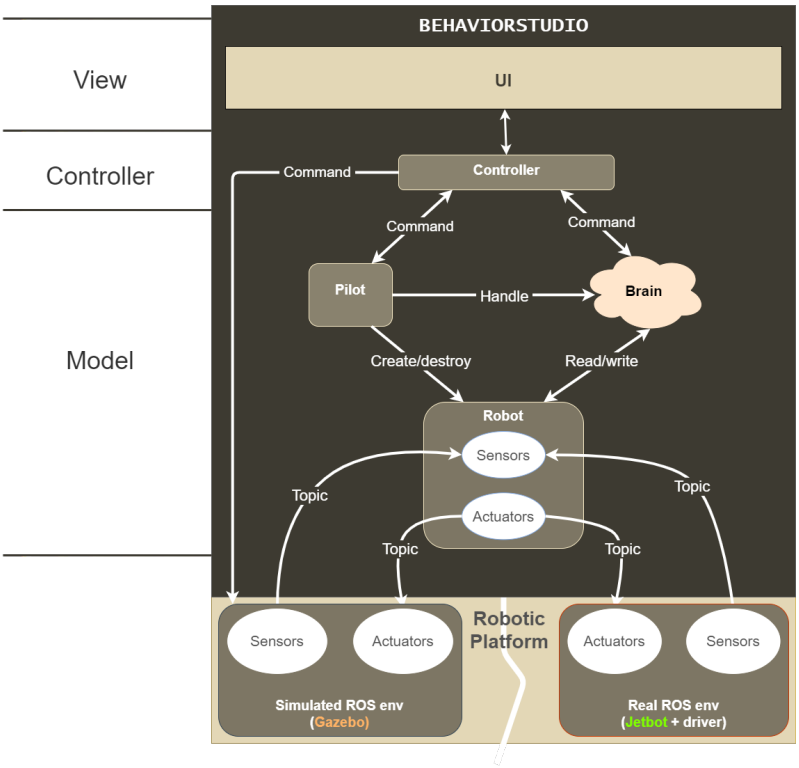


Figura 4.1: Arquitectura de BehaviorStudio

controladora; y por último la interfaz de usuario (UI) que pertenece a la capa de vista del MVC.

4.3. Componentes

En esta sección se detallan los diferentes componentes que componen la arquitectura de la aplicación, tal como ilustra la Figura 4.1. Cada componente se explicará en una sección diferente para mayor claridad.

4.3.1. Piloto

El módulo principal de la aplicación es el piloto. Esta pieza se encarga principalmente de la creación tanto del robot (sensores y actuadores) como del cerebro que va a comandar al robot. El piloto tiene un hilo de ejecución que se actualiza periódicamente cada 50 ms (20 Hz), donde en cada iteración se ejecuta una llamada al cerebro que esté cargado en ese momento comandando al robot, para que éste tome una decisión en función del escenario en el que se encuentre en cada instante. En otras palabras, cada 50 ms (20 Hz) se realiza una llamada al método `execute()` del cerebro cargado, que capturará información del sensor o sensores, realizará una inferencia sobre la información capturada y tomará una decisión sobre cómo debe actuar el robot en ese instante. Esto permite que el control sea reactivo, ya que se toman decisiones en cada iteración en función de las percepciones del robot.

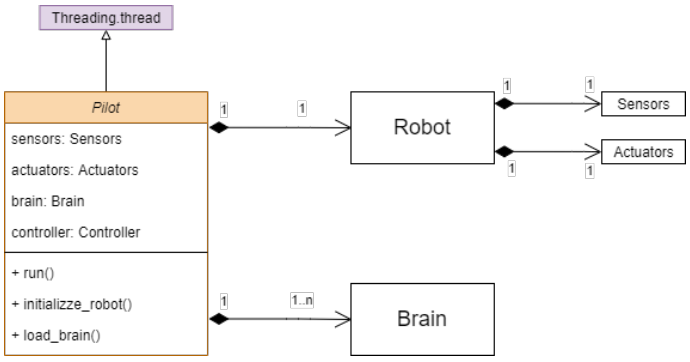


Figura 4.2: Diagrama de clases UML del componente pioto

Otra característica que se apuntó en la introducción de este capítulo, es que BehaviorStudio permite el intercambio de cerebros en caliente, además de cambios en el código fuente de estos cerebros de forma dinámica sin necesidad de reiniciar la aplicación (en tiempo de ejecución). Es el piloto el que se encarga de realizar la carga y descarga de estos cerebros de forma dinámica, a partir de lo que el usuario indique a través del interfaz de usuario.

En la Figura 4.2 se ilustra un diagrama de clases reducido de la clase piloto. Esta clase dispone de los métodos necesarios para crear un robot y cargar cerebros a placer de forma dinámica.

4.3.2. Robot - Sensores y actuadores

Dos elementos fundamentales de los robots son sus sensores y actuadores, ya que sin ellos el robot sería incapaz de percibir (o sensar) ni de interactuar con el entorno que le rodea. Por lo tanto, uno de los pilares principales de BehaviorStudio es la implementación del soporte de los diferentes sensores y actuadores del robot.

La implementación de este componente es sencilla, ya que se trata de un paquete de Python que incluye sendas clases `Sensors` y `Actuators` que se conectan a los robots (reales o simulados) mediante interfaces de ROS. Estas clases reciben una lista de sensores y actuadores, con sus respectivos *topics* de ROS que conectan con las interfaces del robot, desde el fichero de configuración de la aplicación. En la Figura 4.3 se puede ver un pequeño diagrama de clases UML que ilustra parcialmente la implementación de este componente. Esta implementación es flexible, ya que permite incluir una cantidad arbitraria de sensores y/o actuadores sin tener que modificar el código, solamente modificando el fichero de configuración.

La forma en la que se construye un objeto sensor o actuador es especificando el tipo de sensor se quiere construir, un identificador y el *topic* de ROS asociado al publicador o subscriptor de ese sensor/actuador. Adicionalmente se podrá especificar configuración extra para cada sensor/actuador. Todo esto se especifica a través de un fichero de configuración en formato YAML. En el Fragmento 4.1 se puede ver un ejemplo de configuración para el robot Formula1 simulado.

Se pueden agregar tantos sensores o actuadores como tenga el robot en uso, pudiendo

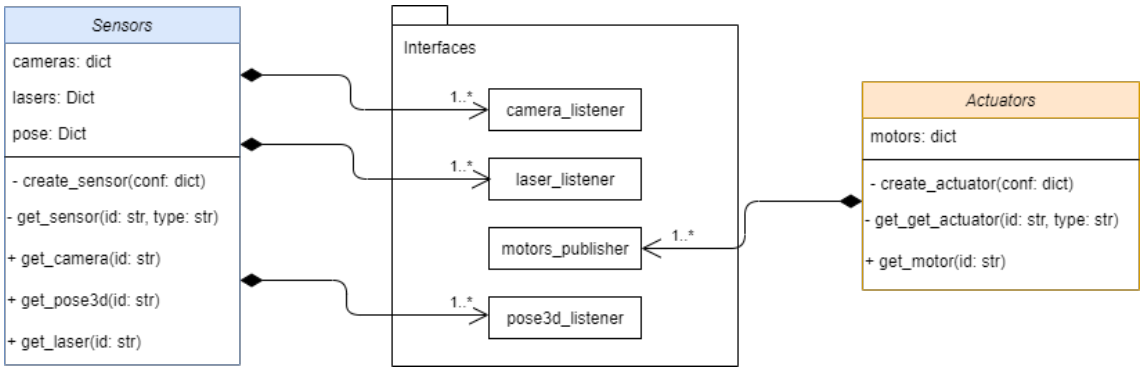


Figura 4.3: Diagrama de clases UML del componente Robot

identificar cada uno de ellos con el nombre proporcionado en la etiqueta *Name*. Los tipos de sensores soportados en la versión actual de BehaviorStudio son: cámaras RGB, láser, y pose3D. Por su parte, los actuadores soportados son únicamente motores.

```
Robot:
   Sensors:
      Cameras:
         Camera_0:
            Name: 'camera_0'
            Topic: '/F1ROS/cameraL/image_raw'
      Pose3D:
         Pose3D_0:
            Name: 'pose3d_0'
            Topic: '/F1ROS/odom'
   Actuators:
      Motors:
         Motors_0:
            Name: 'motors_0'
            Topic: '/F1ROS/cmd_vel'
            MaxV: 3
            MaxW: 0.3
   BrainPath: 'brains/f1/brain_f1_opencv.py'
   Type: 'f1'
```

Fragmento 4.1: Ejemplo de configuración en formato YAML

Las interfaces de ROS que conectan los sensores y actuadores a los robots reales o simulados son las siguientes:

- La cámara se conecta mediate *topics* de ROS que transportan mensajes del tipo `std_msgs/Image`.
- El láser se conecta mediante *topics* de ROS que transportan mensajes del tipo `sensor_msgs/LaserScan`.
- La odometría se conecta mediante *topics* de ROS que transportan mensajes del tipo `nav_msgs/Odometry`.
- Los motores se conectan mediante *topics* de ROS que transportan mensajes del tipo `geometry_msg/Twist`.

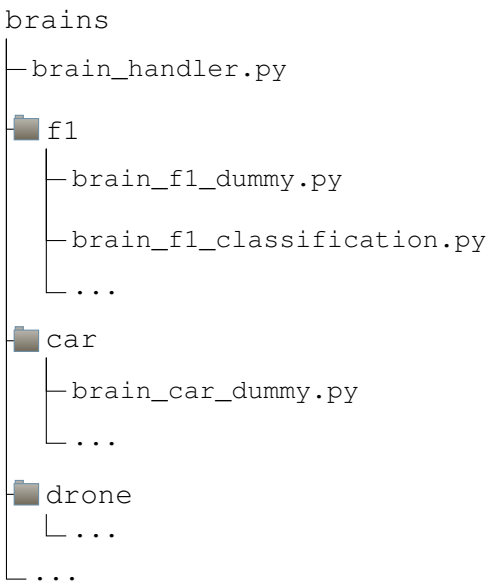


Figura 4.4: Estructura de directorios de cerebros neuronales

4.3.3. Cerebros neuronales

El módulo de cerebros neuronales (llamado *Brains*), es la pieza fundamental de BehaviorStudio, ya que es aquí donde residen los algoritmos que gobernarán a los robots. Como se aprecia en la Figura 4.1, este módulo conecta con los sensores y actuadores de los robots y es gestionado por el piloto, como se explicó en la sección 4.3.1. Además tiene comunicación con el controlador, para que el usuario pueda intercambiar los cerebros que gobiernan al robot, pausar o reanudar su ejecución, o modificar el código fuente; todo ello en tiempo de ejecución. La conexión con los sensores y actuadores del robot es fundamental para que el cerebro que esté en ejecución pueda realizar la toma de decisiones a partir de los datos sensoriales, y comandar acciones al robot en función de los cálculos realizados por los algoritmos que gobiernan los robots.

Todas las implementaciones de los cerebros comparten un interfaz común que «obliga» a seguir un estándar en su programación forzando una guía de estilo para todas las implementaciones que realicen los usuarios. Se muestra un ejemplo de implementación de un cerebro en el Fragmento 4.2, que ilustra la estructura básica de su programación. Como se puede observar, el cerebro de este ejemplo es muy sencillo, ya que genera un comportamiento en el que el robot gira sobre su propio eje a 0,8 rad/s.

Cada cerebro debe implementar una clase llamada *Brain* instanciando en el constructor los sensores y actuadores del robot, identificados por su nombre (en el caso del Fragmento 4.2 las líneas 5 `camera_0` y 9 `motors_0` para la cámara y los motores respectivamente), además de una instancia del manejador de cerebros llamada `handler`, que gestiona las comunicaciones con el módulo controlador. Dispone de un método `update_frame()` que se encarga de actualizar la información que llega al interfaz gráfica de usuario con la información sensorial pertinente cada vez que se invoca. Por último, la función `execute()` es la encargada del bucle principal de ejecución que es llamada desde el módulo Piloto a un ritmo de ejecución determinado (ver sección 4.3.1).

En el caso del Fragmento 4.2, cada vez que el método `execute()` es invocado (línea 12), se ejecutará lo siguiente: que el robot no avance (línea 13), que el robot gire sobre sí mismo a una velocidad de 0,8 rad/s (línea 14), obtener una imagen de la cámara a bordo del robot (línea 15) y actualizar la información que se mostrará en la interfaz de usuario con la imagen capturada (línea 16). En la última instrucción, el primer argumento es `frame_0`, que es el nombre que recibe el cuadro donde se mostrará la imagen capturada de la cámara del robot en el interfaz de usuario. Esto se explicará con más detalle en la sección 4.3.5.

```
1  class Brain:
2
3      def __init__(self, sensors, actuators, handler=None):
4
5          self.camera = sensors.get_camera('camera_0')
6          self.motors = actuators.get_motor('motors_0')
7          self.handler = handler
8
9      def update_frame(self, frame_id, data):
10         self.handler.update_frame(frame_id, data)
11
12     def execute(self):
13         self.motors.sendV(0)
14         self.motors.sendW(0.8)
15         image = self.camera.getImage().data
16         self.update_frame('Camera0', image)
```

Fragmento 4.2: Ejemplo de implementación de cerebro no neuronal

La estructura del árbol de ficheros de este módulo se observa en la Figura 4.4, donde se pueden apreciar los módulos que contienen el código de los diferentes cerebros desarrollados hasta la fecha. Como se puede ver, está dividido por tipo de robot con el objetivo de estructurar todos los cerebros a un mismo nivel de abstracción y que sea sencillo para los usuarios saber dónde incluir sus algoritmos. Cada uno de los ficheros `*.py` dentro de los directorios contiene un cerebro neuronal o no neuronal con comportamientos diferentes. El módulo de python `brain_f1_dummy.py` corresponde al código mostrado en el Fragmento 4.2. El fichero `brain_handler.py` corresponde al manejador de cerebros, que es la clase que se encarga de gestionar la comunicación de cada cerebro con el controlador, que, como se explica en la sección 4.3.4, es el módulo encargado de comunicar la lógica de la aplicación con la capa de visualización.

La plataforma BehaviorStudio está ideada para centrarse en la conducción autónoma con control basado en visión. La Figura 4.4 ilustra muy bien que igualmente está preparada para soportar más tipos de robots y comportamientos.

4.3.4. Controlador

El controlador de una arquitectura MVC (Modelo-Vista-Controlador) actúa como puente de comunicación entre la capa modelo y la capa de visualización. Esto se tra-

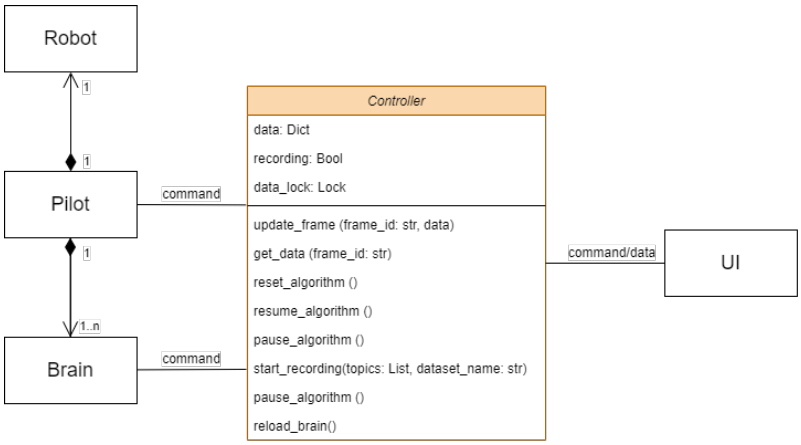


Figura 4.5: Diagrama de clases UML del componente Controlador

duce en el desacople de la parte lógica de la aplicación y la interfaz de usuario, lo que permite que todas las partes del *software* no estén atadas a una implementación concreta, pudiendo modificar los algoritmos sin que cambie la estructura.

