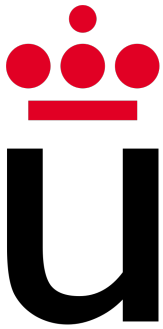


Universidad
Rey Juan Carlos

Escuela de Ingeniería de Fuenlabrada

TRABAJO FIN DE GRADO

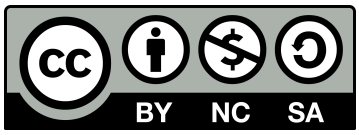
Programación de drones con aprendizaje por imitación y redes neuronales

Grado en Ingeniería Robótica de Software

Realizado por
Adrián Madinabeitia Portanova

Dirigido por
José María Cañas

Curso académico 2023/2024



Agradecimientos

Me gustaría dedicar una página a todas aquellas personas que me han ayudado llegar hasta aquí, principalmente mis padres y mi hermana, sin ellos no sería nada.

También quiero agradecérselo a todos los compañeros de universidad con los que he estado a lo largo de la carrera. No solo se convirtieron en amigos, si no también en una parte fundamental de mi. La carrera no solo mereció la pena por lo que aprendí si no por la gente que conocí.

Gracias también a los profesores que me brindaron sus conocimientos sobre robótica y así poder haber finalizado el grado con profundos conocimientos en este área.

Y por último pero no menos importante gracias al equipo de JDE Robots y Aerostack2 que me ayudaron en el desarrollo del trabajo y estuvieron dispuestos a ayudar a todo lo que estuviera en su mano.

Gracias a todos por tanto.

Quisiera dedicaros unos versos
que valieran lo que vale vuestra luz,
pero no cabe en un folio el universo,
tampoco en una frase cabe mi gratitud.
100 frases, Sharif

Resumen

En los últimos años, la robótica aérea y la inteligencia artificial han tenido un impulso enorme y empiezan a tener un impacto importante en nuestra sociedad. Ejemplos de ello son la comercialización y diversos usos de los drones, así como las diferentes inteligencias artificiales y modelos de redes neuronales que se están desarrollando hoy en día. No obstante, la inteligencia artificial aún tiene varios campos donde su desarrollo no está tan explorado, como es el caso de los drones.

Habitualmente, se utilizan algoritmos de programación clásica para el control de drones. El control de los mismos suele hacerse con cámaras y diversos sensores que requieren un procesamiento intensivo. Esto provoca que el dron necesite como carga útil una unidad de procesamiento para poder tratar todos los datos, o en su defecto, procesarlos en la estación de tierra, lo que añade un retraso en las comunicaciones y posibles errores que podrían afectar a un sistema que es crítico manejar en tiempo real.

Este trabajo consiste en la aplicación de redes neuronales para programar drones y así comprobar su rendimiento en este tipo de sistemas. Se implementará en dos aplicaciones: inicialmente, una aplicación sigue líneas, donde el drone deberá seguir una línea en varios circuitos con el menor error posible; y paralelamente, deberá cruzar ventanas, que consistirá en un control mixto donde el dron será teleoperado, pero si el piloto lo desea, la red neuronal tomará el control del dron y tratará de cruzar las puertas que se presenten en su ángulo de visión.

Gracias a estas dos aplicaciones, podremos cuantificar la mejora que pueden ofrecer las redes neuronales tanto en rendimiento computacional como en la propia aplicación, probando también diferentes arquitecturas neuronales para un análisis completo.

Palabras clave: robótica, drones, inteligencia artificial, redes neuronales, control de drones, procesamiento de datos, teleoperación.

Acrónimos

TFG - Trabajo de Fin de Grado
TFM - Trabajo de fin de Master
ROS 2 - Robot Operating System 2
OpenCV - Open Source Computer Vision
NVDI - Normalized Difference Vegetatio Index
API - Application programming interface
GPU - Graphic Proceessing Unit
CPU - Central Processing Unit
GNU GPL - General Public License
UAS - Unmanned Aerial System
UA - Unmanned Aircraft
IA - Inteligencia artificial
MAV - Micro Air Vehicle
CCN - Convolutional Neural Network
SGD - Stochastic gradient descent
ODE - Open Dynamics Engine
MSE - Mean Square Error
XML - Extensible Markup Language
LIDAR - Light Detection and Ranging
GPS - Global Positioning System

Índice general

Índice de figuras

Índice de tablas

1. Introducción

En este capítulo situaremos el contexto de este trabajo. Se explicará el origen etimológico y literario de la palabra *Robot*, así como los avances en la inteligencia artificial y el *machine learning*. Posteriormente, se expondrá lo que es un robot desde una perspectiva técnica, su clasificación y la posición de los drones en este contexto.

1.1. Robótica

La palabra *robot* deriva de la palabra checa *robota*, que significa trabajador o sirviente. Se popularizó a partir de la obra *R.U.R.* (Rossum's Universal Robots) escrita por Karel Čapek en 1920. En esta obra se plantea la construcción de robots para liberar a las personas de la pesada carga del trabajo.

El término *robótica* fue acuñado por Isaac Asimov, quien definió la robótica como la ciencia que estudia a los robots. En su libro *Yo, Robot* publicado en 1950, Asimov postuló las Tres Leyes de la Robótica, las cuales han tenido un impacto significativo en la forma en que pensamos sobre la interacción entre humanos y robots. Además de las Tres Leyes, Asimov introdujo la Ley Cero en obras posteriores, ampliando el marco ético y moral para los robots:

1. **Ley 0:** Un robot no debe actuar simplemente para satisfacer intereses individuales, sino que sus acciones deben preservar el beneficio común de la humanidad.
2. **Ley 1:** Un robot no puede herir a un ser humano ni dejar de intervenir si este está en peligro de sufrir algún daño.
3. **Ley 2:** Un robot debe obedecer las órdenes de los humanos, salvo si estas contradicen la primera ley.
4. **Ley 3:** Un robot debe proteger su propia existencia, siempre y cuando dicha protección no contradiga las leyes primera y segunda.

Asimov también exploró la psicología y sociología de las sociedades robotizadas, proponiendo escenarios en los que robots y humanos coexisten y

colaboran.

Un robot se compone principalmente de cuatro elementos: sensores, actuadores, una unidad computacional y el software. Los sensores recopilan información del entorno, los actuadores interactúan con ese entorno, la unidad computacional procesa la información recibida y generan señales para los actuadores. Sin embargo, es el software el componente crucial que dota de autonomía a los robots, ya que procesa la información sensorial y la traduce en acciones, permitiendo que el robot pueda entender e interactuar con su entorno. Los robots están diseñados para ayudar y mejorar la vida de las personas, y las tareas que suelen realizar se conocen como las 4D:

- ***The dirty*** son aquellos que involucran tareas con altos niveles de suciedad, residuos o materiales nocivos que pueden ser desagradables o peligrosos para los seres humanos. La automatización de estos trabajos con robots ayuda a minimizar la exposición humana a entornos insalubres y potencialmente dañinos. Muestra de ello es el siguiente brazo robótico 1.1 cuya finalidad es la recolección de basura en una planta de reciclaje.

Figura 1.1: Brazo robótica en planta de reciclaje

- ***The dangerous:*** Son aquellos robots que presentan altos riesgos para la seguridad y la vida de los trabajadores humanos. El uso de robots en estos entornos permite realizar tareas que podrían ser mortales o causar lesiones graves, en la imagen 1.2 se muestran varios modelos de robots usados en expediciones de zonas afectadas por radicación nuclear.

Figura 1.2: Robot de limpieza de restos nucleares

- ***The dull*** consiste en trabajos monótonos y repetitivos son aquellos que pueden llevar a la fatiga y desmotivación en los trabajadores humanos. Los robots pueden realizar estas tareas con gran eficiencia y sin perder precisión, liberando a los humanos para actividades más creativas y de mayor valor. Un ejemplo de esto puede ser la tarea de organizar un almacén como se puede ver en la imagen 1.3 donde un sistema compuesto de varios robots permite organizar el almacén de forma autónoma.

Figura 1.3: Robots en almacenes

- ***The delicate*** son trabajos delicados que requieren precisión extrema y un alto grado de control que puede ser difícil de mantener consistentemente por los seres humanos. Los robots pueden operar con la exactitud necesaria para estas tareas, mejorando los resultados y reduciendo el margen de error, un ejemplo de ello es en la cirugía 1.4, con un cirujano especializado en este tipo de robots,

se puede obtener una colaboración humano-robot mucho mas precisa y menos invasiva que una operación convencional.

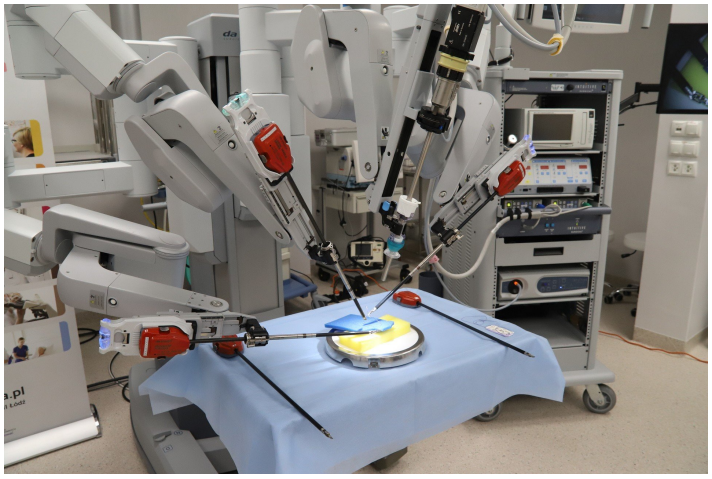


Figura 1.4: Robot en cirugía

Existen diversas formas de clasificar robots, se pueden clasificar por su inteligencia, separación entre robots industriales y robots de servicio o por tarea. De manera general los robots se pueden dividir en:

- **Poliarticulados:** Son artilugios mecánicos y electrónicos destinados a realizar de forma automática determinados procesos de fabricación o manipulación. Suelen ser fijos, aunque también pueden realizar desplazamientos limitados y poseen un espacio de trabajo concreto y limitado. Los mejores ejemplos son los robots industriales 1.5, manipuladores o cartesianos. Estos robots se utilizan ampliamente en la fabricación de automóviles, electrónica y otras industrias donde se requieren operaciones precisas y repetitivas.

Figura 1.5: Robots poliarticulados

- **Móviles**: Están provistos de algún tipo de mecanismo que les permite desplazarse autónomamente, como patas o ruedas 1.6, y reciben información de su entorno a través de sus propios sensores. Son ampliamente utilizados en el transporte de mercancías o en la exploración de lugares de difícil acceso. Pueden ser terrestres, acuáticos, aéreos o espaciales. Ejemplos notables incluyen los robots exploradores como el Rover de Marte y los drones de entrega.

Figura 1.6: Summit: Robot móvil

- **Zoomórficos:** Son aquellos que tratan de reproducir en mayor o menor grado de realismo los sistemas de locomoción de diversos seres vivos. Un ejemplo destacado hoy en día es Spot 1.7, desarrollado por Boston Dynamics[1]. Spot es un robot cuadrúpedo diseñado para imitar la movilidad de los animales, permitiéndole moverse con agilidad en terrenos difíciles y variados. Es utilizado en una amplia gama de aplicaciones, desde inspecciones industriales y mapeo hasta operaciones de búsqueda y rescate.

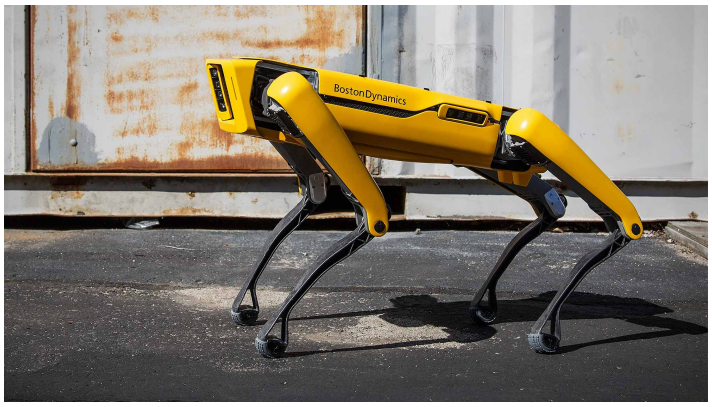


Figura 1.7: Spot: Robot zoomórficos

[1] https://bostondynamics.com/

- **Androides:** Intentan reproducir total o parcialmente la forma y el comportamiento del ser humano. No solo imitan la apariencia humana, sino que emulan también la conducta de forma autónoma. Los androides tienen aplicaciones en entretenimiento, asistencia a personas mayores y discapacitadas, y en estudios de interacción humano-robot. Un ejemplo destacado de estos es el robot Sophia 1.8, desarrollada por Hanson Robotics[2]. Sophia es conocida por su avanzada capacidad de interacción verbal y facial, lo que le permite participar en conversaciones naturales con humanos.

Figura 1.8: Sophia: Robot androide

1.1.1. Robótica área

La robótica aérea se encarga del estudio del comportamiento autónomo de aeronaves no tripuladas, conocidas como UAV (Vehículos Aéreos No Tripulados) o UAS (Sistemas de Aeronaves No Tripuladas). Estos sistemas pueden realizar misiones sin necesidad de tener una tripulación embarcada y su uso es cada vez más extendido. Los drones son un ejemplo común de este tipo de sistemas. Con el avance de la tecnología, los drones están siendo equipados con capacidades avanzadas de navegación autónoma, inteligencia artificial y sensores de alta precisión, sus principales áreas de uso son las siguientes:

[2] https://www.hansonrobotics.com/

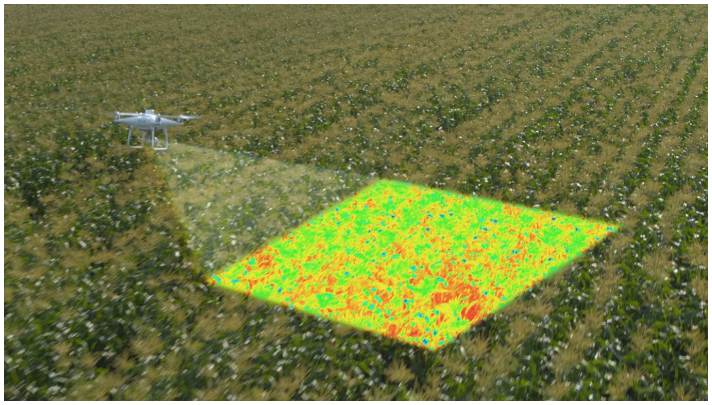


Figura 1.9: DJI P4 multiespectral

La agricultura es uno de los sectores mas importantes hoy en día. Debido al crecimiento de la población, cada vez se requiere mas alimento. Para una correcto análisis y cuidado de los cultivos, los drones son una herramienta muy útil y versátil. Principalmente hay 2 funciones que se están comercializando hoy en día y que surgieren un avance realmente importante en el cuidado de cultivos:

- **Monitorización:** Al tener una imagen aérea, los drones pueden realizar un análisis exhaustivo del estado del cultivo. Con un coste reducido se puede utilizar un dron con cámaras térmicas (NIR) para obtener el NVDI *(Normalized Difference Vegetatio Index)*. El NVDI muestra las zonas donde el cultivo está mas sano, permitiendo conocer al agricultor que secciones requieren fertilizantes o riego. Además también ofrece información respecto a la densidad real del cultivo. Un ejemplo de dron aplicado a la monitorización de cultivos es el dron *DJI P4 multiespectral*[3], el cual también aparece ilustrado en la figura 1.9.
- **Control de malas hierbas:** Otro problema crítico en la agricultura son las malas hierbas y plagas. Los drones permiten fumigar el campo de manera efectiva, usando la cantidad justa de fungicida. Esto permite que el dron contribuya a usar el mínimo fungicida posible y bajar la contaminación producida por los mismos. Otra ventaja a destacar es que al tener al operario alejado de la zona de acción, este mismo evitará aspirar el fungicida. Lo que es un beneficio directo a la salud del propio agricultor.

[3] https://dronity.com/drones/drone-dji-p4-multispectral/

Inspección y mantenimiento

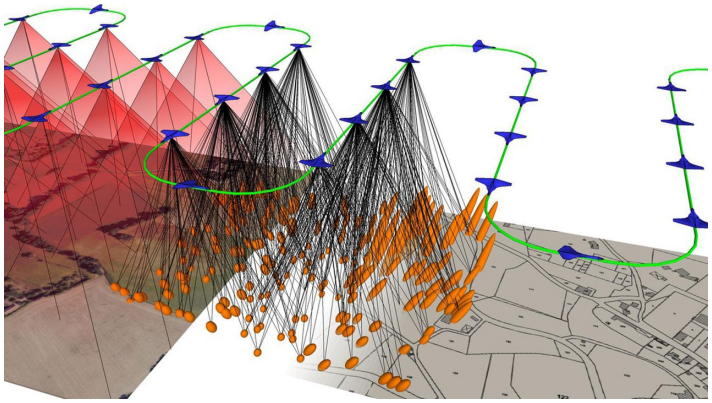


Figura 1.10: Cartografía autónoma

Hay inspecciones que pueden ser peligrosas para un humano, sin embargo un dron permite realizar estas expediciones sin riesgos mayores. Estas misiones pueden ser tanto teleoperadas, en caso de situaciones excepcionales, o con control autónomo en caso de inspecciones rutinarias, ayudando a los operarios centrarse en otras tareas más importantes.

La cartografía con drones es una técnica que cada día se está ampliando más y consiste en la construcción de un mapa, ya sea en 2 o 3 dimensiones del entorno. Al tener una vista aérea, los drones pueden realizar un modelo completo de superficies extensas. Además debido al desarrollo de algoritmos en este área, el dron es capaz de cartografiar una superficie sin necesidad de la atención constante de un operario. En la imagen 1.10 se puede ver el posible recorrido que puede hacer un dron para la inspección de una área extensa.

Otro uso interesante de los drones es el uso de cámaras en el espectro *MWIR*. Este espectro de frecuencia permite la detección de gases, permitiendo anticipar males mayores y fugas en maquinaria. Ambas características sirven especialmente tanto en situaciones rutinarias como inspecciones de cables de alta tensión o maquinaria de minería, como en situaciones especialmente peligrosas como serían inspecciones de zonas contaminadas por radiación o misiones militares.

Drones de rescate

Figura 1.11: Dron de rescate en nieve

La robótica área está generando un especial impacto en las misiones de rescate, debido a las características de los drones, estos permiten explorar y localizar a víctimas tanto de inundaciones, terremotos o accidentes. En entornos que pueden ser peligrosos para las personas o presentar dificultades para el desplazamiento, estas características hacen que los drones estén siendo una herramienta muy útil para los equipos de rescate. En la imagen 1.11 podemos ver un ejemplo de un dron de rescate en entornos nevados. Gracias al dron se puede localizar a una persona en una situación crítica permitiendo a los equipos de rescate desplazarse al punto exacto donde se encuentra la persona en apuros. Esto supone un riesgo menor tanto personal como económico, además de una mayor eficiencia.

1.2. Inteligencia Artificial

La Inteligencia Artificial (IA) se refiere a la capacidad de una máquina para imitar las funciones cognitivas que los humanos asocian con otras mentes humanas, como aprender y resolver problemas. Este campo interdisciplinario combina conocimientos de informática, matemáticas, ingeniería, psicología, lingüística y otras ciencias para crear sistemas que puedan realizar tareas que, hasta hace poco, requerían de la inteligencia humana.

En 1936 se postula que el razonamiento se puede sistematizar de una forma lógica y, teniendo en cuenta que una máquina puede realizar cualquier cálculo que haya sido formalmente definido, se plantea una cuestión teórica en la que se discute si las máquinas realmente pueden pensar. Esta idea se refleja en el libro *Computing Machinery and Intelligence*, publicado en 1950, donde se introduce el Test

de Turing como un criterio para determinar si una máquina exhibe comportamiento inteligente indistinguible del de un humano.

Años después, en la Conferencia de Dartmouth en 1956, se usaría el término de inteligencia artificial formalmente. A esta conferencia asistieron científicos destacados en el campo como John McCarthy, Marvin Minsky, Nathaniel Rochester y Claude Shannon. Un concepto interesante que surgió al año siguiente fue el de singularidad tecnológica, que marca el punto donde las máquinas serían capaces de pensar por sí solas, y al no tener las limitaciones humanas, superarían a los humanos con gran velocidad. Este concepto formó las bases del transhumanismo.

La IA se puede clasificar de varias maneras, dependiendo de su capacidad y funcionalidad:

- **IA débil**: Diseñada y entrenada para realizar tareas específicas, como reconocimiento de voz, clasificación de imágenes o recomendaciones personalizadas. Ejemplos incluyen asistentes virtuales como Siri, Alexa y Google Assistant.
- **IA fuerte**: Se refiere a sistemas con capacidades cognitivas generales similares a las de los seres humanos. Estos sistemas no solo pueden realizar tareas específicas, sino que también pueden aprender y adaptarse a una amplia variedad de actividades. Esta forma de IA aún no se ha logrado.
- **Superinteligencia artificial**: Es una forma hipotética de IA que supera la inteligencia humana en todos los aspectos, desde la creatividad hasta la resolución de problemas y la toma de decisiones.

Dentro de lo que conocemos hoy como inteligencia artificial, se utilizan varias técnicas para que las máquinas tengan estos comportamientos inteligentes, como se puede ver en la figura 1.12. Aunque en este trabajo se trabaje en 3 ramas; robótica, visión y *machine learning*. Nos centraremos en la explicación del *machine learning*.

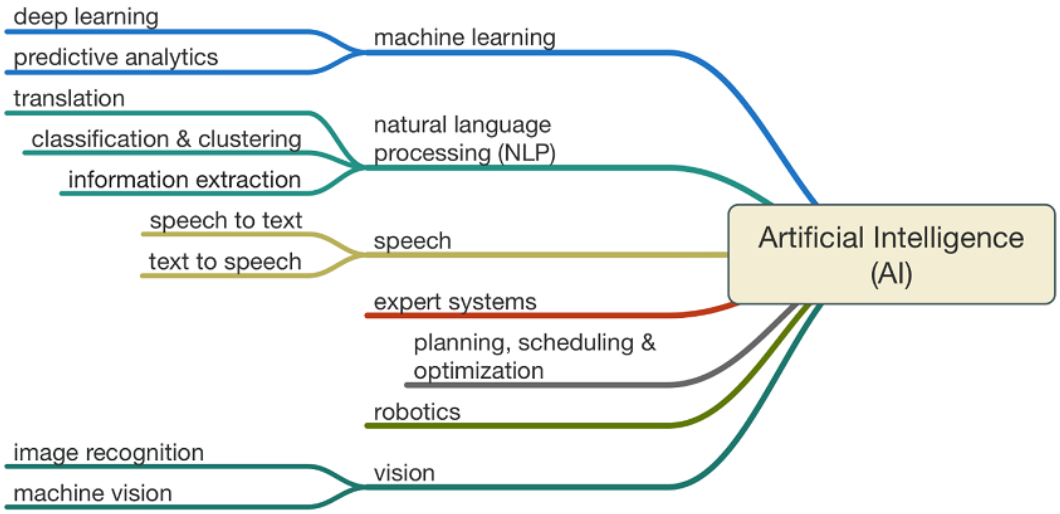


Figura 1.12: Ramificaciones de la IA

El *Machine learning* es una subdisciplina de la inteligencia artificial que se centra en el desarrollo de algoritmos y modelos que permiten aprender y hacer predicciones o tomar decisiones basadas en datos. En lugar de seguir instrucciones programadas de forma explícita, los sistemas de *machine learning* identifican patrones y regularidades en los datos para mejorar su rendimiento en una tarea específica a través de la experiencia. Esta subdisciplina se divide en 3 ramas:

- **Aprendizaje supervisado:** Es un tipo de *machine learning* en el cual el modelo se entrena utilizando un conjunto de datos etiquetados, es decir, datos que incluyen tanto las entradas como las salidas deseadas. El objetivo es aprender una función que mapee las entradas a las salidas correctas. El desarrollo de este trabajo se realizará en esta rama del *machine learning*. El aprendizaje extremo a extremo consiste en una red neuronal donde a partir de una entrada, en nuestro caso una imagen, la red neuronal será capaz de inferir una salida (las velocidades del dron). Para conseguir esto entrenaremos la red con los fotogramas obtenidos a partir de un piloto experto y las velocidades comandadas al dron asociadas a cada fotograma.
- **Aprendizaje no supervisado:** En este enfoque, el modelo se entrena utilizando un conjunto de datos que no están etiquetados, lo que significa que solo se proporcionan las entradas sin las salidas correspondientes. El objetivo es encontrar patrones o estructuras ocultas en los datos, como agrupaciones *(clustering)* o reducción de dimensionalidad.