En BehaviorStudio, el módulo controlador es el encargado de traducir a la parte de lógica todas las órdenes comandadas por el usuario a través del interfaz de usuario (UI); es decir, actúa como intermediario entre el usuario y la lógica de la aplicación. En la Figura 4.5 ilustra un pequeño diagrama de clases con las responsabilidades de este módulo.

En esencia, el controlador se encarga de las siguientes tareas:

- Recoger la información de los sensores que envía el robot y almacenarla para su posterior visualización en el interfaz de usuario.
- Entregar la información sensorial que el interfaz de usuario requiera en cada momento.
- Evitar condiciones de carrera entre la escritura de la información sensorial que llega del robot y la información que va a consumir el interfaz de usuario.
- Gestionar las órdenes de: pausar, continuar, reiniciar el algoritmo controlador tanto en entorno simulado como en entorno real.
- Comandar el cambio de cerebro.
- Gestionar las órdenes de iniciar y parar la grabación de *datasets*

El controlador tiene dos modos de conexión con la capa de visualización: conexión local mediante objetos de Python y conexión remota a través de *sockets* de Python. Esta conexión permite que la capa de lógica y la capa de visualización se ejecuten de forma distribuida en computadores diferentes, útil para liberar de cómputo a procesadores embebidos. En la sección 4.4 se detalla este comportamiento.

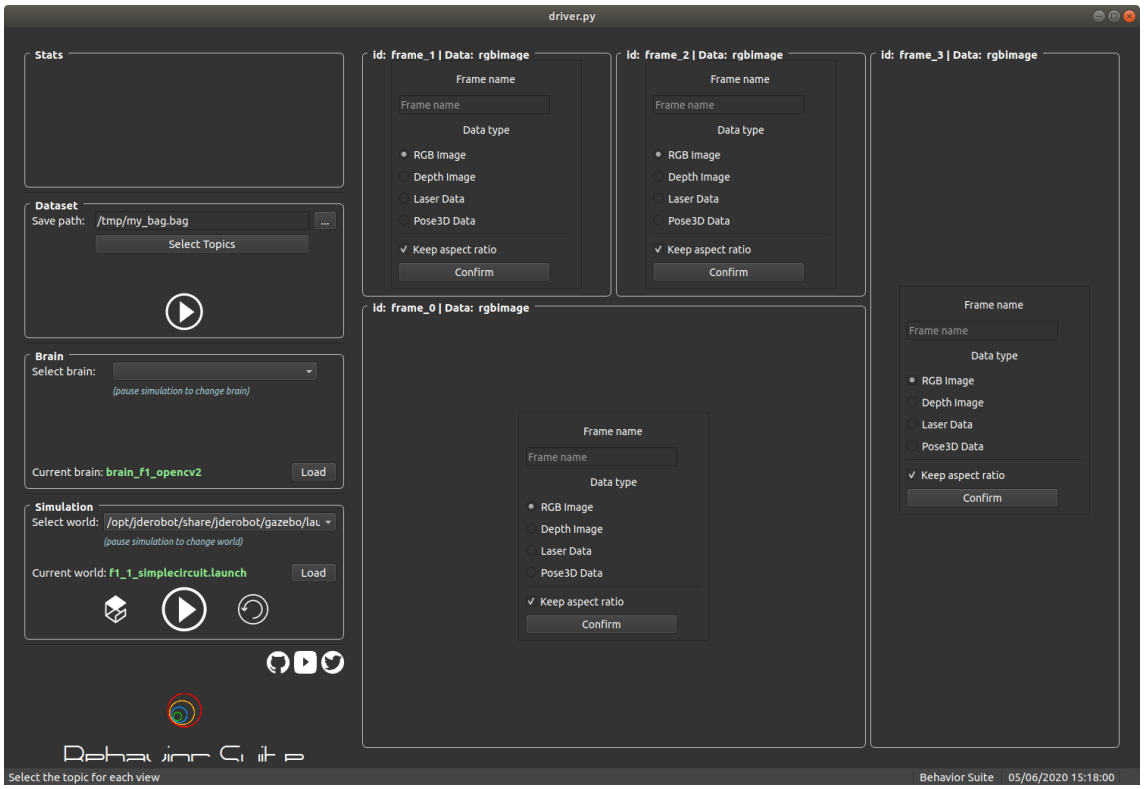


Figura 4.6: Vista principal de BehaviorStudio

4.3.5. Interfaz de usuario - GUI

Con todos los elementos de lógica y control desarrollados, se necesita un mecanismo para permitir al usuario final interactuar con todas las funcionalidades que ofrece la plataforma. Para este propósito existen las interfaces de usuario o UI (*User Interface* por sus siglas en inglés). Las interfaces de usuario pueden ser de diferentes tipos: gráficas (GUI o *Graphical User Interface*), aplicación gráfica de consola, por línea de comandos (CLI o *Command Line Interface*), modo texto en terminal (TUI o *Text User Interface*), etc. Para este proyecto se han desarrollado dos de ellas: el GUI y una aplicación gráfica de consola como prueba de concepto.

La vista principal de BehaviorStudio se puede ver en la Figura 4.6. Este interfaz está dividido en dos bloques principales: la barra de herramientas (izquierda) y la visualización (derecha). A continuación se desgranan todos los componentes uno por uno.

4.3.5.1. Barra de herramientas

La parte izquierda de la vista principal de BehaviorStudio contiene una barra de herramientas que permite al usuario interactuar con el sistema. Esta barra de herramientas ofrece toda la funcionalidad que el usuario podrá llevar a cabo mediante los diferentes botones que la componen. Como se puede observar en la Figura 4.7, la barra de herramientas está dividida en secciones agrupadas por funcionalidad. La primera sección titulada *Stats* está diseñada para mostrar información sobre la aplicación, como consumos de memoria

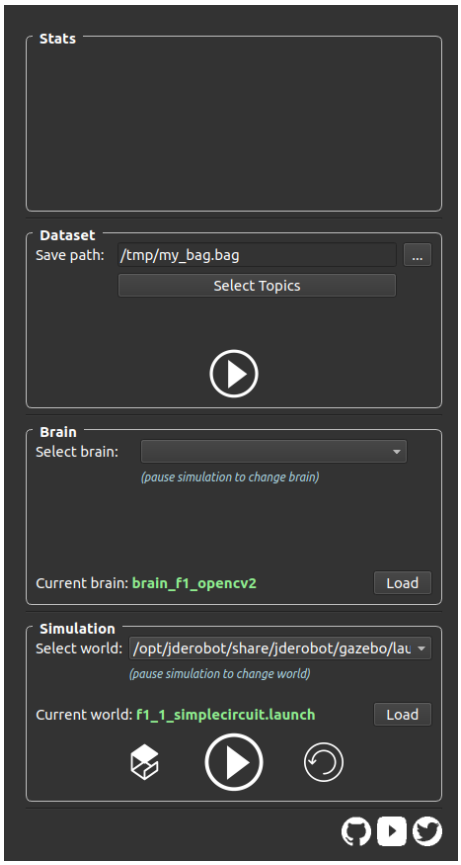


Figura 4.7: Barra de herramientas de BehaviorStudio

y CPU, estado de la conexión con el robot, tiempo de ejecución, etc.

La sección de grabación de datasets, titulada *Dataset*, permite al usuario controlar la grabación de conjuntos de datos a partir de *topics* de ROS. La grabación de los *datasets* se realiza mediante ROSBags, por lo que el usuario deberá indicar una ubicación y un nombre para el archivo de salida `.bag`, y los *topics* de ROS activos que desea grabar. Una vez proporcionada esa información, mediante los botones proporcionados por el GUI, el usuario podrá comenzar o detener la grabación del *dataset* con el botón *play* disponible.

La sección titulada *Brain* es una de las dos más importantes de esta barra de herramientas. Con ella, el usuario podrá cargar diferentes cerebros (o controladores visuales) para el robot en tiempo de ejecución. El desplegable ofrecerá una lista de cerebros disponibles para el robot en uso, pudiendo elegir el usuario el que desee probar en cada momento. Para poder realizar la carga en tiempo de ejecución, el algoritmo que estuviera ejecutando (si hubiera alguno) deberá estar pausado desde la sección de *Simulation*.

La sección de control del algoritmo, titulada *Simulation*, permite que el usuario pueda cargar diferentes entornos de ejecución simulados (diferentes mundos con diferentes robots), además de ofrecer tres botones para el control tanto del simulador como del algoritmo (cerebro) en ejecución:

- El botón con el logo de Gazebo, abre o cierra el cliente del simulador en función de si este ya estaba ejecutando o no. Esto es útil para computadores con poca potencia

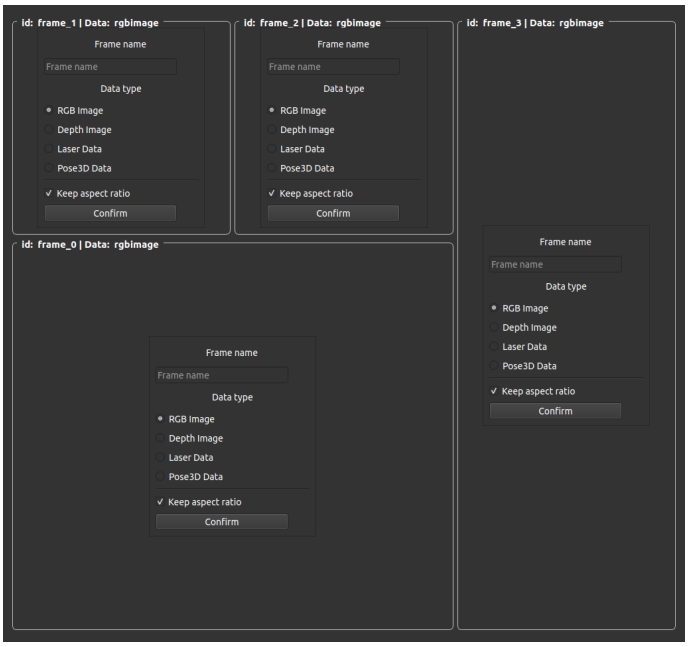


Figura 4.8: Zona de visualización de BehaviorStudio

gráfica, ya que alivia a la CPU para centrarse en la ejecución eficiente del algoritmo. Dado que el GUI de BehaviorStudio ya ofrece visualización de los diferentes sensores, se puede prescindir del cliente del simulador.

- El botón con el icono de *play* permite ejecutar, pausar y continuar un algoritmo concreto especificado en la sección de la barra de herramientas *Brain*. Como se ha explicado más arriba, la carga de cerebros en tiempo de ejecución sólo es posible si el algoritmo está pausado; este botón sirve para este propósito.

- El botón con el icono de una flecha circular sirve para reiniciar el entorno de ejecución simulado. Si el algoritmo del usuario no funciona de forma adecuada y ha provocado un estado no deseado en el entorno simulado (el coche se ha salido de la carretera, ha chocado, etc.), éste puede reiniciar la simulación devolviendo el robot a la posición original cuando se cargó el entorno simulado.

Por último, en la parte baja de la barra de herramientas hay botones para acceder a las diferentes redes sociales del proyecto.

Todas estas herramientas mencionadas se comunican con el sistema mediante el controlador explicado en la sección 4.3.4. Cada acción que el usuario realiza en el interfaz de usuario se traduce a un comando que llegará al sistema para llevar a cabo dicha acción.

4.3.5.2. Zona de visualización

El otro gran bloque de este GUI es el de visualización. Esta parte del interfaz de usuario está diseñada para la representación de la información sensorial que ofrece el robot, como las imágenes capturadas por las cámaras RGB, la información de profundidad ofrecida por las cámaras RGBD, la información de distancia de los láseres, o la odometría del robot.

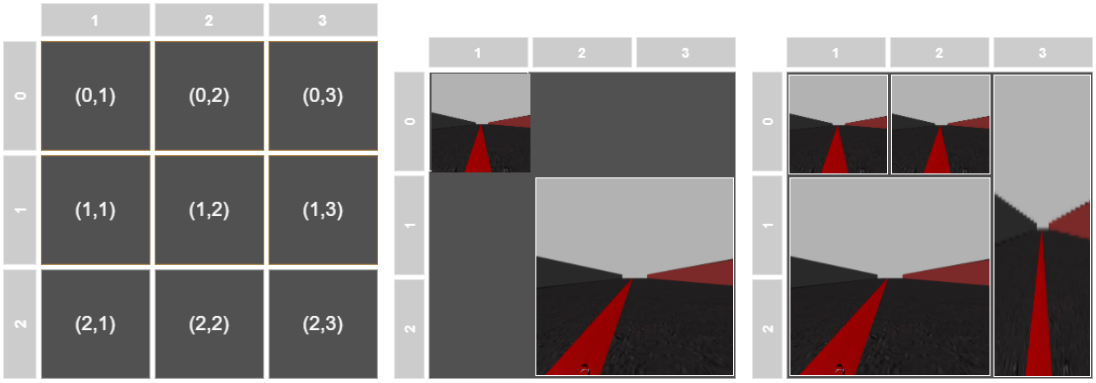


Figura 4.9: (a) Rejilla de distribución de vistas. (b) Disposición de ejemplo. (c) Disposición por defecto

El bloque de visualización está subdividido en cuadros o *frames*, que son totalmente configurables por el usuario. La disposición de estos cuadros es una rejilla de 3x3 que puede ser dispuesta a placer. En la Figura 4.9 se incluye un esquema para ilustrar la naturaleza configurable de estas vistas. La Figura 4.9 (a) muestra la rejilla de 3x3 mencionada con coordenadas, la (b) muestra una configuración de ejemplo para ilustrar que una vista puede ocupar varias casillas y la (c) muestra la configuración por defecto de la plataforma. Para ilustrar cómo se codifica la disposición de la Figura 4.9 (b), se incluye el Fragmento 4.3

```
1
2  Frame_0:
3     Name: "Camera0"
4     Geometry: [0, 1, 1, 1]
5     Data: rgbimage
6
7  Frame_1:
8     Name: "Camera1"
9     Geometry: [1, 2, 2, 2]
10    Data: rgbimage
```

Fragmento 4.3: Ejemplo de configuración de la visualización

Como se puede ver en el Fragmento 4.3, cada cuadro de visualización requiere de un nombre, una geometría y un tipo de dato por defecto. El nombre es importante ya que será el que conecte la lógica del cerebro con la visualización en el GUI. El nombre indicado en esta configuración será el mismo que el usuario tendrá que utilizar en el código del cerebro (Fragmento 4.2 línea 16) cuando quiera mostrar la información sensorial en un cuadro determinado. La geometría indica la disposición de cada cuadro en la rejilla de 3x3 mencionada arriba. En el ejemplo del Fragmento 4.3, en la línea 3, se indica que el primer cuadro de visualización deberá estar posicionado en la coordenada (0,1) con un tamaño (o *span*) de 1 posición en horizontal y 1 posición en vertical; por su parte, en la línea 8 se especifica que el segundo cuadro de visualización deberá estar posicionado en la coordenada (1,2) con un tamaño en horizontal y vertical de dos posiciones. El tipo de dato indica la información que se representará en ese cuadro: imágenes RGB, imágenes DEPTH, información de laser, etc.

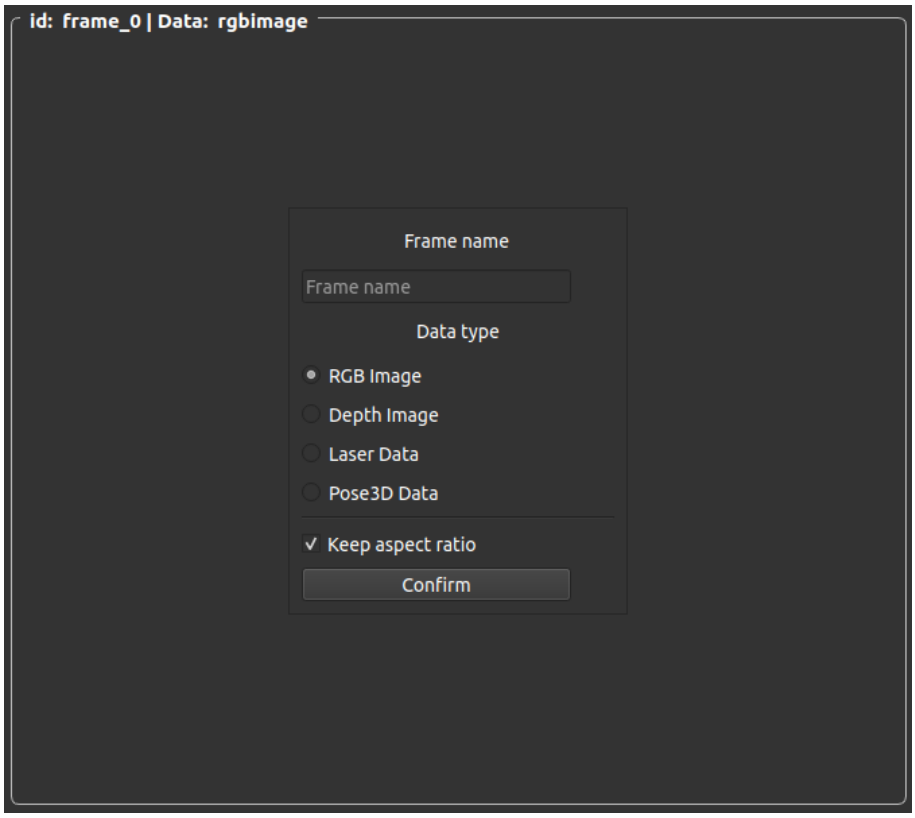


Figura 4.10: Cuadro de visualización sensorial de BehaviorStudio

Una vez que el usuario haya configurado las vistas a su gusto, cuando lance la aplicación verá algo similar a lo mostrado en la Figura 4.6. Como se ve, en esta vista no aparece la información de ningún sensor; esto es debido a que el GUI no mostrará información sensorial hasta que no haya un algoritmo en ejecución. Cuando se ordene a la aplicación ejecutar el algoritmo de control el usuario podrá observar la información sensorial en pantalla. Cabe destacar que los diferentes cuadros de visualización pueden reconfigurarse mientras se está ejecutando un algoritmo para mostrar otros datos diferentes. Como se puede ver en la Figura 4.10, tanto el nombre del cuadro como el tipo de dato podrá configurarse desde el GUI, sobreescribiendo lo indicado en el fichero de configuración con el que se lanzó la aplicación. Esto es especialmente útil si, por ejemplo, se desea utilizar un cuadro de visualización más grande para mostrar más detalles de algún sensor sin necesidad de tener que reiniciar la aplicación.

4.3.5.3. Interfaz por terminal - TUI

Otro tipo de interfaz de usuario es el TUI o *Terminal User Interface*. Básicamente, un TUI sirve para exactamente lo mismo que un GUI (*Graphic User Interface*) pero pintado sobre una consola en lugar de con una librería gráfica basada en ventanas. El uso de TUI supone una ventaja en dos frentes: en aplicaciones minimalistas donde hay pocas (o ninguna herramienta) con las que el usuario pueda interactuar (por ejemplo los programa *htop*, *nvtop* o *ctop* disponibles en los sistemas Linux) ahorrando tiempo de desarrollo de interfaces más complejas; o en aplicaciones embebidas que se ejecuten en sistemas pequeños con menor capacidad de cómputo que un ordenador sobremesa, para disponer de una capa

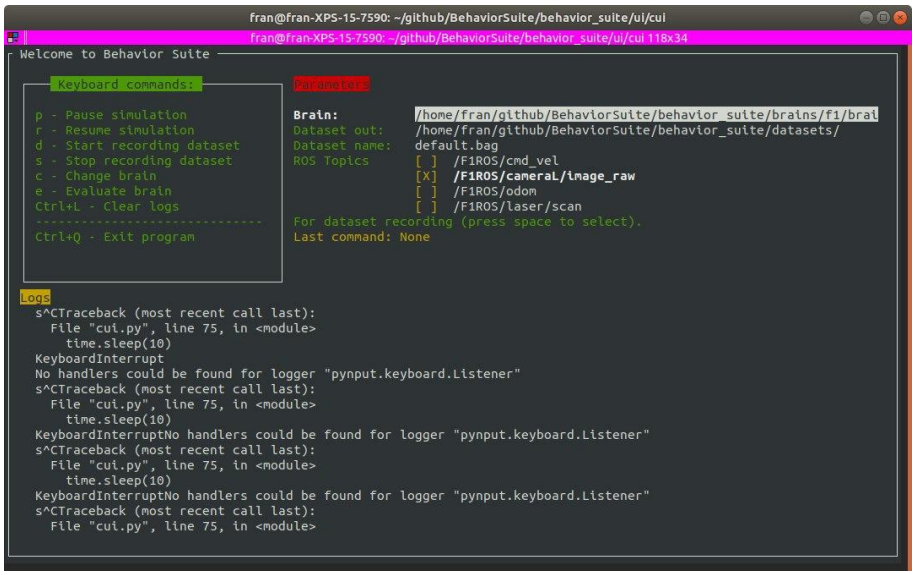


Figura 4.11: Prueba de concepto de un *Terminal User Interface* de BehaviorStudio

de visualización con la que poder interactuar con el sistema.

En BehaviorStudio se ha desarrollado una prueba de concepto de un TUI, con el objetivo de disponer un interfaz gráfico con el que interactuar con el sistema, a bordo del robot. Dado que este tipo de interfaces no requiere de mucha potencia gráfica al ser representados en un terminal, es posible que coexista con el algoritmo de control (que se llevará la mayor parte de la potencia de cómputo).

La funcionalidad de este TUI es la misma que la que ofrece el interfaz gráfico (GUI) pero de forma más limitada ya que han de representarse todas las funcionalidades en una consola. La forma de interactuar con el TUI es a través del teclado, por lo que se ha desarrollado un manejador de eventos que gestione la pulsación de cada tecla. Como se puede observar en la Figura 4.11, los comandos posibles aparecen en una lista a la izquierda de la terminal, de modo que cada tecla se corresponde con una acción diferente. Además, se puede observar que a la derecha hay diferentes herramientas que permiten seleccionar el cerebro a cargar para ejecutar un comportamiento en el robot, para definir la configuración de la grabación de conjuntos de datos (fichero de salida y *topics* de ROS a grabar), además de un registro de eventos en la parte inferior para monitorizar el estado de la aplicación. La principal ventaja de este TUI es que ofrece una manera de interactuar con el robot sin necesidad de cargar el pesado GUI, pudiendo tener centralizada la ejecución de BehaviorStudio dentro del robot. La desventaja, es que se pierde la visualización de la información sensorial. Como se ha mencionado, esto es una prueba de concepto, por lo que es necesario seguir iterando en el desarrollo de esta parte de la plataforma BehaviorStudio.

4.4. Modo de ejecución distribuido

En la sección anterior se ha descrito la interfaz de usuario desarrollada para BehaviorStudio tanto en formato GUI como en formato TUI. Dado que el sistema va a ser ejecutado en una placa de procesamiento pequeña como la Jetson Nano, es lógico pensar

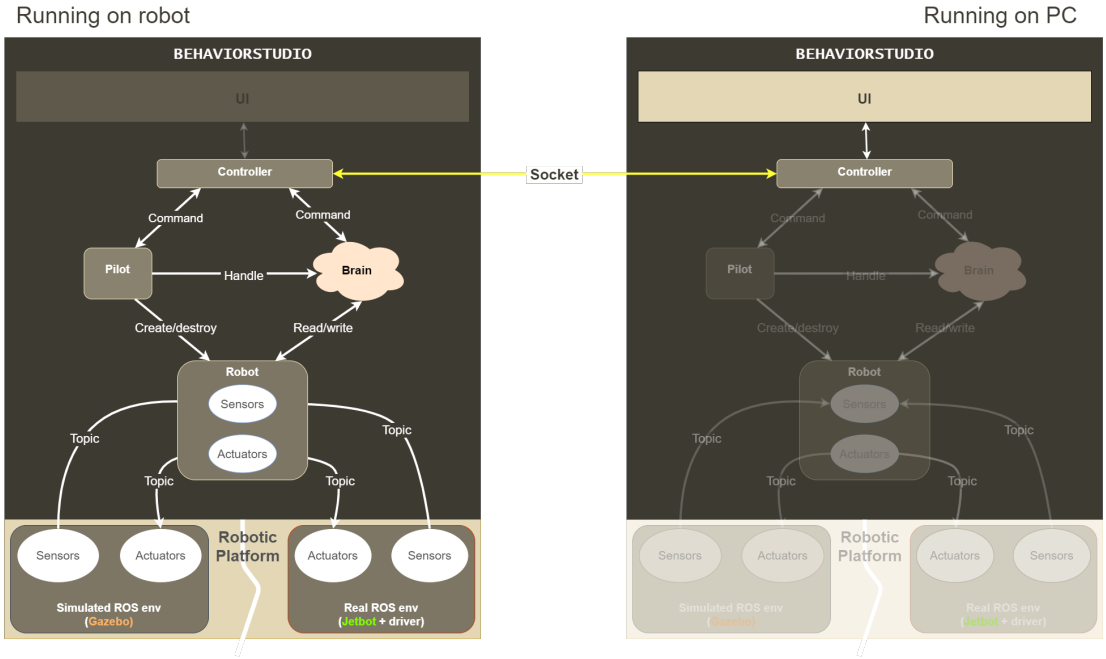


Figura 4.12: Arquitectura del modo *headless* de BehaviorStudio

que sería un desperdicio de recursos el implementar una interfaz de usuario gráfica de las características de las desarrolladas para la plataforma BehaviorStudio. Como alternativa, se puede implementar un TUI para la interacción con el sistema. Sin embargo, esto supone un gran inconveniente: no se pueden visualizar las imágenes de la cámara del robot. Dado que este proyecto trata de resolver el problema de la conducción autónoma mediante controladores visuales, ser capaz de depurar los algoritmos que gobiernan al robot sin poder ver la información de las cámaras sería una tarea complicada. Es por este motivo que se decide implementar un sistema de visualización distribuido, que ha sido bautizado como modo *headless*, para obtener lo mejor de ambos mundos: no derrochar recursos del procesador embebido en representar datos sensoriales, y poder visualizar los datos sensoriales en tiempo de ejecución.

La idea principal de este modo de ejecución es que la capa de lógica (el cerebro) se ejecute en el robot, mientras que la capa de visualización se ejecute en un computador más potente sin necesidad de destinar recursos computacionales a la representación de datos en pantalla. En el caso de este proyecto esto es especialmente importante, ya que la librería de gráficos utilizada (PyQt) es bastante pesada. En la Figura 4.12 se propone la arquitectura de este modo *headless*.

Se puede observar que la arquitectura de la aplicación es la misma que la mostrada en la Figura 4.1, pero duplicada. Esta representación ilustra que son dos instancias de la misma aplicación las que corren tanto en el robot como en el PC de forma distribuida, pero con partes desactivadas. La figura de la izquierda ilustra la instancia que se ejecuta en el robot, donde se puede ver que la capa de visualización está desactivada, mientras que la parte de la lógica y el controlador están activos. En el esquema de la derecha, que se ejecuta en el PC, se observa el caso contrario: la capa de lógica está desactivada mientras que la capa de visualización y el controlador no. Este diseño está pensado para que haya una comunicación entre los controladores de ambas instancias de modo que el

robot pueda enviar la información sensorial al PC que ejecuta la visualización, y el usuario pueda interactuar con el robot de igual manera. Para llevar a cabo esta comunicación entre instancias, se hace uso de *sockets* de Python. Estos *sockets* permiten que dos piezas de *software* puedan intercambiar información a través de la red mediante un protocolo de comunicaciones determinado basado en TCP o UDP. De esta forma, el intercambio entre los datos sensoriales y los comandos del usuario se produce por este medio.

El protocolo de comunicación escogido en este caso es el UDP, ya que se requiere que el sistema sea reactivo. El protocolo UDP se caracteriza por tener menos latencia en el paso de mensajes y por la velocidad de envío de paquetes. Al contrario que TCP, UDP no tiene un mecanismo de seguridad para asegurar que el paquete que se ha enviado por la red ha llegado a su destino, por lo que se pueden perder paquetes sin opción de recuperación. Dado que en el problema que se está tratando se toman decisiones en cada iteración, no es tan crítico la pérdida de paquetes, ya que en pocos milisegundos llegará el siguiente con otros comandos para el robot. En la actualidad, las redes modernas son suficientemente veloces como para que el aporte de velocidad de UDP sea despreciable, no obstante, la latencia es un factor crítico en estos sistemas, ya que es necesario que las decisiones tomadas por el algoritmo de control no se demoren demasiado.

4.5. Validación experimental

Como se explicó en la introducción, BehaviorStudio es una plataforma para la ejecución de comportamientos complejos en robots tanto reales como simulados. Una vez desarrollada la plataforma, se requiere de una validación de su funcionamiento más allá de los test unitarios. Para este propósito se ha construido el llamado piloto explícito. Este piloto es un controlador visual no neuronal basado en operaciones con imágenes mediante el uso de la librería OpenCV en un entorno simulado. Concretamente, el algoritmo realiza un filtro de color sobre la imagen que recibe del robot, donde segmenta la línea central del circuito que ha de seguir. Con la información de la imagen binaria segmentada, se calculan 2 segmentos pertenecientes a la línea filtrada (los que unen la parte baja y media de la imagen, y los que unen la parte media con el horizonte) y se calculan las pendientes de estos para determinar si el robot se encuentra en una recta o a una curva. Esto junto a un controlador PID para corregir la trayectoria hace que el robot sea capaz de completar el circuito de forma autónoma sin salirse de la pista.

Además de ese piloto explícito, se ha creado otro piloto no neuronal a modo de títere que hace que el robot gire sobre sí mismo, y se ha refactorizado un cerebro neuronal de clasificación basado en el trabajo de Vanessa Fernández [9], para ajustarlo al formato de cerebro explicado en la sección 4.3.3. Estos tres cerebros (el explícito, el títere y el neuronal) se utilizan para probar los cambios de cerebro en tiempo de ejecución de la plataforma. Además, dado que el entorno de simulación utilizado hace uso de robots que implementan interfaces ROS, es posible la validación del resto de funcionalidades de la aplicación como la grabación de *datasets*, el cambio de circuitos en tiempo de ejecución y el cambio de vistas de los diferentes sensores. En la web de BehaviorStudio[4] se explican todas sus funcionalidades ilustrándolas con pequeñas imágenes en movimiento creadas para esta validación experimental. También está condensado en unos vídeos cortos en

[4] `https://jderobot.github.io/BehaviorStudio/quick_start/`

Youtube[5] [6].