- **Aprendizaje por refuerzo:** Es un tipo de machine learning en el cual un agente aprende a tomar decisiones mediante la interacción con un entorno dinámico. El agente recibe recompensas o penalizaciones basadas en sus acciones y ajusta su comportamiento para maximizar la recompensa acumulada a lo largo del tiempo.

2. Objetivos y plan de trabajo

Una vez presentado el contexto, se expondrán los objetivos propuestos en el trabajo y el procedimiento seguido para completar cada objetivo y subobjetivo especificado, además de las estrategias y puntos claves para la realización del trabajo.

2.1. Objetivos

El objetivo principal de este trabajo es demostrar que un piloto de dron controlado por una red neuronal, usando como entrada las imágenes de la cámara ventral de abordo, puede llegar a producir iguales o mejores resultados que un piloto humano o programado de manera clásica en dos aplicaciones usando *imitation learning*. Para desarrollar este trabajo, a parte del objetivo principal, se establecieron una serie de subobjetivos para el desarrollo del trabajo.

El objetivo principal consistirá en demostrar que un sistema de *imitation learning* es capaz de aprender y pilotar adecuadamente un dron. Dividiremos este objetivo en 2 subobjetivos:

- Realizar una aplicación sigue líneas donde el dron tenga como entrada la imagen de la cámara frontal. Para la creación del *dataset* se programará un piloto experto.
- Del mismo modo que la aplicación del sigue líneas, se hará otra aplicación donde el dron tenga que cruzar ventanas, complementando con *deepPilot* la componente de altura. Como entrada a ambas redes neuronales, se utilizará también la imagen frontal de la cámara del dron. Sin embargo, la obtención del *dataset* será a partir de un piloto manual.

2.2. Metodología

Este proyecto se ha abordado combinando la metodología ágil de Kanban con el desarrollo en espiral para un control eficiente de tareas generales y subtareas. El

modelo en espiral se caracteriza por la realización de subtareas que dividen la tarea principal en un número determinado de ciclos que serán marcados por cada objetivo mencionado anteriormente. En cada ciclo existen cuatro etapas:

1. **Análisis de requisitos:** En esta etapa se incluyen las reuniones semanales con el tutor, donde se discuten los avances realizados cada semana y se planifica lo que se hará en la siguiente iteración o incremento.
2. **Diseño e implementación:** Una vez definido lo que se realizará durante la semana, se lleva a cabo un estudio teórico o búsqueda de herramientas necesarias para la tarea semanal. Posteriormente, se procede a implementar según lo acordado en la reunión.
3. **Pruebas:** Durante este periodo se prueba el funcionamiento robusto del código. Si las pruebas son satisfactorias, se utiliza el repositorio de GitHub como herramienta para la gestión de código y control de versiones.
4. **Planificación de la siguiente iteración:** Esta etapa incluye la redacción del blog[1] semanal o bitácora, donde se detallan todos los pasos y el trabajo realizado cada semana.

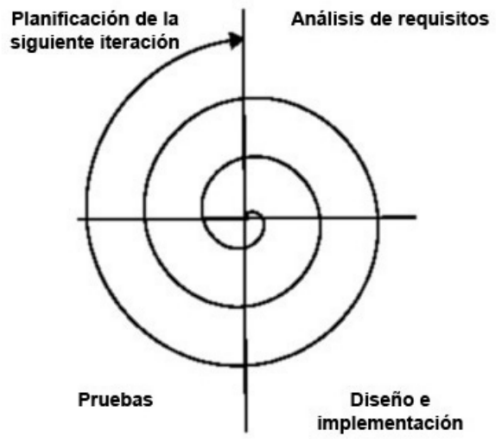


Figura 2.1: Modelo en espiral

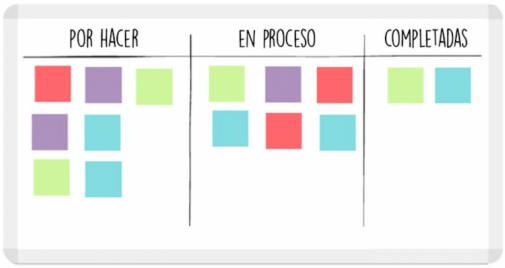


Figura 2.2: Pizarra de Kanban

Respecto a la metodología de Kanban, se trata de un método visual de gestión de proyectos que permite organizar las tareas pendientes de manera visual en columnas. Cada columna representa una etapa de trabajo, en nuestro caso, "Trabajo Pendiente", "En proceso" y "Terminado".

[1] https://roboticslaburjc.github.io/2023-tfg-adrian-madinabeitia/

Aunque esta técnica suele emplearse en equipos, también permite una adecuada organización personal al visualizar de manera gráfica las tareas. Debido a la familiaridad con proyectos anteriores, se utilizó la herramienta virtual Trello[2] para organizar el trabajo semanal. Este trabajo semanal se irá especificando en reuniones semanales con el tutor, donde se expondrán los avances de cada semana, los objetivos conseguidos y el plan para la siguiente semana o fase. Estas reuniones permitirán ajustar la planificación según las necesidades del proyecto y asegurar que se mantenga un progreso constante y alineado con los objetivos establecidos.

Todo el desarrollo del proyecto es *open source* y accesible en el siguiente repositorio de GitHub[3].

2.2.1. Estandarización

En proyectos de *machine learning*, la estandarización del repositorio es crucial para facilitar su reutilización y mantenimiento. Este tipo de proyectos se suele organizar de la siguiente manera:

- **data/**: Para almacenar conjuntos de datos y archivos relacionados.
- **models/**: Para guardar los modelos entrenados.
- **notebooks/**: Para notebooks de Jupyter utilizados en la exploración de datos y experimentación.
- **src/**: Para el código fuente del proyecto, incluyendo scripts de entrenamiento, evaluación y utilidades.
- **docs/**: Para la documentación del proyecto, como instrucciones de instalación y uso.

Siguiendo estas prácticas de estandarización[4] junto al uso de gitHub[5], se puede mejorar la organización, reproducibilidad y mantenibilidad de los proyectos de *machine learning*, facilitando su uso y colaboración por parte de otros desarrolladores.

[2] https://trello.com/
[3] https://github.com/RoboticsLabURJC/2023-tfg-adrian-madinabeitia
[4] https://medium.com/analytics-vidhya/folder-structure-for-machine-learning-projects-a7e451a8caaa
[5] https://github.com/

2.3. Plan de trabajo

Para ambas aplicaciones se seguirán los siguientes puntos principales, permitiendo desglosar la aplicación en problemas más pequeños, siguiendo la filosofía del pensamiento computacional:

1. Estudio del estado del arte en *imitation learning* aplicado a drones para obtener una visión panorámica del problema y ayudar a tomar decisiones y avanzar en el proyecto.
2. Preparación de *drivers* en simulación.
3. Diseño de un piloto experto o teleoperado con el fin de recoger una gran cantidad de muestras y así generar el *dataset* para la red neuronal.
4. Automatización de la generación del *dataset*
5. Análisis del *dataset* e investigación en técnicas de aumento de datos y tratamiento de *dataset*.
6. Entrenamiento de la red neuronal.
7. Implementación del piloto neuronal.
8. Evaluación de los resultados.

3. Marco teórico

En este capítulo se explicarán, a nivel teórico, aspectos básicos de las redes neuronales, enfocándonos en aquellos que conciernen al *machine learning* e *imitation learning*. Además, se describirán los elementos básicos de un dron y su modelo.

3.1. Redes Neuronales

Las redes neuronales son un modelo computacional que se inspira en el cerebro humano y trata de replicar la forma en la que las neuronas biológicas se conectan entre sí. Una red neuronal está compuesta de un conjunto de neuronas y cada neurona se conecta a otra teniendo un peso y umbral asociados. En el marco del *machine learning* y del *imitation learning*, se utilizan diversas arquitecturas de redes neuronales, entre las que se encuentran:

3.1.1. Redes Neuronales Convolucionales

Las Redes Neuronales Convolucionales (CNN) tienen un papel significativo en el ámbito del *imitation learning*, especialmente debido a su capacidad para la clasificación y extracción de información en imágenes. Con las cámaras siendo sensores económicos que proporcionan una gran cantidad de datos visuales, las CNN se convierten en una herramienta esencial.

Este tipo de redes neuronales recibe típicamente una entrada de 3 dimensiones, *w x h x c* donde w define el número de píxeles que tiene la imagen de ancho, h el número de píxeles que tiene de altura y c el número de canales de la imagen. Su arquitectura se basa en la corteza visual del cerebro humano donde se aplican diferentes tipos de capas de neuronas para extraer las características necesarias de la imagen:

- **Capa convolucional:** Esta capa recibe una imagen como entrada y aplica un filtro o *kernel* para producir un mapa de características, lo que reduce la cantidad de parámetros. Los hiperparámetros importantes incluyen el número de filtros (que afecta la profundidad de la salida), el *stride* (la

distancia que se mueve el filtro), y el *zero-padding* (para manejar los bordes de la imagen). Después de la operación de convolución, se aplica una función de activación, siendo ReLU la más común.

- **Capa completamente conectada o *fully connected*:** En este tipo de capa, se conecta cada neurona de la capa de entrada con cada neurona de la capa de salida. La función de esta capa es realizar la tarea de clasificación basándose en las características obtenidas de las capas anteriores a la misma.
- **Capa LSTM:** Puede agregar o eliminar información a través de estructuras llamadas puertas, que actúan como mecanismos de control para regular el flujo de información, permitiendo que ciertas señales pasen mientras bloquean otras. Su característica mas importante es que son capaces de almacenar estados y dependencias temporales, actuando como la memoria del sistema.
- **Capa de *pooling*:** Se utilizan para reducir las dimensiones espaciales. Opera mediante una ventana deslizante sobre el volumen de entrada. Las clases de submuestreo más comunes son:
  - ***Max Pooling:*** Conserva los valores máximos de un grupo de neuronas.
  - ***Average Pooling:*** Calcula el promedio de los píxeles dentro de una ventana para cada píxel del volumen de salida.

3.1.2. Redes Neuronales Recurrentes (RNN)

Las Redes Neuronales Recurrentes (RNN) son adecuadas para modelar datos secuenciales. La idea principal detrás de las RNN es utilizar información secuencial en lugar de información independiente. Este tipo de red neuronal aprende a emplear información pasada en casos donde esa información es relevante, funcionando a modo de memoria a corto plazo. Esto se debe a que la salida de este tipo de redes, retorna como entrada. Gracias a esta retroalimentación, la red es capaz de tener en cuenta su estado anterior y esto les permite capturar dependencias temporales en los datos y generar predicciones o clasificaciones más precisas. Las RNN puede ser especialmente útiles para tener en cuenta posiciones anteriores del dron y actuar en su consecuencia.

3.1.3. Redes de regresión

Las Redes de Regresión se utilizan para predecir el valor real, gradual y continuo de una variable numérica en función de los valores de una o varias variables independientes. A diferencia de las redes de clasificación, las redes de regresión están diseñadas para estimar relaciones continuas entre variables.

3.1.4. Entrenamiento

El entrenamiento de una red neuronal es un proceso crucial en el que se ajustan los pesos de la red para que pueda aprender a realizar una tarea específica. Durante el entrenamiento, la red neuronal se expone a un conjunto de datos de entrada junto con las salidas supervisadas. El objetivo es minimizar la discrepancia entre las salidas predichas por la red y las salidas esperadas.

El algoritmo encargado de ajustar los pesos de cada neurona es conocido como retropropagación. La retropropagación consiste en calcular el gradiente del error con respecto a los pesos de la red y ajustar estos pesos para minimizar el error. Durante la retropropagación, el error se propaga hacia atrás desde la salida de la red hasta las capas anteriores, permitiendo actualizar los pesos en función de su contribución al error total. Esto se realiza utilizando la derivada parcial del error con respecto a cada peso, lo que permite un ajuste preciso de los mismos.