Con todos los elementos dispuestos, se realiza la validación experimental con los siguientes experimentos (el ✓indica que la prueba funcionó con éxito):

- Carga de la aplicación con la configuración por defecto y el piloto explícito como cerebro inicial. ✓
- Ejecución del cerebro explícito para comprobar las conexiones con el robot y la plataforma a nivel de sensores y actuadores. ✓
- Validación de la visualización sensorial a través de las imágenes aportadas por el robot en la zona de visualización de la plataforma. ✓
- Pruebas de control de ejecución del cerebro: pausa y continuación. ✓
- Pruebas de control de la simulación: carga y descarga del cliente *gzclient* y reinicio de la simulación. ✓
- Pruebas de grabación de *datasets* incluyendo la información de la cámara y de los comandos de velocidad. ✓
- Pruebas de intercambio de cerebros entre el piloto explícito, el piloto títere y el piloto neuronal. ✓
- Pruebas de modificación de código fuente del piloto títere y recarga del cerebro ambos en tiempo de ejecución. ✓

Cabe destacar que la infraestructura utilizada para esta validación proviene de la organización de *software* libre JdeRobot[7]; tanto los robots como los circuitos simulados.

Resultado de esta validación fue el correcto funcionamiento de todas las partes de la aplicación para entornos simulados. Casi todas las funcionalidades expuestas en los objetivos fueron cubiertas con éxito. No obstante, BehaviorStudio está diseñado para funcionar también con robots reales. En el siguiente capítulo se cubre este asunto, con la explicación del otro gran objetivo de este proyecto: conducción autónoma en un robot real mediante un controlador visual basado en *deep learning*. Además, los experimentos de los algoritmos llevados a cabo en esa etapa cubren también la validación del funcionamiento de BehaviorStudio con las partes que esta prueba no cubre; a saber: funcionamiento con robots reales y funcionamiento del modo distribuido (*headless*).

[5] `https://www.youtube.com/watch?v=hvoGjUjBs9w&feature=youtu.be`
[6] `https://www.youtube.com/watch?v=-atqALSPQIU&ab_channel=JdeRobot`
[7] `https://jderobot.github.io`

5

Piloto visual con *Deep Learning*

Introducidos ya todos los elementos de este trabajo, es posible centrarse en el objetivo principal: el desarrollo de un controlador visual basado en *deep learning* para la conducción autónoma de un robot real. En este capítulo se describen todos los elementos del desarrollo de este algoritmo de control visual. Para ello se expondrán los conjuntos de datos utilizados para el entrenamiento del modelo, seguido de los controladores de ROS para los sensores y actuadores del robot real (JetBot) para concluir con la explicación de los detalles de las redes neuronales utilizadas. El capítulo concluye con la validación experimental del modelo construido, en la plataforma BehaviorStudio.

5.1. Conjuntos de datos de entrenamiento

Puesto que el algoritmo que se ha desarrollado está basado en aprendizaje supervisado, se necesita un conjunto de datos adecuado sobre los que entrenar el modelo. En aprendizaje automático existen diferentes técnicas de entrenamiento: aprendizaje supervisado, no supervisado, semi-supervisado, auto supervisado y por refuerzo. En este caso se utiliza el aprendizaje supervisado.

En el aprendizaje supervisado se parte de un conocimiento a priori. El modelo trabaja con datos etiquetados, es decir, para una determinada entrada de datos X, se espera que el modelo ajuste sus pesos adecuadamente para que le asigne una etiqueta $\hat{Y}$ a dichos datos de entrada, ajustándose lo máximo posible a la etiqueta proporcionada Y. Es mediante la diferencia o error entre la etiqueta Y y la predicción del modelo $\hat{Y}$ que el algoritmo es capaz de aprender a base de ejemplos. Este tipo de aprendizajes se suele utilizar en problemas de clasificación, donde las etiquetas son categóricas y de regresión donde las etiquetas son numéricas. Usualmente, el reto del aprendizaje supervisado es la obtención de los datos, ya que para que el modelo sea robusto se necesita una gran cantidad de datos etiquetados que tienen que ser representativos y variados; aunque en la práctica, no todos los problemas necesitan una gran cantidad de datos etiquetados, como es el caso de este proyecto.

Dado que el problema consiste en realizar un controlador visual para el robot real, los datos de entrada del algoritmo son únicamente imágenes captadas por la cámara a bordo del robot. Como se verá más adelante, se va a enfocar el problema del controlador visual como un problema de regresión donde el modelo ha de predecir las coordenadas x e y de la imagen que mejor aproximen la dirección de movimiento del robot, por lo que estas

(a) **(b)** **(c)**

Figura 5.1: (a) Imagen captada por el robot, (b) Ejemplo de imagen etiquetada y (c) Ejemplo de imagen etiquetada con información de dirección.

coordenadas x e y son las etiquetas de las imágenes de entrada, que serán normalizadas antes de entrar al modelo. A partir de las coordenadas x e y, se pueden aproximar valores de potencia para comandar a los motores aproximando un ángulo de giro mediante el cálculo del arco tangente entre las coordenadas inferidas por la red. Este valor es posteriormente modulado por un controlador PD y sumado al valor de velocidad lineal obteniendo un valor aproximado de potencia que entregar para cada uno de los motores. La velocidad lineal es constante y se modula para cada experimento.

En la Figura 5.1a, se puede ver una muestra de lo que el robot «ve» a través de la cámara. Un ejemplo de imagen etiquetada con las coordenadas (x, y) se puede observar en la Figura 5.1b, mientras que una intuición de lo que será el comando de actuación del modelo hacia el robot se puede ver en la Figura 5.1c.

Este conjunto de datos ha sido grabado y etiquetado de forma manual, y consiste en 250 imágenes con resolución 224x224 píxeles, grabadas en diferentes sectores de la pista (rectas, curvas a izquierdas, curvas a derechas, etc.) en diferentes localizaciones con diferente iluminación. Esta variabilidad en la muestra es necesaria para que el modelo sea capaz de generalizar bien ante escenarios que nunca se le han presentado, como otros circuitos u otros entornos diferentes con iluminaciones diversas. En la Figura 5.2 se pueden observar algunas muestras de este conjunto de datos y la variabilidad de las mismas.

El preprocesado de las muestras del conjunto de datos antes de entrar al modelo es mínimo, ya que las única transformaciones que se realizan sobre las imágenes son:

- **Re-escalado** de todas las imágenes de entrada a 224x224 píxeles para que todas las muestras tengan la misma resolución espacial. Se ha seleccionado esta resolución ya que es la que acepta el modelo neuronal que se va a utilizar para resolver el problema. Esto se explica más detalladamente en la sección 5.3.

- **Normalización de la muestra**. La normalización en intensidad aplicada a cada capa de la imagen se realiza a través de la ecuación de la variable estandarizada o normalizada (*z-score* o *default score* en inglés. Esta normalización se define como «el número de desviaciones típicas que un valor dado toma con respecto a la media de

Figura 5.2: Ejemplos de algunas muestras del conjunto de datos A

su muestra o población» según la Wikipedia[1]. En este proyecto, se ha normalizado la muestra utilizando esa fórmula con valores de media y desviación típica calculados a partir del conjunto de datos Imagenet, dado que es el conjunto de datos con el que se han pre-entrenado las redes que se han utilizado en este trabajo. La normalización de la muestra ayuda a obtener datos dentro de un rango y reduce el sesgo que mejora y acelera el aprendizaje de los modelos.

- **Modificación del color** (*color jitter*). Para cada muestra se cambia de forma aleatoria el brillo, el contraste y la saturación. Este tipo de transformaciones ayuda a que las redes generalicen mejor, ya que hacen que la muestra sea independiente de los colores concretos que contenga y que el modelo se centre en otro tipo de información, como en este caso, la forma y disposición de las líneas que delimitan la pista.

- **Normalización de las salidas (etiquetas)**. Como se ha comentado más arriba, se ha enfocado este problema como un problema de regresión, por lo que el modelo tendrá que predecir coordenadas en la imagen. Dado que el rango de valores posibles de salida oscila entre 2 órdenes de magnitud (entre 0 y 224) se requiere una normalización de las etiquetas, como técnica de regularización del modelo. Esta normalización previene que algunos pesos de la red se disparen con respecto a otros porque los valores de las etiquetas normalizadas estarán contenidos en rangos más estándar (típicamente entre 0 y 1, o entre -1 y 1), ayudando a que no se produzca el problema de la explosión de gradientes, además de ayudar en la convergencia.

La forma en la que Pytorch realiza estas transformaciones se aplica en la construcción de cada mini-lote (o *mini-batch* en inglés). Esto quiere decir que en cada iteración durante el entrenamiento, Pytorch extraerá copias de las muestras del conjunto de datos de entrada original (sin adulterar), realizará las transformaciones fijas (como el re-escalado y la normalización, y las transformaciones aleatorias como los cambios en los niveles de brillo, contraste y saturación (*color jitter*). Este carácter aleatorio de las transformaciones, hace de éstas una técnica de aumento de datos, ya que se aplican en cada iteración durante el entrenamiento por lo que en cada época, estas transformaciones pueden ser diferentes ante la misma muestra de entrada.

Adicionalmente al conjunto de datos de 250 imágenes con gran variabilidad, se ha grabado otro conjunto aún mayor para tratar de experimentar con diferentes conjuntos de datos. Este nuevo conjunto de datos consta de 550 imágenes con las mismas propiedades que el conjunto de datos principal, pero con la única diferencia de que está sesgado. Todas las imágenes de este nuevo conjunto de datos están grabadas en el mismo escenario (la misma habitación), con una iluminación relativamente uniforme. La Figura 5.1, contiene tres muestras de este conjunto de datos. Todos los conjuntos de datos tienen las mismas propiedades y se le realizan las mismas transformaciones. En resumen, se dispone de 2 conjuntos de datos diferentes:

- **Conjunto de datos A**. Compuesto de 250 imágenes variadas en cuanto a iluminación, y entornos, de diferentes tramos de la pista. Este conjunto de datos está balanceado con respecto a los diferentes tipos de tramos: rectas, curvas a izquierda y curvas a derecha. También está balanceado en cuanto a variabilidad de fondos y de iluminación.

[1] `https://es.wikipedia.org/wiki/Unidad_tipificada`

- **Conjunto de datos B**. Compuesto de 550 imágenes con iluminación y fondos similares. Está balanceado en cuanto a tipos de tramos de la pista: rectas, curvas a izquierda y curvas a derecha, pero está sesgado en cuanto a variabilidad en iluminación y fondos.

5.2. *Drivers* JetBot

Para poder conectar el *hardware* del robot con la plataforma BehaviorStudio y los diferentes cerebros neuronales que gobernarán al mismo, es necesario un mecasnimo de comunicación entre ambas partes. En el caso del robot simulado, los propios modelos y la propia infraestructura de JdeRobot incluía resueltas esas conexiones, al implementar cada sensor y actuador del robot interfaces de comunicación a través de *topics* de ROS. No obstante, no se dispone de esa facilidad en el caso del robot real. Por este motivo, se ha desarrollado un controlador capaz de conectar los sensores del robot JetBot (la cámara) y los actuadores del mismo (los motores) basado también en *topics* de ROS para habilitar la comunicación entre el robot y la plataforma BehaviorStudio.

La imagen que sirve como sistema operativo del robot, que se ha utilizado (JetPack 4.3) dispone de un API en Python para acceder a todos los elementos del robot: cámara, motores, pantalla OLED, batería, etc. Se ha desarrollado un envoltorio que encapsula esa funcionalidad recubriéndola con lógica de *topics* de ROS, de tal forma que se han creado interfaces para la cámara y los motores. Este desarrollo se ha inspirado en un envoltorio ya existente del JetBot para ROS[2], que resuelve el problema de las comunicaciones. No obstante, después de las pruebas realizadas sobre este código se comprobó que las interfaces no funcionaban del todo bien, ya que la imagen de la cámara se obtenía rotada 180 grados y el movimiento de los motores no funcionaba bien en ocasiones. Por este motivo se optó por desarrollar un híbrido entre el código del *driver* de ROS existente para el JetBot y los *drivers* que ofrece la propia imagen JetPack del robot, obteniendo una iteración sobre las dos APIs con pequeños cambios que solucionaban los problemas mencionados. En la Figura 5.3 se puede apreciar un diagrama de la arquitectura de este componente.

Los *topics* asociados a cada sensor/actuador son similares a los utilizados en el robot simulado. Para la cámara, el *topic* utilizado transporta el mismo tipo de mensajes que para el robot real (`std_msgs/Image`) mientras que para los comandos de velocidad lineal y angular de los motores del robot se utilizan mensajes de tipo Float32 de ROS (`std_msgs/Float32.msg`).

Este controlador se ejecuta mediante *scripts* de Python, de tal forma que se pueda automatizar el lanzamiento de los algoritmos mediante el cron de Linux o convirtiéndolos en servicios del sistema para que estén siempre operativos. Una vez funcionando, la conexión con la plataforma es directa, ya que está preparada para aceptar conexiones mediante *sockets* utilizando ROS, al igual que el entorno de simulación.

[2] `https://github.com/dusty-nv/jetbot_ros`

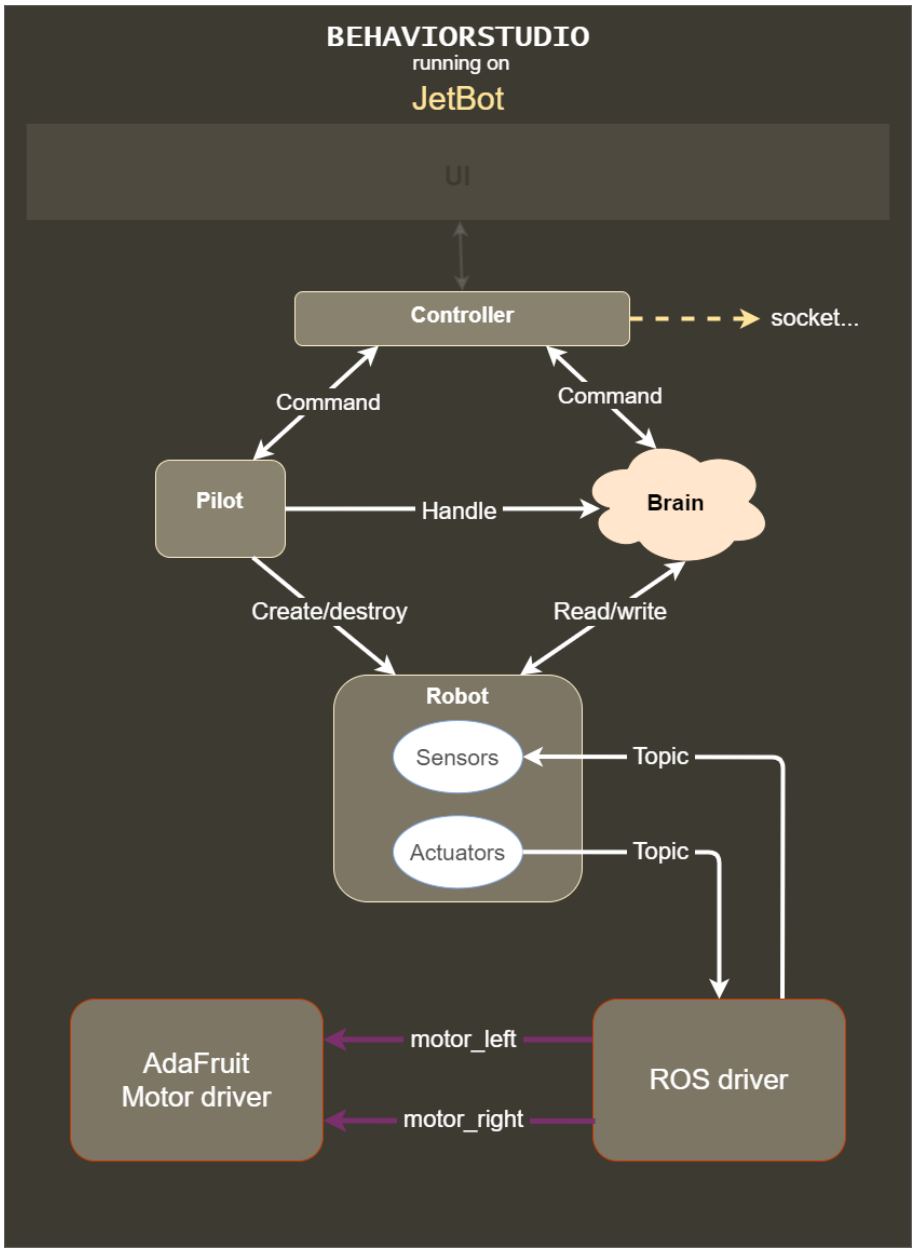


Figura 5.3: Arquitectura de BehaviorStudio corriendo en el robot JetBot

5.3. Redes de regresión para control visual de un robot

Muchas redes neuronales son usadas típicamente para resolver tareas de detección y clasificación. En visión artificial las redes neuronales convolucionales de detección son las más conocidas; arquitecturas como la YOLO que han demostrado ser muy potentes para la clasificación de objetos en imágenes, se utiliza en multitud de trabajos de visión en la comunidad investigadora e incluso en las empresas. No obstante, las redes neuronales también son muy útiles en problemas de regresión. El problema de la regresión consiste en predecir el valor de una variable continua en función del valor de otras variables independientes.

En el caso de las redes neuronales, la regresión se aplica en problemas en los que las variables a predecir no pueden ser categorizadas en diferentes clases; el espacio de posibles predicciones no es discreto. Se considera que la regresión es un problema de clasificación de infinitas clases, ya que trata de modelar la relación existente entre una variable continua (dependiente) que se trata de predecir, en base a otras variables (independientes) que son las entradas al modelo.

Las redes neuronales de regresión en visión artificial se suelen utilizar para solucionar problemas de segmentación a nivel de píxel, estimación de *bounding boxes* en una imagen, detección de pose humana, etc. En este trabajo se utiliza la regresión para estimar las coordenadas XY a las que el robot debería estar mirando, de tal forma que si el robot está desviado de su trayectoria en una curva o una recta, se corrija su posición mediante la inferencia. Es decir, se predicen las coordenadas de la imagen donde se supone que el robot debería dirigirse. Estos valores están directamente relacionadas con las velocidades de avance v y de giro w que se van a ordenar a los motores.

La resolución del problema se basa en alimentar a la red con las imágenes captadas por el robot en cada momento, de tal forma que se infieran las coordenadas XY de la imagen que representa la dirección a la que se debe mover el robot. Como se comentó en secciones anteriores, las etiquetas de los datos ha sido normalizadas a valores entre -1 y 1, por lo que el modelo inferirá las coordenadas con esa normalización. A partir de dichas coordenadas se aproximará un ángulo de giro mediante el cálculo de la arcotangente con signo entre ambas coordenadas de la inferencia. Este ángulo será corregido mediante un controlador PD (Proporcional-Derivativo), para dotar al movimiento de suavidad y que no sea tan reactivo ante cambios. Además a la ecuación se suma el sesgo de las ganancias de los motores, para corregir desviaciones en la trayectoria naturaleza de la limitación de los motores. La ecuación resultante es:

$$pd\_val = angle * kp + (angle - angle_last) * kd$$

aplicando la correción del sesgo:

$$steer = pd\_val + steer\_bias$$

aplicando una fuerza final a cada motor resultado de la suma de la velocidad lineal y la velocidad angular calculada en las ecuaciones anteriores con límites entre 0 y 1, de modo

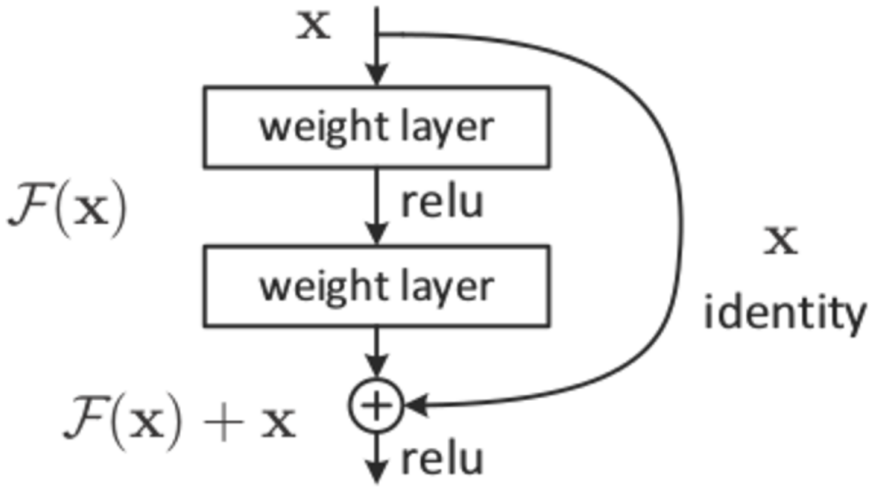


Figura 5.4: Bloque residual de la arquitectura ResNet.

que si el valor agregado de velocidad lineal y angular supera el umbral por encima o por debajo, se apliquen los valores máximo o mínimo respectivamente.

Para solucionar el problema se han seleccionado dos tipos de arquitecturas de redes neuronales, una basada en redes residuales y otra basada en redes convolucionales convencionales: ResNet y MobileNet respectivamente. En las siguientes secciones se ofrece una descripción breve de ambas arquitecturas.

5.3.1. Arquitectura ResNet

La arquitectura ResNet surgió en 2015 [45] con la idea de solucionar el problema que sufrían todas las redes de clasificación del momento, que consistía en que las repuestas se degradaban exponencialmente al aumentar la profundidad de las redes neuronales, el conocido «desvanecimeinto del gradiente». El problema se solucionó con la inclusión de determinadas conexiones que «saltaban» capas de las redes tradicionales, permitiendo aumentar el número de capas sin que el rendimiento de la red se viera afectado, y solucionando así el problema del desvanecimiento del gradiente. Estas conexiones fueron implementadas mediante «bloques residuales» formados esencialmente de dos capas convolucionales y una conexión que suma la salida de esas capas y la entrada, como se aprecia en la Figura 5.4

Existen numerosas variantes de esta arquitectura con diferentes número de capas, desde 18 hasta 152. En este trabajo se exploran dos de ellas: la arquitectura ResNet-18 y la arquitectura ResNet-34. Estas arquitecturas se muestran en la Figura 5.5

Pytorch ofrece multitud de redes pre-entrenadas[3] que están listas para ser utilizadas sin necesidad de un ajuste fino si no se requiere. Estos modelos pueden ser instanciados a través de la biblioteca `torchvision` que está dedicada a la visión artificial. En concreto, las redes utilizadas en este trabajo (la ResNet-18 y la ResNet-34) están pre-entrenadas con el conjunto de datos ImageNet para la tarea de clasificación de imágenes. Dado que

[3] `https://pytorch.org/docs/stable/torchvision/models.html`

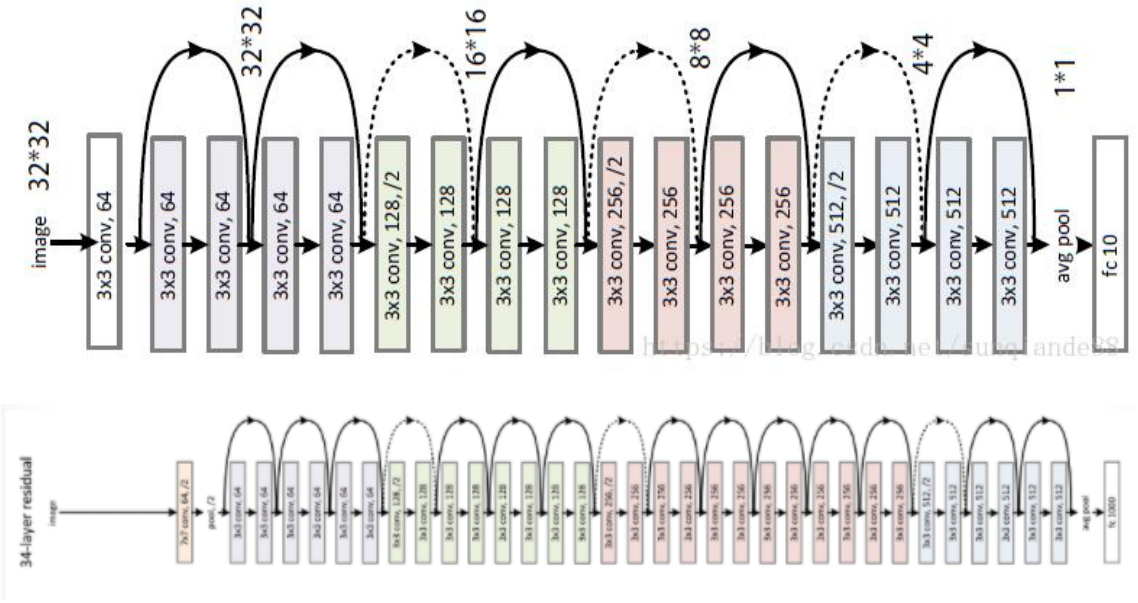


Figura 5.5: Arquitecturas ResNet-18 (arriba) y ReNet-34 (abajo)

se va a abordar el problema de la conducción autónoma como una tarea de regresión, es necesario hacer una transferencia de dominio del conocimiento de estas redes, de tal forma que aprendan a realizar inferencia sobre nuestro problema en concreto. La forma de instanciar estos modelos pre-entrenados es posible a través del API de la biblioteca `torchvision` como muestra el Fragmento 5.1

```
1  import torchvision
2
3  # instancia de una ResNet-18 pre-entrenada
4  model = torchvision.models.resnet18(pretrained=True)
5
6  # instancia de una ResNet-34 pre-entrenada
7  model = torchvision.models.resnet34(pretrained=True)
8
9  # instancia de una red MobileNet v2
10 model = torchvision.models.mobilenet_v2(pretrained=True)
```

Fragmento 5.1: Ejemplo de instanciación de diferentes modelos pre-entrenados.

5.3.2. Arquitectura MobileNet

Los modelos basados en redes neuronales más grandes y potentes requieren de una potencia de cómputo altísima tanto para entrenar como para realizar inferencias. No obstante, en los últimos años han surgido modelos basados en redes neuronales que están diseñados para funcionar en dispositivos móviles y de recursos limitados como los los sistemas embebidos. Este es el caso de MobileNet.

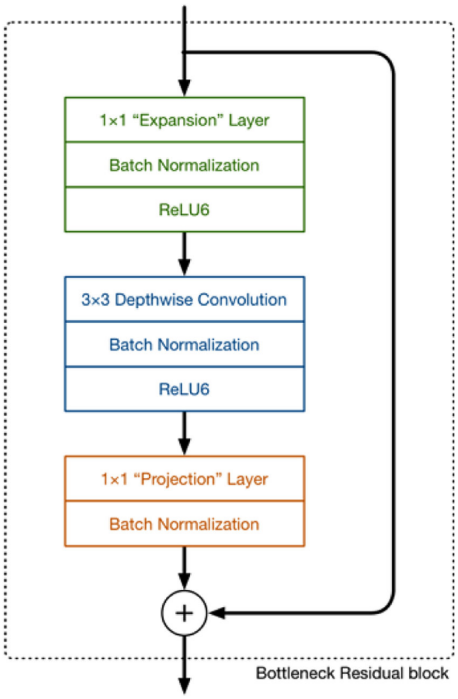


Figura 5.6: Bloque residual *bottleneck* de la arquitectura MobileNet v2.

Para conseguir esto, Mobilenet utiliza los llamados bloques *bottleneck*, que se pueden observar en la Figura 5.6. Este bloque básicamente realiza una factorización sobre la convolución normal separándola en varias capas. La primera capa llamada *depthwise convolution* aplica un único filtro por canal de la imagen, posible al ser precedido por una convolución 1x1 previamente. La capa siguiente recibe el nombre de *pointwise convolution* y se encarga de crear características nuevas en base a la combinación lineal de los canales de entrada de la imagen.

Al igual que con las arquitecturas ResNet, Pytorch ofrece también la arquitectura MobileNet v2. Esta red también está pre-entrenada con el conjunto de datos ImageNet para la clasificación de imágenes, lista para ser utilizada. Esta arquitectura está disponible en el mismo sitio que las ResNet mencionadas en arriba y se instancia como se muestra en el Fragmento 5.1.

5.4. *Transfer Learning*

Una técnica que se puede aplicar en el *deep learning* es la transferencia de aprendizaje, comúnmente conocida como *transfer learning*. La técnica de *transfer learning* consiste en seleccionar una red neuronal ya entrenada con un conjunto de datos determinado y usarla como punto de partida para que la red aprenda a desarrollar otra tarea diferente. Por ejemplo, se puede tomar un modelo neuronal sin entrenar como la ResNet-18, entrenarla con un conjunto de datos para una tarea concreta (ImageNet para clasificación de imágenes, por ejemplo) obteniendo una red entrenada, a la que denominaremos ResNet-18 en este ejemplo, para dicha tarea. Durante el proceso de aprendizaje, ResNet-18 habrá aprendido un amplio conjunto de filtros de características que pueden ser reutilizadas pa-

ra tareas parecidas, por lo que en este momento se puede partir de esa red pre-entrenada para que aprenda a realizar otra tarea (como la de la estimación de coordenadas en una imagen) con otro conjunto de datos diferente que con los que fue pre-entrenada. El modelo resultante sobre el que se ha aplicado *transfer learning* se distingue en este documento con un asterisco «*», de tal forma que en este ejemplo concreto el modelo re-entrenado se denominará ResNet-18*. Esta técnica otorga mejoras significativas en el proceso de aprendizaje, ya que entre otras cosas requiere menos ajuste de hiperparámetros, ayuda a la convergencia, está validado el buen rendimiento de la arquitectura de antemano y suele requerir un entrenamiento menos exhaustivo.

Para entrenar los modelos seleccionados para este trabajo, se ha aplicado *transfer learning* por los motivos expuestos más arriba. A pesar de que los modelos están ya entrenados con un dominio específico, en este caso con el conjunto de datos de ImageNet, es necesario que aprenda a resolver la tarea de la conducción autónoma, por lo que hay que reentrenar los 3 modelos con un conjunto de datos del problema concreto a resolver. Este reentrenamiento requiere primero de unos datos concretos (se describieron en la sección 5.1) y segundo de un ajuste de algunos hiperparámetros como el tamaño de los lotes que van a entrar al modelo, el número de épocas de entrenamiento o el porcentaje de datos que se usarán para entrenar y validar el modelo. Durante el desarrollo de el trabajo se han probado diferentes configuraciones sobre el circuito de pruebas (el circuito *Training* que se introduce en la siguiente sección). Dado que se ha aplicado *transfer learning*, los únicos parámetros que se han modificado han sido los necesarios para el reentrenamiento: el número de épocas de entrenamiento, el tamaño del mini-lote, el optimizador y la proporción en los conjuntos de datos de entrenamiento y de pruebas. Para los 3 modelos entrenados se han definido los mismos parámetros, por simplicidad: 70 épocas de entrenamiento, tamaño de mini-lote de 8, optimizador Adam con parámetros por defecto y una proporción de los conjuntos de entrenamiento-test de 80-20 (80 % de los datos utilizados para el entrenamiento y el 20 % restante para test).

El segundo experimento realizado se ha llevado a cabo mediante el entrenamiento de una red ResNet-18. Como se explicó en la sección 5.3.1, las redes residuales que ofrece Pytorch han sido entrenadas con el conjunto de datos de ImageNet, por lo que se ha aplicado la técnica de *transfer learning*. Se ha demostrado que esta técnica aumenta de forma considerable el rendimiento de las redes a nivel de inferencia ya que han sido pre-entrenadas con conjuntos de datos muy grandes y muy bien construidos, por lo que los filtros que se aprendieron durante ese entrenamiento aportan un conocimiento a priori que se aprovecha al realizar el cambio de dominio. Las arquitecturas ResNet que ofrece Pytorch pre-entrenadas varían desde las 18 capas hasta las 152. Para este proyecto se ha utilizado la red más pequeña, la ResNet-18, que proporciona un buen equilibrio de rendimiento y eficiencia para la Jetson Nano.