Durante el entrenamiento tenemos que tener en cuenta los siguientes conceptos:

- ***Learning Rate:*** Determina la magnitud de los ajustes de peso en cada iteración del algoritmo de optimización. Una tasa de aprendizaje más alta puede acelerar el entrenamiento, pero también puede causar oscilaciones o incluso divergencia en el proceso de optimización.
- ***Momentum:*** Es un parámetro que ayuda a acelerar la convergencia y suaviza el proceso de optimización. Permite que el optimizador acumule velocidad en la dirección del gradiente y reduce la cantidad de oscilaciones durante el entrenamiento.
- ***Batch Size:*** Es el número de ejemplos de entrenamiento que se utilizan en una iteración del algoritmo de optimización. El uso de lotes más grandes puede acelerar el entrenamiento al permitir el procesamiento paralelo de ejemplos, pero también puede requerir más memoria.
- ***Función de pérdida:*** Es una medida que cuantifica la discrepancia entre las

predicciones de la red y los objetivos reales en los datos de entrenamiento. Durante el entrenamiento, el objetivo es minimizar esta función de pérdida para mejorar la precisión de la red.

- ***Epoch:*** Es un ciclo completo a través de todo el conjunto de datos de entrenamiento. En cada época, la red procesa y actualiza sus parámetros utilizando cada ejemplo al menos una vez. El entrenamiento puede constar de múltiples épocas para permitir que la red aprenda de manera más completa los patrones en los datos.
- ***Overfitting:*** Ocurre cuando la red se ajusta demasiado a los datos de entrenamiento y no generaliza bien a nuevos datos. Para evitar el sobreajuste, se pueden utilizar técnicas como la regularización y la validación cruzada durante el entrenamiento.
- **Criterio de finalización de entrenamiento:** Determina el momento o situación que marca el fin del entrenamiento. Hay diversos criterios para finalizar el entrenamiento de una red neuronal, aunque en nuestro caso será por un error de predicción menor a cierto umbral, hay otras técnicas como máximo número de iteraciones y estabilización del error en los datos de entrenamiento.

3.1.5. PilotNet

Podemos encontrar referencias del uso y aplicación de esta red en [1], donde PilotNet se utiliza para el control autónomo de vehículos. En este trabajo, se esta red para predecir la dirección de conducción a partir de imágenes de la cámara del vehículo, lo que permite a un coche autónomo tomar decisiones en tiempo real sobre el rumbo a seguir. Además en otro trabajo de Bhattacharyya [2], se revisa el uso de esta red para la conducción de vehículos autónomos.

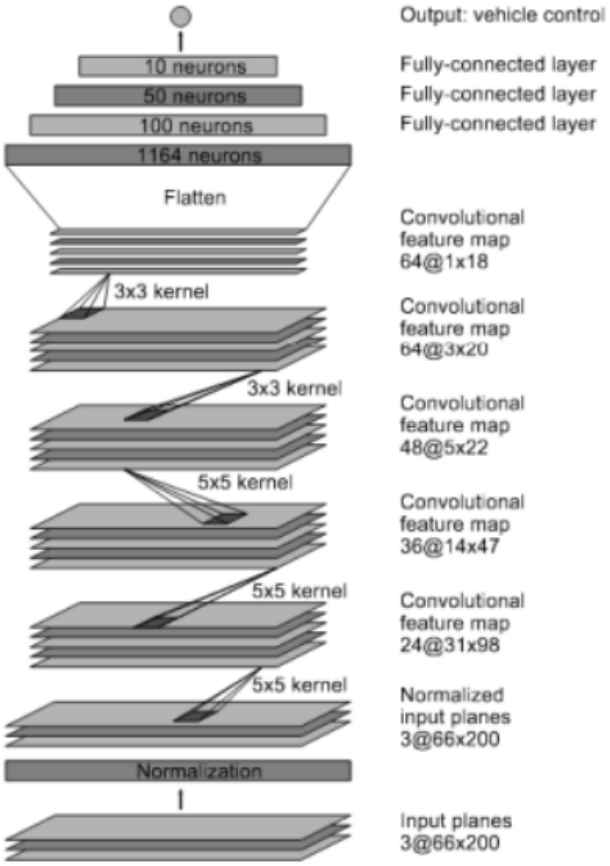


Figura 3.1: Infraestructura PilotNet

PilotNet consta de 9 capas, que incluyen una capa de normalización, 5 capas convolucionales y 3 capas *fully-connected* o densas. Las 2 primeras capas convolucionales usan un *stride* de 2x2 y un kernel de 5x5, mientras que las 3 últimas no tienen *stride* y poseen un kernel de 3x3.

Las 3 capas *fully-connected* fueron diseñadas para funcionar como un controlador de dirección, pero no es posible saber exactamente qué partes actúan como tal. Las activaciones de los mapas de nivel superior se convierten en máscaras para las activaciones de niveles inferiores utilizando el siguiente algoritmo:

1. En cada capa, las activaciones de los mapas de características se promedian.
2. El mapa con el promedio más alto se escala según el tamaño del mapa de la capa de abajo.
3. El mapa promediado aumentado de un nivel superior se multiplica después con el mapa promediado de la capa de abajo, lo que resulta en una máscara de

tamaño intermedio.

4. La máscara intermedia se escala al tamaño de los mapas de la capa inferior de la misma manera que en el paso 2.
5. El mapa intermedio mejorado se multiplica nuevamente con el mapa promediado de la capa de abajo.
6. Los pasos 4 y 5 se repiten hasta que se alcanza la entrada. La última máscara, que tiene el tamaño de la imagen de entrada, se normaliza a un rango de 0-1 y se convierte en la máscara de visualización final.

3.1.6. DeepPilot

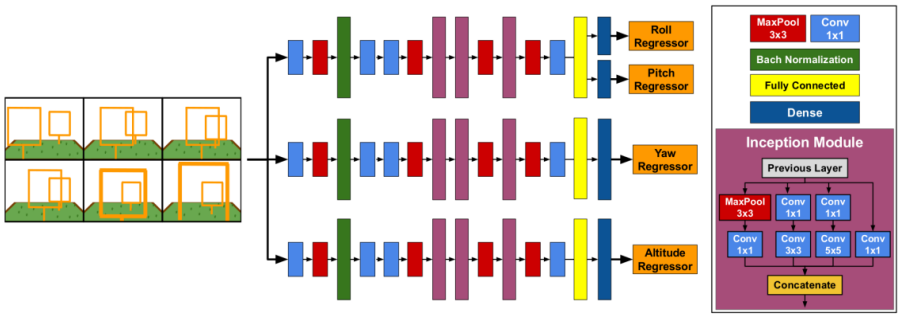


Figura 3.2: Infraestructura DeepPilot

Podemos encontrar en la literatura ejemplos como el caso del artículo publicado por Leticia Oyuki Rojas-Perez, *DeepPilot: A CNN for Autonomous Drone Racing*[3], donde se analiza y muestra esta red como una de las más fiables para el control de drones con redes neuronales extremo a extremo.

El fundamento de esta CNN, consiste en que recibe imágenes de la cámara del dron y predice 4 órdenes de vuelo que representan la posición angular del dron en yaw, pitch y roll, y la componente vertical de la velocidad, refiriéndose a la altitud.

Se basa en tener 3 modelos que corren paralelamente, uno para pitch y roll, otro para yaw y otro para la altitud. Debido a la flexibilidad de Keras[1] y TensorFlow[2], se pueden ejecutar a la vez estos 3 modelos con la misma arquitectura (la única diferencia serán los pesos). Cada rama tendrá 4 capas convolucionales, 3

[1] https://keras.io/
[2] https://www.tensorflow.org/

módulos de interceptación, 1 capa totalmente conectada y 1 capa de regresión. Cuando se entrena esta red neuronal, a cada módulo se le pasa un conjunto de datos donde el movimiento predominante sea el que se quiere entrenar.

En trabajos como el de *Conducción autónoma de un vehículo en simulador mediante aprendizaje extremo a extremo basado en visión* de Vanessa Martínez [4] se plantea el uso de imágenes consecutivas como entrada, para añadir una tendencia de movimiento y reducir el número de capas. Primero se hace una aproximación con un fotograma, pero haciendo pruebas con 2, 4, 6 y 8 fotogramas se observa que al añadir más fotogramas a la entrada, se actúa con un proceso de memoria, ya que contienen una tendencia de movimiento.

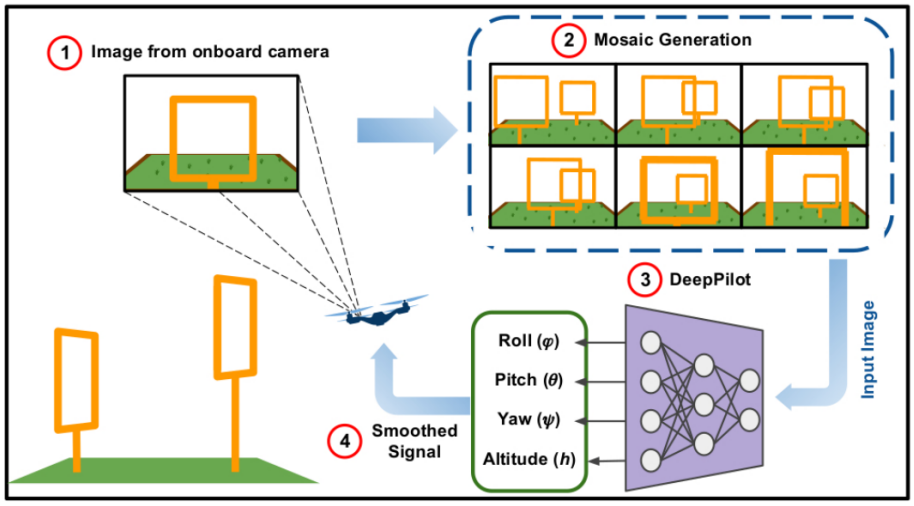


Figura 3.3: Fotogramas apilados *DeepPilot*

3.2. Infraestructura de un UAS

El control en drones es una disciplina compleja que implica la gestión y coordinación de varios subsistemas para asegurar un vuelo estable y seguro. Para entender completamente cómo operan los drones y diseñar aplicaciones específicas, es esencial desglosar sus componentes y analizar cada uno de ellos en detalle. Un UAS consta de seis elementos principales, cada uno de los cuales desempeña un papel fundamental en el funcionamiento general del sistema. Estos elementos son:

3.2.1. UA (Unmanned Aircraft)

El UA, o Aeronave No Tripulada, es la parte principal del sistema. Es la aeronave en sí misma, que puede variar en tamaño, forma y capacidad según su uso previsto. En nuestro caso se considera que trabajaremos con UAS de tipo MAV o *Small UAS*. En su caso el Crazyflie[3] como MAV y el Tello[4] como *Small UAS*. Los MAV (*Micro Air Vehicles*) son drones pequeños, ligeros y altamente maniobrables, diseñados para misiones específicas como la inspección de infraestructuras o la vigilancia. Por otro lado, los *Small UAS* son drones de tamaño compacto, fáciles de transportar y operar, ideales para tareas como la fotografía aérea, la cartografía o el entretenimiento. Otras características que podemos encontrar en la literatura, como es el caso del trabajo de Fin de Master de Pedro Arias Perez, *Infraestructura de programación de robots aéreos y aplicaciones visuales con aprendizaje profundo* [5] se describen en la imagen 3.4.