Tras los reentrenamientos respectivos se obtuvieron sendas redes de regresión: ResNet-18*, ResNet-34* y MobileNet-v2* ya adaptadas a la conducción automática del robot JetBot.

5.4.1. Métricas de evaluación de los entrenamientos

Es necesario definir algún tipo de métrica para evaluar cómo de bien o de mal han ido los entrenamientos de las diferentes redes de regresión. En este caso, las métricas más

típicas de validación que se utilizan para este tipo de redes son las de el error cuadrático medio (MSE o *Mean Squared Error* por sus siglas en inglés) y el error absoluto medio (MAE o *Mean Absolut Error* por sus siglas en inglés). Este tipo de métricas se calculan mediante la diferencia entre las etiquetas de los datos y las predicciones de la red.

La métrica MAE se calcula como el promedio de la diferencia entre los valores de las etiquetas y los valores predichos por la red ante una entrada determinada. Esta medida ayuda a determinar cómo de diferentes son estos dos valores, ya que un error alto indica que la red no se está acercando al valor real que se espera (la diferencia es mayor), mientras que un error bajo indica que las predicciones de la red y los valores reales se parecen. No obstante, esta métrica no aporta información sobre la dirección del error, por lo que no se puede determinar si el error es provocado por predicciones que estén por encima o por debajo del valor de la etiqueta. La ecuación matemática del MAE se expresa como sigue:

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

Donde N es el número total de muestras a entrenar, y_i es el valor de la etiqueta de la muestra i-ésima, e $\hat{y}_i$ es el valor predicho por la red para la muestra i-ésima.

Por otra parte, la métrica MSE se calcula como el promedio del cuadrado de la diferencia entre los valores de las etiquetas y los valores predichos por la red. Las métricas son muy similares, pero la ventaja de MSE sobre MAE es que el primero facilita el cálculo del gradiente. Esto es así ya que los fallos en las predicciones penalizan mucho más al calcular el cuadrado del error, por lo que la diferencia entre errores grandes y errores pequeños es más pronunciada. La ecuación matemática del MSE se calcula como sigue:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Donde, al igual que para MAE, N es el número total de muestras a entrenar, y_i es el valor de la etiqueta de la muestra i-ésima, e $\hat{y}_i$ es el valor predicho por la red para la muestra i-ésima.

Tanto MAE como MSE ayudan a determinar cómo de bien ha ido el entrenamiento de las redes neuronales. Este tipo de métricas de entrenamiento son muy útiles para determinar si las redes tienen algún problema como capacidad de aprendizaje, sobreajuste, subajuste, etc. Teniendo un diagnóstico en tiempo real del entrenamiento es posible implementar técnicas como la parada temprana (*Early stopping*), la planificación de la tasa de aprendizaje (*scheduler*), entre otras, que ayudan a una detección temprana de problemas en las redes, dado que, por norma general, las redes neuronales profundas requieren grandes tiempos de entrenamiento. En la Tabla 5.1 se recoge las métricas promedio de las redes reentrenadas: Resnet18, ResNet-34 y MobileNet v2.

Es posible que aunque una red arroje buenas métricas de evaluación en el entrenamiento, no se refleje en un buen comportamiento del robot real con ella. Es decir, se puede dar el caso de que a pesar de que las métricas de entrenamiento resulten prometedoras, la ejecución real no sea capaz de solucionar el problema para el que fue entrenada provocando

Red	*Mean Absolute Error*	*Mean Squared Error*
Resnet-18	0.538007	0.098720
Resnet-34	0.627700	0.369343
MobileNet v2	0.594553	0.259610

Tabla 5.1: Promedio de las métricas de entrenamiento para las redes utilizadas.

que el robot no sea capaz de completar el circuito. Este es el caso de la red Resnet-34, que aunque arroje resultados similares al resto de las redes probadas, no ha sido capaz de completar ningún circuito. Esto se debe a que el cálculo del error se hace de forma promedio, de tal forma que algunas predicciones pueden ser muy buenas, pero otras pueden ser muy malas, lo que se camufla al realizar la media. Por eso estas métricas pueden dar falsa sensación de éxito. No obstante, en este caso concreto, el problema ha sido el tamaño de la red. La red ResNet-34 es casi el doble de grande que la ResNet-18. Se ha comprobado experimentalmente que la ResNet-18 pone casi al límite las capacidades de la Jetson Nano en cuanto a recursos de CPU y memoria, por lo que la razón principal por las que la ResNet-34 no ha podido completar ningún circuito es por la latencia introducida en cada iteración para obtener una inferencia (llegando a tardar casi 1 segundo en tomar una decisión en cada iteración que es un retardo inaceptable en nua aplicación de control robótico).

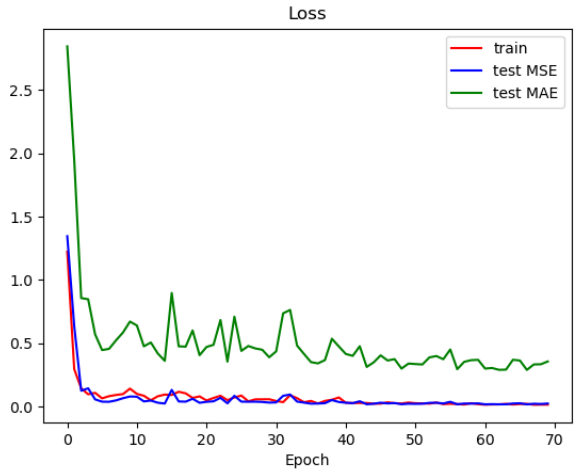
Las métricas de validación explicadas dan buenas pistas sobre cómo ajustar los parámetros de entrenamiento. En las Figuras 5.7 se pueden observar los valores de estas métricas en el proceso de reentrenamiento en las diferentes redes. La Tabla 5.1, muestra el promedio de esas métricas para las 70 épocas de reentrenamiento.

Como se puede observar, las redes ResNet convergen muy pronto (en torno a las 4 épocas), mientras que MobileNet oscila algo más. No obstante, la pérdida en los tres casos es muy baja, por lo que el error cometido es poco. Esto indica que los parámetros de entrenamiento están bien ajustados. Además, al haber aplicado la técnica de *transfer learning*, el entrenamiento es más rápido y más eficiente. En este caso concreto, se podría haber implementado una parada prematura del entrenamiento, ya que en el caso de las ResNet, a partir de la época 4-5 no se mejora más al estar el error ya muy cercano a cero. En el caso de MobileNet, el entrenamiento no se estabiliza hasta la época 50 aproximadamente. Esto se debe a que el modelo es más pequeño y no generaliza tan bien como los dos anteriores a pesar de haber aplicado *transfer learning* igualmente. No obstante, se ha conseguido que el entrenamiento sea exitoso al no haberse producido ningún problema de subajuste o sobreajuste.

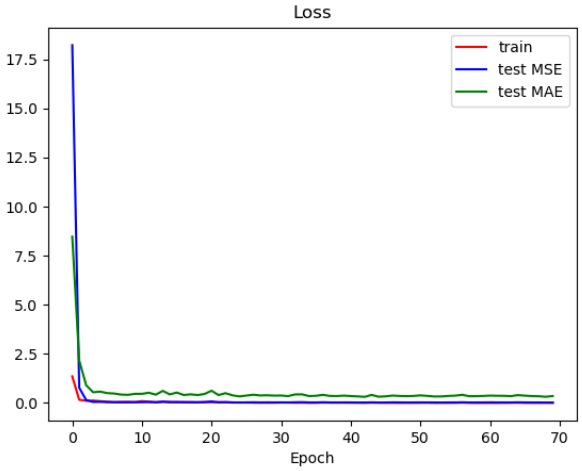
En la siguiente sección se validará el desempeño de los modelos reentrenados para resolver la tarea en entornos reales. Se obvia la inclusión de la red ResNet-34 ya que no ha conseguido completar ningún circuito debido a problemas de rendimiento con la placa Jetson Nano.

5.5. Validación experimental

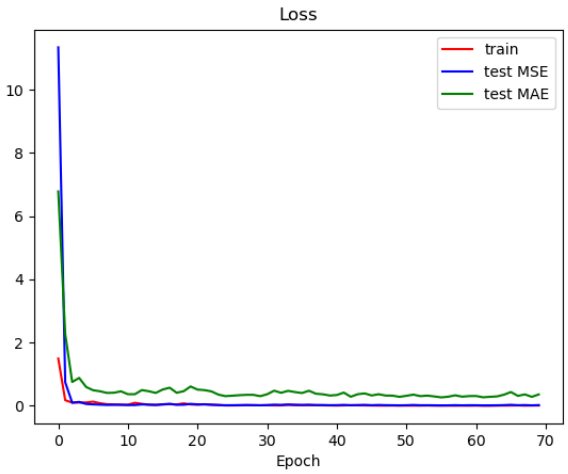
Una vez reentrenados los modelos neuronales del controlador visual como se ha explicado en las secciones anteriores, en esta sección se explican los experimentos que validan



(a) MobileNet v2



(b) ResNet-18



(c) ResNet-34

Figura 5.7: Métricas de pérdida en entrenamiento en 70 épocas de las diferentes redes utilizadas.

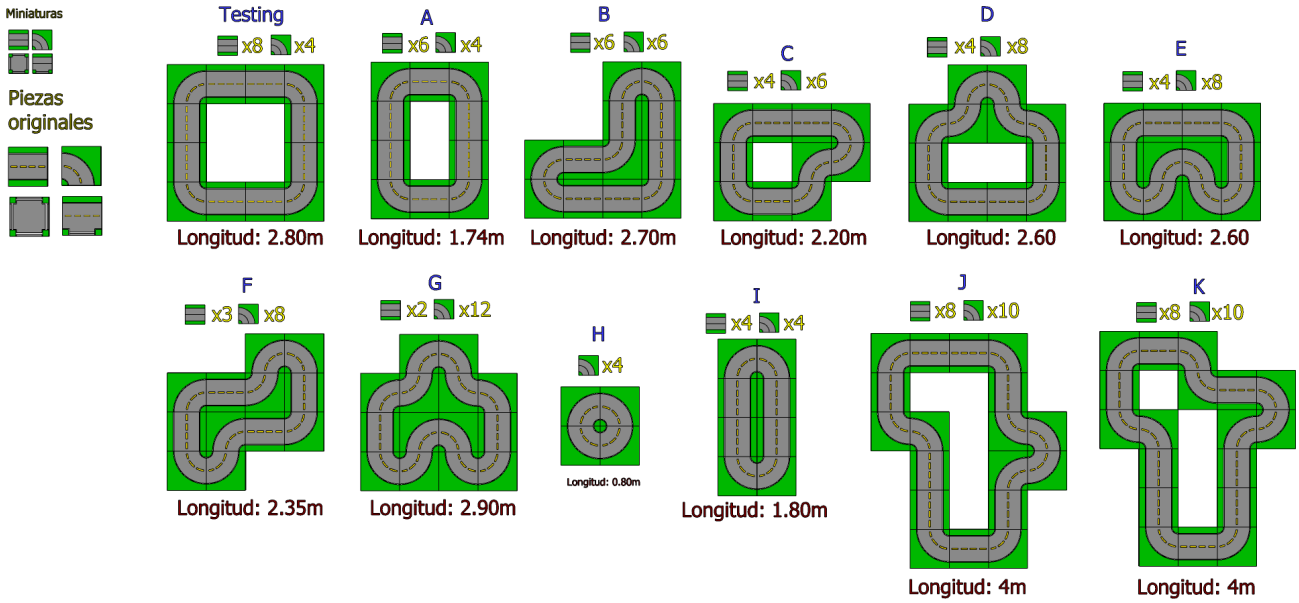


Figura 5.8: Diferentes circuitos a resolver por el robot.

el trabajo realizado. Como se explicó en el capítulo de introducción, el objetivo es que un robot real sea capaz de completar vueltas a un circuito, o varios en este caso, utilizando como algoritmo de control los modelos neuronales basados en *deep learning* conseguidos. A continuación se describen los diferentes circuitos sobre los que se ha validado el trabajo, las métricas de evaluación utilizadas para validar el correcto funcionamiento de los modelos, los experimentos realizados y los resultados de dichos experimentos.

En la sección 5.4.1 se explicaron las métricas que se utilizan para validar el entrenamiento de los modelos. No obstante, aún falta definir algún tipo de métrica operativa, que refleje la realidad del aprendizaje conseguido, es decir, poner a prueba las capacidades de la red aprendida. Para ello se define una métrica operativa que será el tiempo que tarda el robot en completar una vuelta al circuito; de este modo, sabremos cómo se comportan las redes entrenadas en ejecución.

5.5.1. Circuitos de test

Para validar que el robot puede solucionar la tarea de la conducción autónoma, es necesario algún tipo de carretera que seguir. Para este propósito se han diseñado diferentes circuitos pequeños a partir de piezas de Lego, como se apuntó en la introducción (Figura 3.5). En concreto se han diseñado 12 circuitos diferentes con configuraciones variadas de curvas y rectas. La longitud de cada segmento recto mide 25 *cm*, mientras que las curvas tienen una longitud de unos 20 *cm* aproximadamente. Todos los segmentos rectos son exactamente iguales, al igual que ocurre con los segmentos curvos. En la Figura 5.8, se pueden observar los circuitos propuestos.

Como se puede observar, cada circuito tiene un número diferente de curvas y rectas dispuestos de menor a mayor dificultad, de tal forma que se pueda evaluar la robustez de los modelos entrenados ante circuitos nunca vistos. En las secciones de ejecución típica

Circuito	Longitud (metros)	# Rectas	# Curvas
Training	2.80	8	4
A	1.74	6	4
B	2.70	6	6
C	2.20	4	6
D	2.60	4	8
E	2.60	4	8
F	2.35	3	8
G	2.90	2	12
H	0.80	–	4
I	1.80	4	4
J	4.00	8	10
K	4.00	8	10

Tabla 5.2: Propiedades de los 12 circuitos propuestos.

se muestran los resultados en cuanto a tiempo por vuelta de cada uno de los circuitos propuestos. La Tabla 5.2 muestra un resumen de las características de cada uno de los circuitos. El circuito de *Training* recibe ese nombre ya que es en ese circuito en el que se probaron los modelos hasta conseguir el ajuste óptimo de los hiper-parámetros de las redes.

La evaluación realizada consiste en que el robot sea capaz de completar una vuelta a cada uno de los circuitos en sentido horario y en sentido antihorario. Esto ha de aplicarse a cada uno de los dos modelos reentrenados: Resnet18* y MobileNet-v2*.

5.5.2. Ejecución típica MobileNet-v2*

En este primer experimento se ha hecho uso de una arquitectura MobileNet v2 para entrenar una red con los conjuntos de datos disponibles. Como se explicó en la sección 5.3.2, este tipo de arquitectura está diseñada para ser implementadas en dispositivos pequeños, como placas de procesamiento tipo RaspberryPi, Serie Jetson y dispositivos móviles. Este ha sido el motivo por el cual se ha seleccionado este tipo de arquitectura para tratar resolver el problema de la conducción autónoma en un robot real, ya que la potencia de cómputo de la Jetson Nano es poca en comparación con un computador de escritorio, además de que sus recursos tanto de memoria, CPU, como de alimentación son muy limitados. Por eso se requiere que el tiempo de inferencia sea lo más rápido posible, ya que se trata de resolver un problema de navegación en tiempo real, imposible de lograr con arquitecturas más profundas.