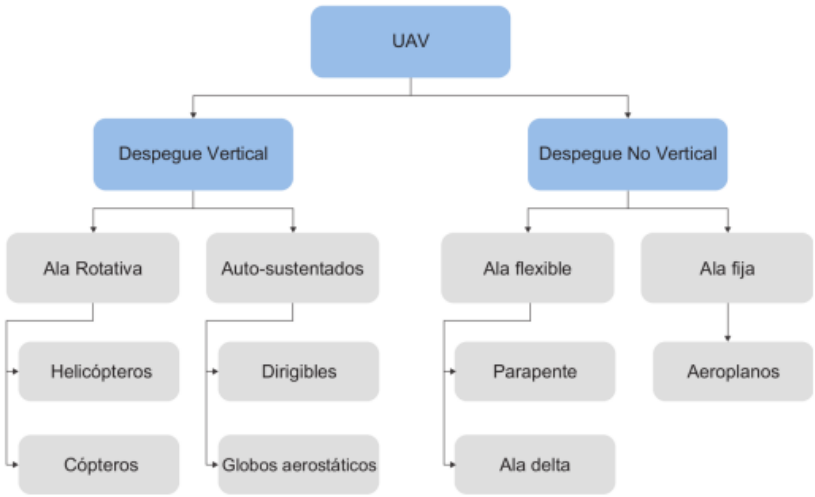


Figura 3.4: Clasificación de UAV's

3.2.2. Estación de control

La estación de control es desde donde se maneja el UA. Puede ser una estación terrestre, una consola de control remoto u otro dispositivo desde el cual el operador humano interactúa con el sistema. Tenemos 2 modos principales, automático y semi-automático. La idea del trabajo es que el dron se controle de manera automática ante estímulos reactivos, aunque implementaremos también

[3] https://www.bitcraze.io/products/crazyflie-2-1/
[4] https://store.dji.com/es/shop/tello-series

soluciones semi-automáticas para la implementación del autopiloto. QGroundControl[5] es un ejemplo de software que nos permite usar un ordenador como estación de control, este software nos permite comunicarnos con el dron mostrando sus métricas, parámetros de vuelo y herramientas de planificación de vuelo. Se puede ver una vista de la aplicación en la imagen 3.5

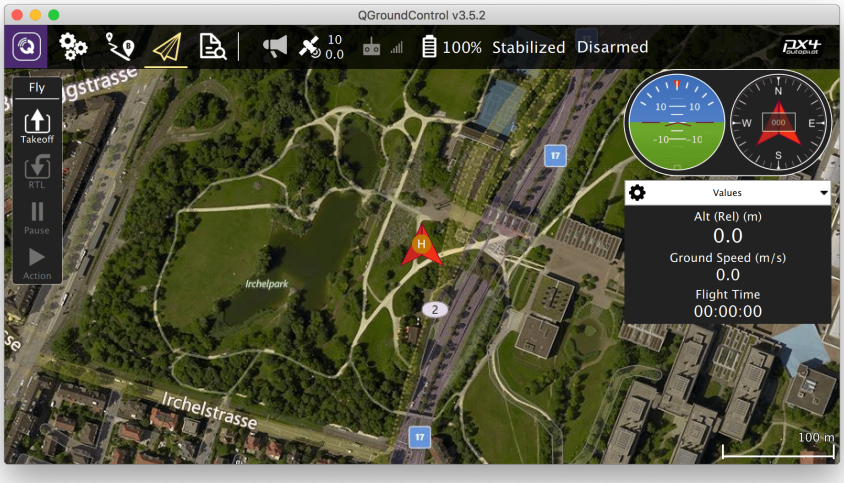


Figura 3.5: QGroundControl

3.2.3. Enlace de datos y puertos

El enlace de datos y los puertos son los canales de comunicación que permiten la transmisión de datos entre el UA y la estación de control. Esto puede incluir enlaces de radio, redes de comunicación por satélite u otros medios de comunicación. Es crucial garantizar una comunicación estable y segura entre la estación de control y el UA para una operación efectiva y segura del sistema.

3.2.4. Carga de pago

La carga de pago se refiere a cualquier equipo o dispositivo que se transporte y utilice en el UA para llevar a cabo una tarea específica. Esto puede incluir cámaras, sensores, sistemas de entrega u otros dispositivos especializados. En nuestro caso, la carga de pago consistirá en dos cámaras. La ventral será utilizada para tener la referencia del suelo y mantener la altura, mientras que la cámara frontal servirá para la aplicación en sí, ya sea el seguimiento de líneas o el cruce de ventanas. Esto

[5] http://qgroundcontrol.com/

proporcionará información visual crucial para la navegación y la ejecución de la misión.

3.2.5. Despegue y sistema de recuperación

El sistema de despegue y recuperación se encarga de las operaciones de despegue y aterrizaje del UA. Puede incluir dispositivos como catapultas, lanzadores o sistemas de aterrizaje automático. Es fundamental contar con un sistema confiable y seguro para el despegue y aterrizaje, ya que estas fases son críticas para la integridad del UA y la seguridad de la operación.

3.2.6. Elemento humano

El elemento humano se refiere a la participación humana en el sistema de UAS. Esto puede incluir operadores que controlan el UA, ingenieros que mantienen y reparan el sistema, o personal de apoyo involucrado en la planificación y ejecución de las misiones. La colaboración entre humanos y sistemas de UAS es esencial para garantizar una operación eficiente y segura, así como para abordar desafíos y resolver problemas durante la ejecución de la misión.

3.2.7. Software

La función del software es unificar todos los elementos y permitir al piloto saber el estado del dron y manejar el mismo de manera adecuada para que pueda realizar su misión de manera satisfactoria. Las plataformas de control más utilizadas son *px4*[6] y *Ardupilot*[7], estas plataformas proporcionan un conjunto de funcionalidades para navegación, control y monitorización de drones. Ambas son código abierto y poseen una extensa comunidad.

[6] https://px4.io/
[7] https://ardupilot.org/

3.3. Modelo del dron

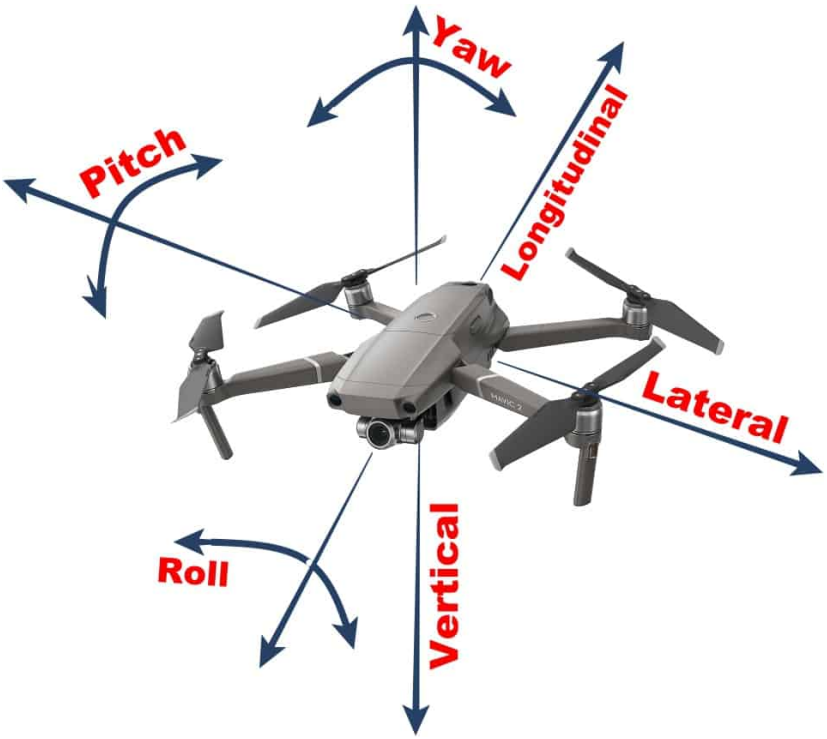


Figura 3.6: Grados de libertad de un dron

El modelo del dron describe su movimiento en un espacio tridimensional (ver Fig. 3.6). Los drones multirrotores tienen 6 grados de libertad, pudiendo moverse en dirección lateral, longitudinal y vertical, y pudiendo girar en *yaw*, *pitch* y *roll*. A continuación se presentan las ecuaciones que describen su comportamiento físico:

- Movimiento en el eje x (longitudinal):

$$\ddot{x} = \frac{1}{m}(F\sin(\theta)\cos(\psi) + F\sin(\phi)\sin(\psi)) - \frac{1}{m}F_d^x$$

- Movimiento en el eje y (lateral):

$$\ddot{y} = \frac{1}{m}(F\sin(\theta)\sin(\psi) - F\sin(\phi)\cos(\psi)) - \frac{1}{m}F_d^y$$

- Movimiento en el eje z (vertical):

$$\ddot{z} = \frac{1}{m}(F\cos(\theta)\cos(\phi)) - g - \frac{1}{m}F_d^z$$

- Rotación alrededor del eje x (roll):

$$\ddot{\phi} = \frac{\tau_\phi}{I_\phi}$$

- Rotación alrededor del eje y (pitch):

$$\ddot{\theta} = \frac{\tau_\theta}{I_\theta}$$

- Rotación alrededor del eje z (yaw):

$$\ddot{\psi} = \frac{\tau_\psi}{I_\psi}$$

Donde:

- F es la fuerza generada por los rotores.
- m es la masa del dron.
- g es la aceleración debido a la gravedad.
- F_d^x, F_d^y, F_d^z son las fuerzas de arrastre en los ejes x, y, z respectivamente.
- $\tau_\phi, \tau_\theta, \tau_\psi$ son los torques alrededor de los ejes x, y, z respectivamente.
- I_ϕ, I_θ, I_ψ son los momentos de inercia alrededor de los ejes x, y, z respectivamente.

Estas ecuaciones y modelos han sido ampliamente discutidos en la literatura, como en el artículo titulado *A Review of Quadrotor Unmanned Aerial Vehicles: Applications, Architectural Design and Control Algorithms*[6], y en el Trabajo de Fin de Grado de Manuel Lanuza Fabregat, titulado *Diseño del modelado y control de un Quad-Rotor*, presentado en la Universidad Politécnica de Cataluña [7].

4. Infraestructura

4.1. ROS2

Como base e infraestructura para el desarrollo software de este trabajo se utilizó *ROS2 Humble 16.0.9*. ROS[1] es un conjunto de librerías y herramientas para el desarrollo de sistemas robóticos con características de diseño que lo vuelven idóneo para el desarrollo del trabajo, como son:

- **Arquitectura distribuida:** Utiliza una arquitectura distribuida que permite la comunicación entre múltiples nodos de forma eficiente, lo que facilita la construcción de sistemas robóticos complejos y modulares.
- **Flexibilidad en comunicación:** Proporciona diferentes mecanismos de comunicación, como *topics*, servicios y acciones, que permiten a los nodos intercambiar datos y comandos de manera flexible y eficiente.
- **Soporte multiplataforma:** ROS está diseñado para ser compatible con varias plataformas, incluyendo Linux, Windows y macOS, lo que permite desarrollar y ejecutar aplicaciones robóticas en diferentes sistemas operativos.
- **Escalabilidad:** ROS está diseñado para ser escalable, lo que significa que puede adaptarse a diferentes tamaños y complejidades de sistemas robóticos. Desde aplicaciones simples con unos pocos nodos hasta sistemas distribuidos complejos con cientos de nodos.

ROS posee 3 componentes principales, el *workspace*, el grafo computacional y la comunidad. El *Workspace* consiste en el software instalado para el control de ROS. Aquí es donde ejecutaremos nuestra aplicación. Además, disponemos de otras herramientas útiles, como las ROSbags, que nos permitirán grabar las velocidades e imágenes del dron para el posterior entrenamiento de la red neuronal.

El *workspace* de ROS también incluye utilidades para el desarrollo de software, como el paquete de herramientas de línea de comandos de ROS, que proporciona comandos para crear, compilar y administrar paquetes de ROS. También tenemos acceso a herramientas de visualización, como RViz, que nos permite visualizar

[1] https://docs.ros.org/en/humble/index.html

datos sensoriales en tiempo real y depurar el comportamiento de nuestro dron.

En ROS, los nodos o procesos son programas que se ejecutan de forma independiente y se combinan en un grafo para realizar diversas tareas. Estos nodos se comunican entre sí de tres formas principales:

1. **Topics:** Consisten en una forma de comunicación asíncrona basada en el modelo publicador-subscriptor. Un nodo puede publicar mensajes en un topic y otros nodos pueden suscribirse para recibir esos mensajes.
2. **Servicios:** Los servicios en ROS permiten la comunicación síncrona entre nodos. Un nodo puede ofrecer un servicio que proporciona una funcionalidad específica, y otros nodos pueden solicitar esa funcionalidad enviando una solicitud al nodo que ofrece el servicio.
3. **Acciones:** Las acciones en ROS son un mecanismo de comunicación que permite la ejecución de comportamientos complejos y estructurados entre nodos. A diferencia de los servicios, las acciones permiten la comunicación asíncrona entre un cliente y un servidor. El cliente envía una meta al servidor y espera hasta que se complete la tarea asociada a esa meta. Durante el proceso, el servidor puede proporcionar actualizaciones periódicas sobre el progreso de la tarea.

Al ser software libre, ROS es una herramienta muy útil que nos permite utilizar software desarrollado por otros programadores para nuestra aplicación. Además ofrece una amplia gama de recursos y soporte para los usuarios. Esto incluye foros de discusión en línea, listas de correo, canales de chat en tiempo real y grupos de usuarios locales.

4.2. OpenCV

Para este trabajo se utilizó la versión 4.9.0 de OpenCV[2]. OpenCV es una librería para el procesamiento general de imágenes. Es de código abierto y, aunque está nativamente escrita en C++, también soporta Python y Java. Se enfoca en la eficiencia y funcionalidad en tiempo real, gracias a su integración con NVIDIA[3]

[2] https://opencv.org/

[3] https://www.nvidia.com/Download/index.aspx?lang=en-us

CUDA[4] y OpenGL[5]. Por lo tanto, es una herramienta idónea para tareas como el reconocimiento de colores y contornos, así como cálculos relativos a la imagen.

Además de su enfoque en el procesamiento en tiempo real, OpenCV ofrece una amplia gama de algoritmos y herramientas para diversas aplicaciones de visión artificial. Esto incluye funciones para el filtrado de imágenes, detección de características o seguimiento de objetos. Esta librería se utilizó de manera auxiliar para desarrollar el piloto experto del sigue líneas permitiendo al dron diferenciar la línea, y a partir de la misma saber que velocidad comandar. Algunas de las funciones que ofrece OpenCV son:

- **Operaciones Morfológicas:** Utiliza operaciones morfológicas como la erosión y dilatación para eliminar pequeñas discontinuidades o separar regiones conectadas.
- **Algoritmos de Etiquetado de Regiones:** Aplica algoritmos de etiquetado de regiones para identificar y etiquetar las regiones conectadas en la imagen. Luego, se pueden eliminar regiones siguiendo un determinado criterio.
- **Filtros de Conectividad:** utiliza filtros de conectividad para preservar solo las partes de la línea que son conexiones contiguas.
    - Transformada de Hough paramétrica (o circular)
    - Operadores de detección de bordes
    - Algoritmos de seguimiento de contornos
- **Algoritmo de Búsqueda de Componentes Conectados:** Implementa un algoritmo que busque y mantenga solo los componentes conectados en la imagen, descartando aquellas partes no contiguas.

[4] https://developer.nvidia.com/cuda-toolkit
[5] https://www.opengl.org/

4.3. Simulador

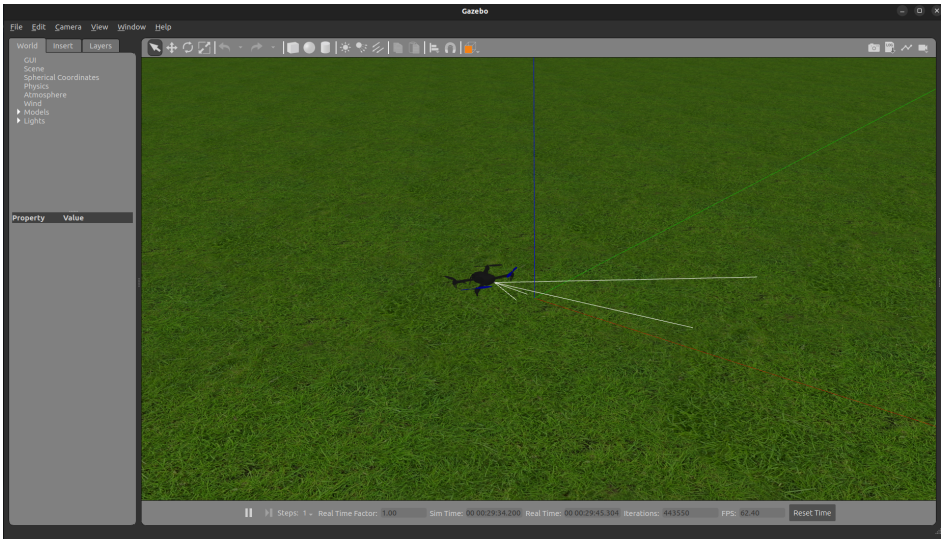


Figura 4.1: Visualización Gazebo

Gazebo[6] emerge como un pilar en el ámbito de la simulación robótica, ofreciendo un entorno de código abierto bajo la licencia Apache 2.0. Su versatilidad y precisión lo convierten en una herramienta fundamental para proyectos como este, especialmente por su integración fluida con ROS.

En este proyecto, se emplean dos versiones distintas de Gazebo para diferentes propósitos. Para la generación del *dataset* en la aplicación de seguimiento de líneas, se opta por Gazebo 11 debido a su compatibilidad con la infraestructura de *RoboticsInfrastructure*[7]. Por otro lado, se utiliza Gazebo Fortress para la aplicación de detección de ventanas, aprovechando la disponibilidad del repositorio referencia *project_crazyflie_gates*[8]. Se escogió Gazebo por que ofrecía características ideales como:

- **Simulación de físicas:** Gazebo tiene capacidad para simular de manera precisa y realista las interacciones físicas entre los objetos virtuales. Para lograr esto, emplea una variedad de motores de física avanzados, tales como ODE *(Open Dynamics Engine)*, Bullet, Simbody y DART. Estos motores permiten modelar con precisión fenómenos físicos complejos, incluyendo fuerzas, colisiones y materiales. Esta característica es fundamental para

[6] https://gazebosim.org/home
[7] https://github.com/JdeRobot/RoboticsInfrastructure
[8] https://github.com/aerostack2/project_crazyflie_gates

probar el comportamiento de robots y sistemas de control en entornos simulados antes de llevar a cabo implementaciones en el mundo real, proporcionando un entorno seguro y controlado sin riesgo de daños materiales o lesiones.

- **Entorno:** Se pueden crear entornos detallados con una amplia variedad de objetos, texturas y condiciones de iluminación. Estos entornos generalmente se codifican en el formato SDF, basado en XML.
- **Modelado de sensores y actuadores:** Gazebo proporciona modelos de sensores y actuadores que imitan de manera realista el comportamiento y las limitaciones de los modelos reales. Esto incluye cámaras, sensores LIDAR, GPS y actuadores como motores y servos.

4.4. Aerostack2

Aerostack2[9] es un entorno de código abierto creado para ayudar a diseñar y construir la arquitectura de control de sistemas robóticos aéreos, integrando múltiples algoritmos como algoritmos de visión, controladores de movimiento, métodos de auto-localización y mapeo o algoritmos de planificación de movimientos.

Se utilizó la versión 1.0.9, Aerostack2 es útil para construir sistemas aéreos autónomos en entornos complejos y dinámicos, y también es una herramienta de investigación útil para la robótica aérea para probar nuevos algoritmos y arquitecturas, es versátil para construir diferentes configuraciones de sistemas con diversos grados de autonomía. Sus características más importantes son:

- Puede ser utilizado para vuelos de teleoperación, pero también ser utilizado para construir sistemas robóticos autónomos para realizar misiones aéreas sin asistencia de operador.
- Se puede emplear para volar un enjambre de drones heterogéneos para realizar misiones con múltiples robots aéreos o un solo dron.
- Es independiente del hardware, lo que significa que se ejecuta en computadoras portátiles convencionales o en computadoras a bordo como Nvidia Jetson NX. Aerostack2 se ha utilizado en diferentes plataformas aéreas, incluyendo, pero no limitado a: plataformas DJI (Matrice 210RTKv2,

[9] https://aerostack2.github.io/_00_getting_started/index.html

Matrice 300, Ryze Tello), autopilotos Pixhawk y drones Crazyflie. El marco puede operar en simulación y en un entorno real de manera similar, lo que simplifica el desarrollo de *Sim2Real*.

- Completa modularidad, permitiendo que los elementos se cambien o intercambien sin afectar al resto del sistema. La arquitectura basada en *plugins* permite utilizar diferentes implementaciones para cada tarea..

  Esta plataforma emplea ROS2 y Linux como *middleware*. Su infraestructura se puede ver en el esquema 4.2 el cual es explicado a continuación:

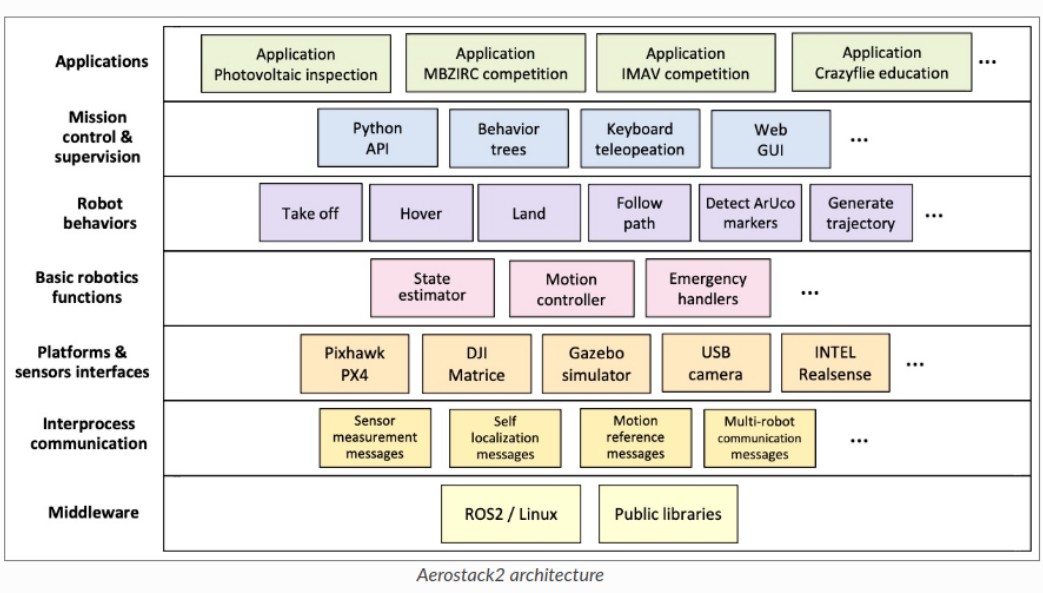


Figura 4.2: Infraestructura de Aerostack2

Comunicación de interprocesos

Incluye componentes que facilitan la comunicación entre procesos operando en tiempo real, mejorando la interoperabilidad entre los mismos. Como se muestra en la imagen, los módulos de control son los siguientes con sus respectivas comunicaciones:

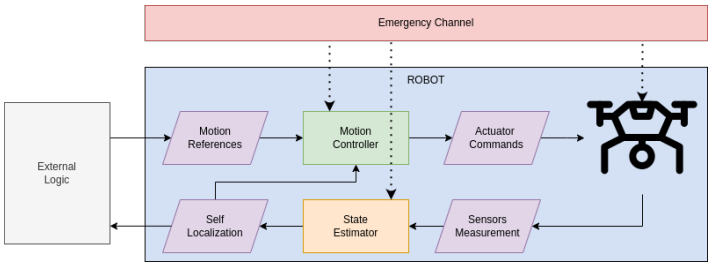


Figura 4.3: Comunicaciones de Aerostack2

- ***Motion reference*:** Principalmente se encarga de los modos de vuelo entre los que se encuentran el control de posición, el control de velocidad, el control de velocidad en plano, control de trayectoria, control en posición de *yaw* y control de velocidad en *yaw*.
- ***Actuator Commands*:** Una vez seleccionada la referencia de movimiento, en este canal se mandarán los comandos de actuación.
- **Medidas de sensores:** El estado estimador usa las medidas para realizar una fusión sensorial.
- **Canal de emergencia:** En caso de afrontar una situación de emergencia, este canal permite al operador tomar la acción necesaria para poner a salvo el sistema o prevenir un riesgo mayor.

Plataforma y sensores

Estos son componentes que sirven como interfaces con plataformas aéreas y sensores. Aerostack tiene varias interfaces que permiten operar tanto con plataformas físicas (por ejemplo, con plataformas Pixhawk o DJI) como con plataformas simuladas (por ejemplo, usando drones simulados con el entorno Gazebo), además de diferentes tipos de sensores.