Como se ha explicado en la sección 5.4, los modelos han sido entrenados con 2 tipos diferentes de conjuntos de datos, cada uno dividido en conjunto de entrenamiento y conjunto de pruebas. Los conjuntos de datos utilizados para entrenar estos modelos son los A y B mencionados en la sección 5.1 con un ratio de entrenamiento-test de 80-20 (80 % de los datos utilizados para el entrenamiento y el 20 % restante para test). Los hiper-parámetros de la red son idénticos para los entrenamientos con los diferentes conjuntos de datos, para

ilustrar de forma certera el impacto de los diferentes conjuntos sobre la red.

Comparado con el entorno simulado donde las condiciones son controladas, en entornos reales no se tiene control total de las condiciones en las que el robot va a trabajar. Esto quiere decir que existen factores externos que pueden comprometer el funcionamiento de los modelos entrenados. En el caso particular de este trabajo, algunos de esos factores son:

- **Iluminación**. La iluminación en los experimentos no ha sido uniforme en todas las ejecuciones debido a que los experimentos fueron llevados a cabo a diferentes horas del día en una habitación con ventanas al exterior. Los reflejos, la intensidad lumínica, etc. pueden provocar que los modelos que funcionan bien en simulación dejen de hacerlo al trasladarlo a un entorno real.

- **Desniveles en la pista**. El suelo en el que se han construido las pistas para los experimentos, está hecho de un material rugoso (una especie de moqueta), que no está nivelada en todos los puntos. Es esta irregularidad la que ha provocado desniveles en distintos tramos de la pista, lo que afecta a la conducción al no haber realimentación de falta de velocidad durante el comportamiento para estos casos.

- **Par motor**. Los motores incluidos en el kit del JetBot son motores de continua de juguete. Estos motores suelen utilizarse para juguetes de control remoto, por lo que no son especialmente potentes, especialmente a velocidades bajas. La falta de par motor hace que el robot no sea capaz de operar a velocidades bajas, provocando la pérdida de control fino al navegar por una pista estrecha como la que se ha utilizado en este problema. Ha sido necesario ajustar las ganancias de los motores en cada ejecución para encontrar la configuración óptima para el funcionamiento correcto en cada circuito. El control PID absorbe en cierta medida esa heterogeneidad de bajo nivel.

- **Superficie**. El material con el que están construidas las pistas es plástico, esto provoca que, en ocasiones, el agarre de los neumáticos del robot no sea del todo bueno. La falta de par de los motores del robot junto con la superficie sobre la que se mueve y los desniveles, hace que en determinados tramos en determinadas pistas el robot se pare, haciendo falta intervención externa (un pequeño empujón) para que el robot continúe con su trayectoria. Además, dado que las pistas utilizadas son de Lego City, cada módulo de curva y recta tiene pequeñas protuberancias (para acoplar otras piezas de Lego) que provocan que el robot pierda control sobre el movimiento al salirse del carril.

En las Tablas 5.3 y 5.4 se reflejan los resultados de la red MobileNet-v2* cuando se reentrena con los conjuntos de datos A y B respectivamente.

Se puede observar que en ambas tablas las vueltas en sentido horario se completan más rápido que en sentido antihorario sistemáticamente. Esto es debido a uno de los factores mencionados más arriba, en concreto los desniveles del circuito sumados a la poca potencia de los motores a velocidades bajas. Se asume que los parámetros de las ganancias de los motores están optimizados para cada ejecución, ya que se han ajustado manualmente en cada una de las pruebas realizadas. A grandes rasgos, se observa también que la red entrenada con el conjunto de datos A ofrece mejores resultados tanto en tiempo como en número de circuitos completados que la red entrenada con el conjunto de datos B. Esto se puede deber al sesgo de los datos durante el entrenamiento, ya que el conjunto de datos B es

Circuito	Longitud (m)	Tiempo sentido h (s)	Tiempo sentido ah (s)
Training	2.80 m	19.6"	21.9"
A	1.74 m	14.5"	15.7"
B	2.70 m	15.9"	16.5"
C	2.20 m	13.4"	15.7"
D	2.60 m	16.2"	17.8"
E	2.60 m	13.5"	14.1"
F	2.35 m	13.2"	14.0"
G	2.90 m	-	-
H	0.80 m	1.2"	1.3"
I	1.80 m	6.5"	7.0"
J	4.00 m	-	-
K	4.00 m	-	-

Tabla 5.3: Tiempos por vuelta en los 12 circuitos utilizando la red MobileNet-v2* con el conjunto de datos A (los - indican que el modelo no ha conseguido completar una vuelta)

Circuito	Longitud (m)	Tiempo sentido h (s)	Tiempo sentido ah (s)
Training	2.80 m	22.1"	23.9"
A	1.74 m	15.9"	16.1"
B	2.70 m	-	-
C	2.20 m	14.1"	-
D	2.60 m	-	-
E	2.60 m	-	15.6"
F	2.35 m	14.2"	15.4"
G	2.90 m	-	-
H	0.80 m	0.9"	1.0"
I	1.80 m	6.8"	6.8"
J	4.00 m	-	-
K	4.00 m	-	-

Tabla 5.4: Tiempos por vuelta en los 12 circuitos utilizando la red MobileNet-v2* con el conjunto de datos B (los - indican que el modelo no ha conseguido completar una vuelta)

muy homogéneo en cuanto a niveles de iluminación y fondos de las imágenes. La red puede haber aprendido características comunes a esas imágenes que deterioran el rendimiento de la red en otros entornos. En cambio, en el conjunto de datos A la variabilidad es mayor, por lo que ese efecto se minimiza.

Como se puede observar, con el conjunto de datos A, la red es capaz de resolver la mayor parte de los circuitos. No obstante, hay 3 de ellos (el G, el J y el I) en los que no ha sido capaz de completar ninguna vuelta. Las razones pueden ser variadas, pero en los experimentos se ha observado que en las chicanes el robot se salía. Se ha comprobado experimentalmente que el motor derecho requiere de más potencia para trabajar al mismo nivel que el izquierdo, por lo que existe un desajuste entre ambos que se ha intentado paliar modificando los parámetros de las ganancias sin éxito. Otra de las razones posibles son los cambios de iluminación y los brillos especulares muy intensos que había en algunos tramos de las pistas que introducen inferencias espurias que pueden desviar la trayectoria del robot de forma irrecuperable.

Con el conjunto de datos B, se puede observar claramente que la falta de variabilidad de los datos es determinante en el rendimiento de la red. Como se observa, el robot no

es capaz de completar muchos de los circuitos. A todos los problemas mencionados anteriormente se suman las inferencias pobres debido al cambio de entorno. La conclusión de este experimento es que aunque se disponga de una gran cantidad de datos para reentrenar (550 imágenes contra 250), si éstas no son representativas del problema a resolver, el funcionamiento de la red no será el adecuado; por lo tanto, se ha demostrado que la variabilidad en los datos es clave para este problema en particular.

5.5.3. Ejecución típica ResNet-18*

Como en el experimento anterior, esta red ha sido reentrenada con ambos conjuntos de datos A y B. También se ha reentrenado al modelo con los mismos hiper-parámetros para ambos conjuntos con el objetivo de comparar el desempeño de la red con conjuntos de datos diferentes y refrendar o no las conclusiones obtenidas en el primer experimento. La única diferencia entre ambos experimentos, además de la arquitectura utilizada, han sido las horas del día en la que se han realizados los experimentos; este factor también puede influir en los diferentes resultados debido a los cambios de iluminación.

Al igual que sucedía en el experimento anterior, cada experimento ha requerido del ajuste de las ganancias de los motores para compensar el desajuste de par entre los motores derecho e izquierdo.

En las Tablas 5.5 y 5.6 se pueden observar los resultados de los experimentos llevados a cabo con la red ResNet-18 entrenada con los conjuntos de datos A y B respectivamente:

Circuito	Longitud (m)	Tiempo sentido h (s)	Tiempo sentido ah (s)
Training	2.80 m	18.4"	21.7"
A	1.74 m	13.9"	14.8"
B	2.70 m	14.3"	14.9"
C	2.20 m	11.4"	11.9"
D	2.60 m	14.0"	14.5"
E	2.60 m	13.7"	14.0"
F	2.35 m	12.4"	12.5"
G	2.90 m	-	-
H	0.80 m	1.4"	1.4"
I	1.80 m	7.4"	8.2"
J	4.00 m	18.4"	19.7"
K	4.00 m	18.5"	20.1"

Tabla 5.5: Tiempos por vuelta en los 12 circuitos utilizando la red Resnet-18* con el conjunto de datos A (los - indican que el modelo no ha conseguido completar una vuelta)

Como se puede observar, en este caso las conclusiones son similares a las obtenidas en el experimento anterior. El conjunto de datos A ha permitido que la red sea capaz de resolver todos los circuitos menos uno en concreto, el G. El robot ha padecido de los mismos problemas que en el experimento anterior, ya que los desniveles, los brillos especulares y el par motor se mantenían, que pueden ser el causante de que el robot no haya sido capaz de completar una vuelta en ningún sentido en el circuito G. No obstante, se sigue apreciado la diferencia entre el sentido horario y el sentido antihorario debido a los desniveles de las pistas en determinados puntos.

Circuito	Longitud (m)	Tiempo sentido h (s)	Tiempo sentido ah (s)
Training	2.80 m	21.1"	22.9"
A	1.74 m	14.8"	15.5"
B	2.70 m	12.9	-
C	2.20 m	14.3"	-
D	2.60 m	-	-
E	2.60 m	-	15.9"
F	2.35 m	14.1"	14.7"
G	2.90 m	-	-
H	0.80 m	1.0"	1.1"
I	1.80 m	9.1"	9.0"
J	4.00 m	-	-
K	4.00 m	-	-

Tabla 5.6: Tiempos por vuelta en los 12 circuitos utilizando la red Resnet-18* con el conjunto de datos B (los - indican que el modelo no ha conseguido completar una vuelta)

El resultado de este experimento es prometedor, ya que el robot ha sido capaz de completar casi todos los circuitos que se propusieron en primera instancia como objetivo. Como se aprecia, en casi todos los circuitos, el modelo ResNet-18* es más rápido que el modelo MobileNet-v2*. Esto se debe a que el modelo ResNet-18 es más grande que el modelo MobileNet-v2 por lo que las inferencias son más precisas, lo que se traduce en que el robot sigue una trayectoria más eficiente, minimizando los zigzagueos. No obstante, se ha comprobado que este modelo pone al límite las capacidades computacionales de la placa para este problema, ya que las inferencias tenían una mínima latencia, lo que hacía que en ocasiones puntuales el robot se saliera del circuito al no ser capaz de reaccionar a tiempo ante un cambio de dirección.

Los resultados arrojados por este experimento validan la resolución del problema de la conducción autónoma para este trabajo en concreto, ya que se ha conseguido uno de los objetivos principales del proyecto. La red ResNet-18* ha sido capaz de solucionar todos los circuitos propuestos (menos uno), haciendo uso del conjunto de datos A. Se ha conseguido un único modelo que es capaz de completar al menos una vuelta en cada uno de los circuitos propuestos de forma consistente. No obstante, ambos modelos neuronales han conseguido un buen desempeño con el conjunto de datos A, ya que a pesar de que MobileNet no ha conseguido completar todos los circuitos, en algunos como el E, el H y el I, ha sido más rápido que el modelo ResNet18*. En la Tabla 5.7 se pueden observar los resultados comparativos de los dos modelos entrenados con el conjunto de datos A.

		Resnet-18⋆		MobileNet-v2⋆	
Circuito	Longitud (m)	Tiempo sentido h (s)	Tiempo sentido ah (s)	Tiempo sentido h (s)	Tiempo sentido ah (s)
Training⋆	2.80 m	**18.4"**	**21.7"**	19.6"	21.9"
A	1.74 m	**13.9"**	**14.8"**	15.5"	16.7"
B	2.70 m	**14.3"**	**14.9"**	15.9	16.5"
C	2.20 m	**11.4"**	**11.9"**	13.4"	15.7"
D	2.60 m	**14.0"**	**14.5"**	16.2"	17.8"
E	2.60 m	13.7"	**14.0"**	**13.5"**	14.1"
F	2.35 m	**12.4"**	**12.5"**	13.2"	14.0"
G	2.90 m	-	-	-	-
H	0.80 m	1.4"	1.4"	**1.2"**	**1.3"**
I	1.80 m	7.4"	8.2"	**6.5"**	**7.0"**
J	4.00 m	**18.4"**	**19.7"**	-	-
K	4.00 m	**18.5"**	**20.1"**	-	-

Tabla 5.7: Comparativa de mejores resultados de Resnet-18* y MobileNet-v2* (los resultados en negrita indican los mejores tiempos)

6

Conclusiones y trabajos futuros

Como último capítulo para cerrar este Trabajo de Fin de Máster, a continuación se presentan las conclusiones alcanzadas, además de evaluar los resultados obtenidos en función de los hitos marcados al principio del mismo. Por último se dedicará un último punto a mencionar las posibles líneas futuras de investigación que este proyecto abre.

6.1. Conclusiones

Antes de la realización de este proyecto se acordó que el objetivo principal era el desarrollo de un algoritmo de control visual basado en *deep learning* para resolver el problema de la conducción autónoma en robots reales. Como objetivo adicional, se optó por el desarrollo de una plataforma *software* para la ejecución de comportamientos complejos en los robots bautizada BehavioStudio, que fuera compatible con entornos de ejecución simulados y reales.

Tras este trabajo se dispone de esa plataforma estable, flexible y robusta para probar algoritmos de comportamientos complejos en robots, además de dos modelos basados en redes neuronales que solucionan el problema de la conducción autónoma a partir de datos visuales en el robot real JetBot. Estos dos puntos son hitos importantes tanto para el desarrollo personal, como para la organización de *software* libre JdeRobot, ya que ahora dispone de una plataforma para probar algoritmos de este tipo que podrá ser aprovechada por estudiantes de máster y doctorado. El trabajo previo en el que se basa este mismo [9], exploró de forma exhaustiva el comportamiento de diferentes modelos neuronales en conducción autónoma con éxito, pero se limitó al entorno de simulación. El trabajo aquí descrito amplía ese estudio llevando a cabo un estudio similar pero sobre robots reales.

De los dos objetivos principales de este proyecto, el más complejo ha sido el de solucionar el problema de la conducción autónoma, ya que ha requerido un estudio profundo de muchas de las técnicas existentes para controladores visuales, además de incluir la dificultad de hacer que el sistema desarrollado funcionara en un procesador de limitados recursos como es la Jetson Nano. Nos encontramos con infinidad de problemas tanto a nivel de *software* como a nivel de *hardware* para conseguir un modelo robusto que solucionara la conducción autónoma, ya que los entornos simulados ofrecen reproducibilidad exacta de los algoritmos desarrollados, pero en entornos reales hay muchos factores externos que pueden afectar a la hora de hacer pruebas. Como se ha mencionado, se han encontrado problemas diversos en cuanto a capacidad computacional, problemas con la potencia de

los motores, problemas con las pistas utilizadas, iluminación, etc.; esto ha supuesto todo un reto y muchas horas de ensayo-error hasta conseguir los resultados deseados. A nivel de *software*, afortunadamente los problemas han sido menos. Debido al uso de *transfer learning* ya se partía de una base sólida de los modelos neuronales utilizados, aunque igualmente ha habido que ajustarlos finamente. La parte más tediosa de este objetivo fue la grabación y el etiquetado de los conjuntos de datos utilizados, ya que cada imagen se grabó y etiquetó a mano, lo que ha consumido bastante tiempo.

En cuanto al desarrollo del controlador visual basado en *deep learning*, dado que las técnicas más modernas de entrenamiento permiten que los entrenamientos de los modelos no requieran de mucho ajuste (gracias a la técnica de *transfer learning* sobretodo), no ha supuesto demasiada complicación. El mayor reto ha estado en ajustar los parámetros hasta conseguir el resultado deseado. Además, gracias al estudio previo de Vanessa Fernández, se ha facilitado la toma de decisiones en algún frente basándonos en sus resultados.

Por lo que respecta al robot JetBot, la decisión de adquirir un kit de ensamblaje fue acertada, ya que ahorró gran cantidad de trabajo de diseño de piezas y montaje con impresoras 3D involucradas. El kit adquirido es sencillo de montar y dispone de todas las piezas necesarias para tener un robot funcional. No obstante, el mayor obstáculo han sido los motores que vienen de base con el kit. Al tratarse de motores genéricos de juguete, el par motor era muy bajo, lo que impedía al robot moverse a velocidades muy bajas. Este hecho ha impedido realizar un ajuste muy fino del movimiento del robot sobre los circuitos, ya que al aumentar arificialmente la velocidad se perdía control sobre el movimiento, lo que provocaba que las trayectorias del robot fueran erráticas. Además, uno de los motores (el derecho) estaba defectuoso. A igualdad de potencia, el motor derecho ofrecía menos revoluciones, por lo que aplicar una potencia del 60 % a ambos motores al mismo tiempo hacia que el robot tuviera deriva hacia la derecha, al girar el motor derecho más lento que el izquierdo. Este desajuste ha influido negativamente en las soluciones propuestas, ya que ha hecho falta un ajuste de las ganancias de ambos motores para cada experimento realizado, llegando en ocasiones a no poder completar circuitos. No obstante, este problema se soluciona fácilmente adquiriendo motores de más calidad.

Por otro lado, en cuanto al desarrollo de la plataforma *software* BehaviorStudio, lo más costoso fue dar con un diseño que soportara toda la funcionalidad que se quería desde un principio. Se cuidó mucho el diseño de las abstracciones, de tal modo que el *software* resultante tuviera alta cohesión y bajo acoplamiento, lo que hace la plataforma fácil de mantener. Sin embargo, a pesar de haber cuidado la etapa de diseño, surgieron también dificultades durante el desarrollo, que no se habían previsto. Por ejemplo, una de las librerías utilizadas para renderizar componentes en 3D no era compatible con la versión de Python que se utilizó inicialmente para el desarrollo, por lo que hubo que cambiar una gran parte del código para superar esa dificultad. Además, el diseño del GUI (*Graphic User Interface*) fue la parte que más tiempo consumió, ya que, a pesar de tener experiencia previa con la biblioteca Qt, la gran cantidad de elementos necesarios para cubrir toda la funcionalidad requirió de muchas horas de diseño y pruebas.

La parte más compleja del desarrollo *software* fue hacer que los cerebros fueran intercambiables en tiempo de ejecución. Esta es una de las características más importante de BehaviorStudio, ya que permite que la depuración de los algoritmos desarrollados sea instantánea evitando los tiempos de cerrar y volver a abrir la aplicación con las cargas que eso conlleva, sobretodo si se trabaja en entornos simulados. La capacidad que tiene la

plataforma de adaptarse en caliente a cambios en los diferentes cerebros abre la puerta a que los usuarios puedan modificar los cerebros de sus robots casi en tiempo real. Esta es la piedra angular de BehaviorStudio, ya que su cometido principal es facilitar las pruebas de los algoritmos de los usuarios, y poder ver reflejados los cambios rápidamente hace que la depuración de los cerebros sea más sencilla y sobretodo más cómoda para el usuario final.

Otro reto importante fue hacer que BehaviorStudio fuera generalista, ya que a pesar de que está centrada en el problema de la conducción autónoma, siempre se tuvo en mente que a medio o largo plazo diera soporte a diferentes tipos de robots como drones, o incluso humanoides, para diferentes tipos de comportamiento. El encajar todas las piezas *software* supuso un pequeño reto que fue llevado a cabo satisfactoriamente ya que se probó la plataforma con diferentes robots simulados y funcionó. Únicamente falta la validación con robots reales variados. Esta decisión se tomó principalmente para cubrir todo el espectro de robots con los que se trabaja en JdeRobot, para que más usuarios pudieran benficiarse de la plataforma.

Como recapitulación de las novedades aportadas en este trabajo, ahora se dispone de una plataforma *software* completamente funcional para la ejecución de algoritmos basados en redes neuronales que gobiernen diferentes tipos de robots. Además, se ha demostrado que se puede obtener un controlador visual basado en aprendizaje profundo para solucionar el problema de la conducción autónoma en robots reales con toda la problemática que ello conlleva.

Para finalizar esta sección, dejaremos constancia de lo aprendido durante el desarrollo de este trabajo. Este proyecto en concreto implica el conocimiento de multitud de tecnologías muy diferentes entre sí, tanto *software* como *hardware*. Tener una base sólida en el conocimiento de las redes neuronales aplicadas a la visión ha sido clave para la consecución de los objetivos de este proyecto. Además de los conocimientos previos de alguna de las tecnologías utilizadas, ha sido necesario familiarizarse con las librerías de Pytorch, de Qt3D, y estudiar el estado actual del *hardware* embebido más potente para este tipo de proyectos.

6.2. Trabajos futuros

Puesto que no es posible abarcar todas las funcionalidades que ofrece un proyecto de estas características, en este trabajo nos hemos centrado en resolver el problema de la conducción autónoma con un controlador visual basado en *deep learning* sin explorar otras vías muy interesantes. Con los objetivos conseguidos, se abren multitud de posibilidades de desarrollo y líneas de investigación, utilizaremos esta sección para comentarlos.

En cuanto a la plataforma BehaviorStudio, el desarrollo de esta primera iteración ha dejado en el tintero elementos muy interesantes por explorar. Por ejemplo, la inclusión de un módulo evaluador de comportamientos que ofreciera en tiempo real métricas de cómo de bien o de mal lo está haciendo un modelo comparado con otros de referencia, o incluso con modificaciones del mismo. Esto abre la puerta a otros proyectos interesantes como la posibilidad de crear *pipelines* de pruebas en los que puedas programar ejecuciones con diferentes modelos y sea la plataforma la que te devuelva las métricas de calidad de los mismos.

Además se puede incluir la posibilidad de ampliar la gama de sensores y actuadores soportados por la plataforma, obteniendo un centro de mando completo con el que conocer todos los datos sensoriales que ofrece nuestro robot. Se puede ampliar la capacidad de conectarse a diferentes tipos de robots a través del interfaz de usuario, realizar una versión web de la capa de visualización para que la plataforma pueda ser desplegada como servicio web, incluir soporte para otros simuladores más realistas o dedicados, etc.

En lo que respecta al controlador visual, se abren líneas de investigación muy interesantes. En este trabajo nos hemos centrado en resolver el problema como un problema de regresión supervisado utilizando redes neuronales convolucionales. No obstante, se pueden investigar otras vías como incluir las normas de circulación poniendo señales de tráfico en la trayectoria del robot, utilizar varios modelos simultáneamente que se encarguen de diferentes tareas (pilotar, evitar obstáculos, detectar señales de tráfico, etc.), aplicar aprendizaje por refuerzo en lugar de aprendizaje supervisado, incluir temporalidad en la toma de decisiones a través de arquitecturas más complejas como LSTMs o incluso los nuevos Transformers, eliminar las pistas haciendo que el robot navegue por espacios interiores como casas o naves, incluir sensores adicionales como LIDAR para implementar algoritmos de SLAM, y muchas otras.

Por último, y en línea de optimización del proyecto, la conexión entre los robots y la plataforma de ejecución se ha llevado a cabo utilizando la versión *Melodic Morenia* de ROS, pudiendo explorarse la inclusión de ROS2 tanto en BehavioStudio como en el robot real.

Bibliografía

[1] Krizhevsky, Alex and Sutskever, Ilya and Hinton, Geoffrey E. ImageNet Classification with Deep Convolutional Neural Networks *Advances in Neural Information Processing Systems 25*, 2012

[2] Devlin, Jacob and Chang, Ming-Wei and Lee, Kenton and Toutanova, Kristina BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding *arXiv preprint arXiv:1810.04805*, 2018

[3] Dean A. Pomerleau. ALVINN: An autonomous land vehicle in a neural network. *Technical report, Carnegie Mellon University. URL:https://papers.nips.cc/paper/95-alvinn-an-autonomous-land-vehicle-in-a-neural-network.pdf*, 1989.

[4] Yann LeCun, Urs Muller, Jan Ben, Eric Cosatto, Beat Flepp. Off-RoadObstacleAvoidancethroughEnd-to-EndLearning. *URL:http://yann.lecun.com/exdb/publis/pdf/lecun-dave-05.pdf*, 2005.

[5] Mariolis I, Peleka G, Kargakos A, et al. Pose and category recognition of highly deformable objects using deep learning. *Advanced Robotics (ICAR)*, 2015.

[6] Gao Y, Hendricks LA, Kuchenbecker KJ, et al. Deep learning for tactile understanding from visual and haptic data. *arXiv preprint arXiv:1511.06065*, 2015.

[7] Hwang J, Jung M, Madapana N, et al. Achieving "synergyïn cognitive behavior of humanoids via deep learning of dynamic visuo-motor-attentional coordination. *Humanoid Robots (Humanoids)*, 2015

[8] Jain A, Koppula HS, Soh S, et al. Brain4Cars: car that knows before you do via sensory-fusion deep learning architecture. *arXiv preprint arXiv:1601.00740*, 2016.

[9] Vanessa Fernández Martinez Conducción autónoma de un vehículo en simulador mediante aprendizaje extremo a extremo basado en visión *Trabajo de Fin de Máster*, 2019.

[10] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., Polosukhin, I. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[11] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal,Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[12] Mariusz Bojarski, Philip Yeres, Anna Choromanska, Krzysztof Choromanski, Bernhard Firner, Lawrence Jackel, Urs Muller. Explaining How a Deep Neural Network Trained with End-to-End Learning Steers a Car. *arXiv preprint arXiv:1704.07911*, 2017.

[13] Hesham M. Eraqi, Mohamed N. Moustafa, Jens Honer. End-to-End Deep Learning for Steering Autonomous Vehicles Considering Temporal Dependencies. *arXiv preprint arXiv:1710.03804*, 2017.

[14] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., Fei-Fei, L. ImageNet: A large-scale hierarchical image database *CVPR*, 2009.

[15] Keith Sullivan, Wallace Lawson. Reactive Ground Vehicle Control via Deep Networks. *URL:https://pdfs.semanticscholar.org/ec17/ec40bb48ec396c626506b6fe5386a614d1c7.pdf*, 2017.

[16] Lu Chi, Yadong Mu. Deep Steering: Learning End-to-End Driving Model from Spatial and Temporal Visual Cues. *arXiv preprint arXiv:1708.03798*, 2017.

[17] Jinkyu Kim, John Canny. Interpretable Learning for Self-Driving Cars by Visualizing Causal Attention. *arXiv preprint arXiv:1703.10631*, 2017.

[18] Mariusz Bojarski, Anna Choromanska, Krzysztof Choromanski, Bernhard Firner, Larry Jackel, Urs Muller, Karol Zieba. VisualBackProp: efficient visualization of CNNs. *arXiv preprint arXiv:1611.05418*, 2017.

[19] Gustav von Zitzewitz. Survey of neural networks in autonomous driving. *URL:https://www.researchgate.net/publication/324476862_Survey_of_neural_networks_in_autonomo* 2017.

[20] Yunpeng Pan, Ching-An Cheng, Kamil Saigol, Keuntaek Lee, Xinyan Yan, Evangelos A. Theodorou, Byron Boots. Agile Autonomous Driving using End-to-End Deep Imitation Learning. *arXiv preprint arXiv:1709.07174*, 2018.

[21] Jonas Heylen, Seppe Iven, Bert De Brabandere, Jose Oramas M., Luc Van Gool, Tinne Tuytelaars. From Pixels to Actions:Learning to Drive a Car with Deep Neural Networks. In *IEEE Winter Conference on Applications of Computer Vision*, 2018.

[22] Ana I. Maqueda, Antonio Loquercio, Guillermo Gallego, Narciso García, Davide Scaramuzza. Event-based Vision meets Deep Learning on Steering Prediction for Self-driving Cars. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[23] del Egio J., Bergasa L.M., Romera E., Gómez Huélamo C., Araluce J., Barea R. Self-driving a Car in Simulation Through a CNN. *In: Fuentetaja Pizán R., García Olaya Á., Sesmero Lorente M., Iglesias Martínez J., Ledezma Espino A. (eds) Advances in Physical Agents. WAF 2018. Advances in Intelligent Systems and Computing, vol 855. Springer, Cham*, 2019.

[24] Eder Santana, George Hotz. Learning a Driving Simulator. *arXiv preprint arXiv:1608.01230*, 2016.

[25] Udacity. Public driving dataset. *URL:https://eu.udacity.com/course/self-driving-car-engineer-nanodegree–nd013*, 2017.

[26] Udacity's Datasets. Public driving dataset. *URL:https://github.com/udacity/self-driving-car/tree/master/datasets*, 2016.

[27] NuScenes's Datasets. Public driving dataset. *URL: https://www.nuscenes.org/*, 2019.

[28] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio López, Vladlen Koltun. CARLA: An Open Urban Driving Simulator. *arXiv preprint arXiv:1711.03938*, 2017.

[29] Gazebo. Simulator. *URL:http://gazebosim.org/*, 2018.

[30] CARLA. An Open Urban Driving Simulator. *URL:http://carla.org/*, 2017.

[31] Udacity's Self-Driving Car Nanodegree. Self-Driving Simulator. *URL:https://github.com/udacity/self-driving-car-sim*, 2017.

[32] Kitti's Self-Driving Car Nanodegree. Karlsruhe Institute of Technology. *URL:http://www.cvlibs.net/datasets/kitti/*, 2017.

[33] Deepdrive. Self-Driving Simulator. *URL:https://deepdrive.io/*, 2017.

[34] NVIDIA Jetson TK1. Embedded systems hardware *URL:https://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html*, 2014.

[35] NVIDIA Jetson TX1. Embedded systems hardware *URL:https://la.nvidia.com/object/jetson-tx1-dev-kit-la.html*, 2015.

[36] NVIDIA Jetson Nano. Embedded systems hardware *URL:https://www.nvidia.com/es-es/autonomous-machines/embedded-systems/jetson-nano/*, 2019.

[37] NVIDIA Jetson Xavier. Embedded systems hardware *URL:https://www.nvidia.com/es-es/autonomous-machines/embedded-systems/jetson-agx-xavier/*, 2018.

[38] NVIDIA Tegra K1. Embedded systems hardware *URL:https://la.nvidia.com/object/tegra-k1-processor-la.html*, 2013.

[39] Levels of Driving Automation. Automate driving levels of driving automation are defined in new SAE International Standard J3016. *URL:https://www.smmt.co.uk/wp-content/uploads/sites/2/automated_driving.pdf*, 2017.

[40] Follow line. JdeRobot RoboticsAcademy. *https://github.com/JdeRobot/RoboticsAcademy/tree/master/e* 2019.

[41] Google TPU Google. *URL:https://cloud.google.com/tpu/docs/tpus*, 2019.

[42] Francisco Pérez Salgado Caminatas basadas en ondas acopladas para el humanoide nao en Gazebo-5 y JdeRobot *Proyecto final de carrera*, 2015.

[43] Hinton, Geoffrey E., Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep beliefs *Neural compuattion 18.7*, 2006.

[44] Taigman, Yaniv, et al. Deepface: Closing the gap to human-level performance in face verification. *Porceedings of the IEEE conference on computre vision and pattern recognition*, 2014.

[45] He, Kaiming et al. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.