Funciones robóticas

Aerostack2 incluye un conjunto de componentes de software que implementan algoritmos especializados correspondientes a funciones robóticas aéreas esenciales para la operación autónoma, como la estimación de estado, el control de movimiento y otras funciones básicas (por ejemplo, manejo de emergencias, etc.).

Comportamientos

Este nivel incluye un conjunto de componentes correspondientes a diferentes comportamientos de robot proporcionados por Aerostack para la operación autónoma. Cada componente encapsula los algoritmos utilizados para implementar un comportamiento particular (por ejemplo, despegue, planeo, generación de trayectorias, etc.) junto con mecanismos para monitorear la ejecución para facilitar la especificación de planes de misión. Cada comportamiento tiene

una interfaz uniforme que es común para todos los comportamientos para facilitar su uso.

Control de Misión

Este nivel incluye componentes que facilitan la especificación de misiones para la operación autónoma de drones. Por ejemplo, se pueden utilizar árboles de comportamiento para indicar y visualizar con una estructura jerárquica los tareas realizadas por el dron. Por otro lado, Aerostack también proporciona una API que permite especificar misiones de manera flexible utilizando el lenguaje Python. Esta API hace uso de otro componente para abstraer información relevante en forma de representación simbólica de creencias. Además, Aerostack proporciona herramientas al usuario para monitorear y controlar manualmente la ejecución de la misión.

Aplicación

El nivel superior corresponde a las aplicaciones específicas construidas con los componentes de los niveles inferiores. Aerostack tiene ejemplos de aplicaciones que pueden servir a los desarrolladores como referencia y guía sobre cómo construir drones que operen de manera autónoma.

4.5. Python

Python es un lenguaje de programación de alto nivel, interpretado, multiparadigma y de tipado dinámico. Fue creado a finales de los 80 por Guido van Rossum y su diseño se caracteriza por su simplicidad y legibilidad. Es uno de los lenguajes más populares en la actualidad debido a su versatilidad, facilidad de aprendizaje y amplia comunidad de desarrolladores. Durante el desarrollo del trabajo se utilizó la versión 3.10.12

4.5.1. Pytorch

PyTorch *(V2.1.2+cu121)* es un entorno de *Deep Learning* de cómputo numérico de alto rendimiento centrado en el cálculo en paralelo utilizando tensores. Su

principal ventaja frente a otras herramientas radica en su capacidad para utilizar grafos dinámicos denominados *reverse-mode auto-differentiation*, lo que permite cambiar el comportamiento de las redes neuronales de forma arbitraria sin coste adicional. Esto lo hace muy potente para trabajar con GPU o CPU, permitiendo el cálculo distribuido a través de la librería. Gracias a su simplicidad y extensa documentación se eligió esta opción para el tratamiento del *dataset* y el entrenamiento de la red.

4.5.2. Albumentation

Albumentations *(V1.4.0)*[10] es una librería de Python utilizada en deep learning y visión artificial para aumentar la calidad de los modelos entrenados. El propósito del aumentado es crear nuevas imágenes a partir de una original para generar nuevas muestras de entrenamiento a partir de la misma imagen. Las técnicas más empleadas son:

- **Transformaciones geométricas**: Cambios en la forma, tamaño, posición o orientación de una imagen.
- **Cambios en el espacio de color**: Modificaciones en la representación del color de una imagen, como conversiones entre espacios de color o ajustes de saturación y tono.
- **Filtros de kernel**: Técnicas que aplican operaciones locales a los píxeles de una imagen, como el desenfoque, la nitidez o la detección de bordes, utilizando máscaras de convolución.
- **Eliminación de partes aleatorias**: Proceso de eliminar elementos no deseados o ruidosos de una imagen, como manchas, artefactos o elementos irrelevantes.
- **Combinación de imágenes**: Integración de múltiples imágenes en una sola, ya sea mediante superposición, fusión o composición.
- **Cambio de perspectiva**: Ajuste de la vista de una imagen para simular diferentes ángulos de visión o puntos de vista.
- **Deformaciones elásticas**: Transformaciones que distorsionan una imagen de manera controlada para simular efectos de deformación o elasticidad.
- ***Mirroring***: Replicación de una imagen reflejándola horizontal o verticalmente, o mediante rotaciones para crear simetrías o efectos especiales.

[10] https://albumentations.ai/

En el siguiente artículo de Vladimir Lyashenko, *Data Augmentation in Python: Everything You Need to Know* [8] encontramos una comparación de las distintas liberarías para aplicar la técnica de augmentation:

Librería	**Pipeline simple**	**Pipeline complejo**
Augmentor	31.9	28.2
Albumentations	10.9	17.7
ImgAug	12.04	30.9
Transforms	9.8	15.2

Tabla 4.1: Comparación de tiempo de ejecución entre diferentes bibliotecas de aumento de datos

5. Aplicación sigue líneas

La primera aplicación que se realizará para explorar el desempeño de *imitation learning* en drones será el seguimiento de línea. Aprovechando circuitos previamente creados para el entrenamiento de coches con este mismo propósito, como es el caso del TFM de Vannessa Martínez, *Conducción autónoma de un vehículo en simulador mediante aprendizaje extremo a extremo basado en visión* [4], donde se implementa para vehículos terrestres, o el TFM de Ignacio Arranz, *Conducción autónoma de un robot con visión mediante aprendizaje por refuerzo.* [9], donde se implementa este sistema con aprendizaje por refuerzo.

Para esta aplicación, será necesario crear un conjunto de datos lo suficientemente grande para que el dron pueda obtener resultados óptimos, así como recorridos de referencia para cuantificar los resultados. Esta aplicación, al ser una tarea sencilla, nos permitirá demostrar si *PilotNet* es realmente aplicable y eficaz en drones. Además, al ser una aplicación fácilmente cuantificable, se podrán obtener datos precisos de rendimiento, centrándose principalmente en la desviación entre la ruta ideal y la ruta seguida por el piloto neuronal, junto al tiempo que se tarda en completar el circuito.

5.1. Piloto experto algorítmico

Para entrenar una red capaz de seguir líneas en diferentes circuitos de manera efectiva, lo primero que se tuvo que desarrollar fue un piloto experto algorítmico con el que generar el *dataset* de entrenamiento. Uno de los aspectos cruciales en *imitation learning* es la calidad del *dataset* y con el piloto algorítmico se evita la necesidad de tener a un piloto manejando constantemente el dron, tarea que sería muy cansada para el operador y además muy susceptible a errores. Se desarrolló un piloto experto capaz de seguir la línea por sí mismo utilizando técnicas tradicionales de programación.

5.1.1. Detección de línea

Para detectar la línea de forma precisa, se aplicaron filtros de color de OpenCV a la imagen obtenida por la cámara ventral del dron para diferenciar únicamente el color rojo. Sin embargo, esto no proporcionaba una solución perfecta, ya que había diferentes elementos en el circuito que también eran rojos e interferían con el sistema.

Para abordar este problema, se realizó una operación de apertura en la imagen. La apertura consiste en una combinación de erosión para una posterior dilatación, con ambas técnicas conseguiremos eliminar el ruido de la imagen y resaltar la línea de manera efectiva. Primero, aplicamos la erosión para reducir el tamaño de los objetos no deseados y eliminar pequeños detalles que puedan interferir con la detección de la línea. Luego, aplicamos la dilatación para restaurar el tamaño de la línea y conectar segmentos que pudieran estar interrumpidos o incompletos, asegurando así que la línea esté bien definida y continua.

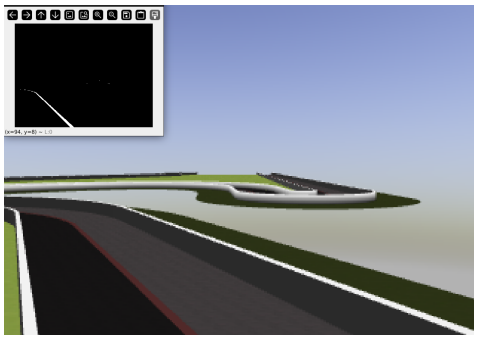


Figura 5.1: Visualización de filtro de color

Figura 5.2: Apertura aplicada

Después de aplicar la apertura, surgió otro problema: debido a la altura del dron, este detectaba líneas distantes y no solo las contiguas, lo que provocaba que el dron actuara de manera poco efectiva perdiendo precisión al considerar partes más lejanas del circuito. Para resolver este inconveniente, se exploraron varias posibilidades y se concluyó en la aplicación de un filtro de Canny para encontrar los bordes de las líneas y distinguir entre diferentes regiones. El filtro de Canny se basa en 5 etapas principales:

1. **Reducción de ruido**: Se aplica un filtro Gaussiano para suavizar la imagen y reducir el ruido, lo cual mejora la detección de bordes.

2. **Cálculo del gradiente de la intensidad**: Se utilizan operadores de derivadas (como operadores de Sobel) para calcular la magnitud y la dirección del gradiente de la intensidad de la imagen.
3. **Supresión de no máximos**: Se realiza un proceso de supresión para mantener únicamente los píxeles que forman parte de los bordes más marcados.
4. **Umbralización**: Se aplican dos umbrales a la imagen de gradiente resultante: un umbral bajo y un umbral alto. Los píxeles con valores de gradiente que superan el umbral alto se consideran píxeles de borde fuertes, mientras que los píxeles que están por debajo del umbral bajo se descartan.
5. **Conexión de bordes por histéresis**: Finalmente, se conectan los píxeles de borde débiles que están cerca de los píxeles de borde fuertes para formar bordes completos en la imagen.

Figura 5.3: Localización de contornos

Gracias a las funciones de OpenCV, esta operación se realiza con bajo coste computacional. Después de aplicar el filtro, se selecciona el contorno más grande y se rellena este mismo, lo que da como resultado una imagen binaria donde los píxeles en blanco corresponden a la línea adyacente. En la figura 5.4 podemos ver la imagen filtrada después de la dilatación y en la figura 5.5 podemos ver la imagen resultante después de aplicar el filtro de Canny.

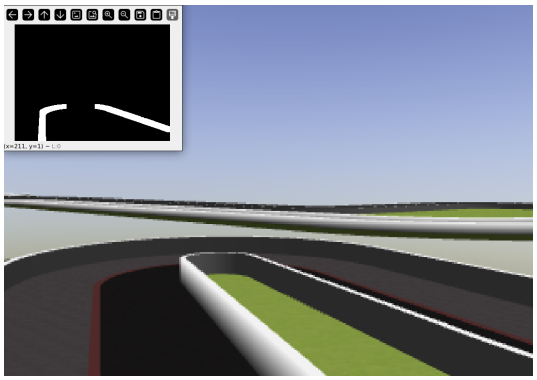


Figura 5.4: Imagen sin selección de contornos

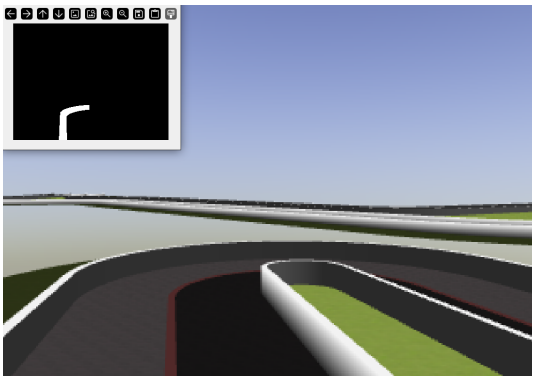


Figura 5.5: Imagen con selección de contornos

5.1.2. Control de seguimiento

Para el control del seguimiento de línea, se implementaron estrategias para regular tanto la velocidad angular en *yaw* como la velocidad lineal del dron. En esta aplicación, las componentes de altitud y el movimiento lateral (*roll*) se dejaron constantes, centrándonos solo en el control de las dos primeras. Se utilizaron técnicas de control y filtros complementarios para mejorar la estabilidad y precisión del sistema. Cabe destacar que la parte de control y la de detección de línea se realizaron en nodos distintos para distribuir el software de manera adecuada.

Velocidad Angular

Para calcular la velocidad angular, se tomaron como referencia varias franjas en la imagen captada por el dron. La media del centro del contorno en estas franjas se utilizó para determinar la desviación del dron respecto a la línea. La desviación media de esta línea de puntos respecto al centro de la imagen se consideró como el error en la velocidad angular. A este error se le aplicó un controlador derivativo aumentando la estabilidad del sistema. Los puntos medios de cada franja vienen representados de verde en la imagen 5.6.

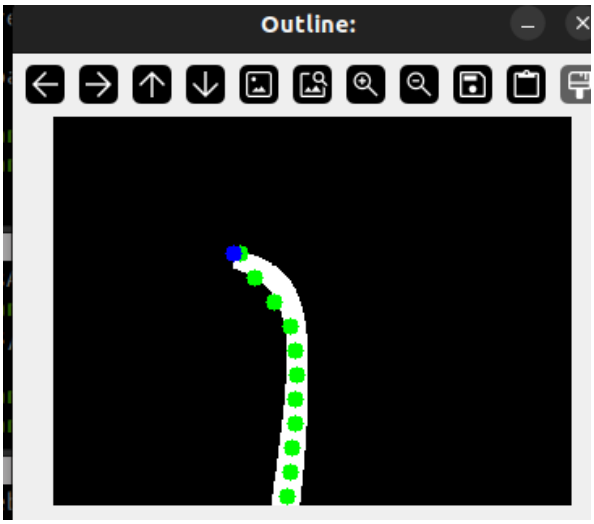


Figura 5.6: División de línea

El controlador derivativo se encarga de anticipar las tendencias del error, calculando la tasa de cambio del error en el tiempo. Esto significa que, si el dron está alejándose rápidamente de la línea, el componente proporcional se encargará de contrarrestar este error antes de que el mismo aumente significativamente. La parte derivativa se encargará de que la contrarespuesta no sea excesiva y el dron oscile en torno a la línea.

A pesar de las oscilaciones que pueda sufrir la velocidad angular, como se muestra en la figura 5.7, los cambios eran mínimos. Para no perder reactividad, se decidió mantener la salida sin ningún adicional, dejando el controlador con un PD y un limitador de velocidad como única técnica de control.

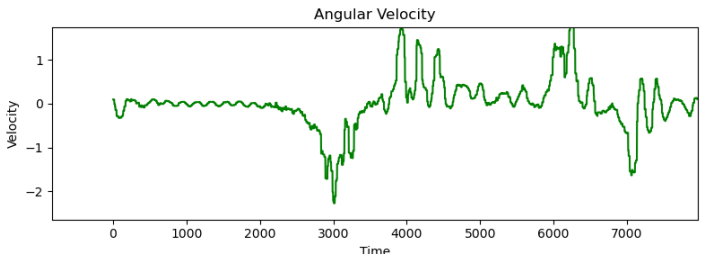


Figura 5.7: Gráfica de velocidad angular

Velocidad Lineal

La velocidad lineal se calculó de manera inversamente proporcional al error; es decir, a menor error, mayor será la velocidad, pero en este caso solo se tomará el punto mas lejano de la línea (punto azul en la figura 5.6), de esta forma, el dron

tendrá una mayor anticipación y tendrá mas tiempo para frenar, evitando cambios excesivamente bruscos. Tanto para la velocidad angular como para la lineal, en esta componente, también se implementó un controlador PD para evitar cambios bruscos en la velocidad y lograr un mejor control del dron. Sin embargo, esto no fue suficiente para un control óptimo, por lo que se consideró la adición de filtros complementarios al sistema de control. Estos filtros tienen la función de suavizar las señales de control y reducir los cambios bruscos en el dron, mejorando así la estabilidad general del seguimiento de línea.

Además, se encontraron otras técnicas de control de velocidad, como se expone en el artículo *Autonomous Drone Racing: Time-Optimal Spatial Iterative Learning Control within a Virtual Tube* [10], donde se discuten métodos adicionales para optimizar la velocidad de respuesta en sistemas mecánicos. En este artículo, se propone un enfoque sin modelo para carreras de drones autónomos. La metodología incluye el uso de un tubo virtual en el que el dron debe ajustar su trayectoria de manera óptima en términos de tiempo, aprendiendo iterativamente comandos de control para pasar por las puertas de la carrera lo más rápido posible. Dando la idea de generar una especie de tubo donde se puedan limitar las posiciones o velocidades del dron.

Para el control de la componente lineal, se empleó un filtro de paso bajo para suavizar la reactividad. Cambios bruscos en la velocidad lineal pueden causar que la cámara del dron se eleve y desestabilice provocando que el aprendizaje de la red sea mucho más complejo, por lo que se trató en reducir este problema en medida de lo posible.

Como se puede apreciar en las gráfica 5.8, la media añade un pequeño offset y reduce el tiempo de reacción. Aunque esto disminuye la reactividad, los resultados en la práctica son bastante buenos y justifican su uso. Dado que la frecuencia de muestreo es de 233.17 Hz, se ejecuta una iteración cada 4 ms. Suponiendo un buffer de medidas de 150 muestras, un cambio total de velocidad se produciría en 643 milisegundos. Este tiempo es bastante aceptable para evitar frenadas bruscas que puedan desestabilizar el dron y provocar que la cámara se eleve.

Figura 5.8: Gráfica de velocidad lineal

5.2. Creación y manejo de conjunto de datos

El *dataset* es el punto más crítico a la hora de desarrollar un proyecto de *machine learning*, por lo que su correcta utilización puede marcar la diferencia entre el éxito o el fracaso de nuestra aplicación.

Inicialmente, el *dataset* se almacena en rosbags. Con el fin de mejorar la generalización y permitir su utilización en diversas aplicaciones, se estructuró en dos directorios: uno para las imágenes grabadas en un formato reducido para optimizar el almacenamiento, y otro para los archivos de etiquetas. Cada archivo de etiqueta lleva el mismo nombre que su respectiva imagen asociada, estas se asignan por emparejamiento de sellos temporales de cada uno.

Para la obtención de este *dataset*, se creó un script de automatización capaz de ejecutar la aplicación con el piloto experto algorítmico, grabar los datos necesarios y cerrar la aplicación para la posterior estandarización de los datos obtenidos. Facilitando así la obtención de un *dataset* extenso sin la necesidad de la supervisión humana.

5.2.1. Balanceado de *dataset*

La redistribución de etiquetas, realizada en base a gráficos y análisis previos, permitió una mejora en los resultados, aunque no fue suficiente para resolver todos los problemas.

Para asegurar el adecuado funcionamiento de la red neuronal, se procedió a dividir el *dataset* en distintos umbrales de velocidad, denominados *labels*. Estos *labels* definen la clasificación del *dataset* en velocidades bajas, altas e intermedias tanto en velocidad angular como lineal, dado el uso de *PilotNet*.

Dividir el *dataset* en estos umbrales de velocidad tiene varias ventajas importantes.

1. Facilita el entrenamiento de la red neuronal al permitir que el modelo aprenda a manejar diferentes rangos de velocidad de manera específica.
2. Al tener categorías claramente definidas, el modelo puede ajustar sus parámetros para optimizar el rendimiento en cada rango de velocidad, lo que mejora la precisión y eficiencia del dron en situaciones variadas.
3. Ayuda a abordar el problema del desbalance de datos, una situación común en *datasets* reales donde ciertos casos son mucho más frecuentes que otros. Al ajustar la ponderación de los casos menos frecuentes, se asegura que la red neuronal no se sesgue hacia los casos más comunes, sino que también aprenda a manejar adecuadamente las situaciones menos probables pero potencialmente críticas. Esto es crucial para el desempeño robusto del dron, ya que le permite reaccionar de manera adecuada a una amplia gama de escenarios, incluyendo aquellos que ocurren con menor frecuencia pero que son vitales para su operación segura y eficiente.

Para graficar de manera precisa tanto el número de muestras como su distribución se utilizaron la librería de Python matplotlib[1]. En parte superior de la gráfica 5.9 se presenta la cantidad de muestras asociadas a cada etiqueta, mientras que la gráfica inferior muestra la distribución de las etiquetas junto con sus respectivas velocidades.

Es importante destacar que las distribuciones a la izquierda corresponden al *dataset* original sin procesar, mientras que las distribuciones a la derecha son del *dataset* parcialmente balanceado. Este último ajuste equilibra los casos más frecuentes y los más raros para preservar patrones repetitivos.

[1] https://matplotlib.org/

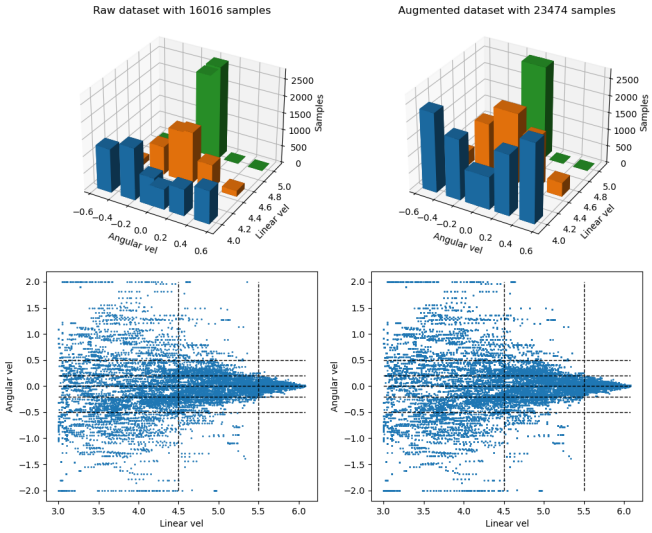


Figura 5.9: Distribución *dataset*

5.2.2. *Augmentation*

La *augmentation* se realizó de forma *offline*. En otras palabras, antes del entrenamiento. El aumento de datos es una técnica en *machine learning* que se utiliza para incrementar el tamaño y la diversidad del conjunto de datos de entrenamiento. Esto se logra aplicando diversas transformaciones y perturbaciones a los datos originales, generando nuevas muestras artificiales que ayudan a mejorar la capacidad del modelo para generalizar y manejar variaciones en los datos reales. Dentro de las técnicas de aumento de datos, se implementaron únicamente aquellas que podrían contribuir significativamente a mejorar el rendimiento de nuestro modelo.

Transformaciones geométricas

Esta técnica implica el desplazamiento lateral de la imagen. Al desplazar la imagen horizontalmente, se desplaza proporcionalmente el error horizontal. Considerando el control ejercido por el piloto experto, el primer paso para determinar la nueva velocidad asociada a la nueva muestra será normalizar el error añadido en píxeles respecto al máximo error de píxeles ($Max\_errpixel$), utilizando los límites de velocidad ($MaxVel$ y $MinVel$):

$$err_{Vel} = \frac{errPixel}{Max\_errpixel} \times (MaxVel - MinVel) + MinVel$$

Una vez obtenido el error en velocidad, se aplicarán las ganancias proporcionales del controlador PID del piloto experto para mantener la consistencia:

$$err_{\text{angular}} = err_{Vel_{\text{Angular}}} \times Kp_{\text{angular}}$$
$$err_{\text{lineal}} = err_{Vel_{\text{Lineal}}} \times Kp_{\text{lineal}}$$

Sumando este error a la velocidad asociada a la imagen, obtendremos una muestra nueva para aumentar el tamaño de nuestro *dataset*. 5.13

Espacio de color

A través del ajuste de la saturación, cambio de tono o normalización del color, incrementamos la variabilidad de los datos de entrenamiento y mejoramos la capacidad del modelo para generalizar a datos nuevos y no vistos independientemente de la iluminación del entorno. 5.10, 5.12

Mirroring

El *mirroring* consiste en generar la imagen espejo de otra imagen. En nuestra aplicación, esta técnica es aplicable ya que permite girar la imagen e invertir el signo de la velocidad angular, proporcionando ejemplos adicionales útiles para el modelo. En aplicaciones no simétricas no se podría utilizar esta operación como en el caso del seguir un carril concreto. 5.11

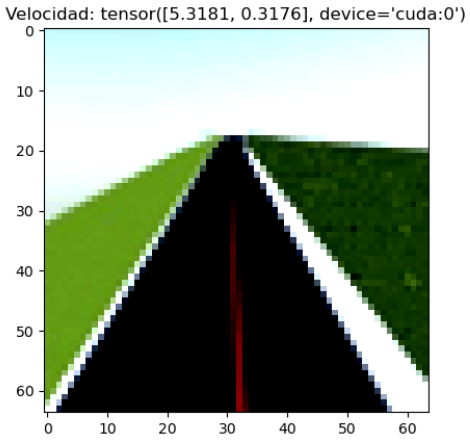


Figura 5.10: Cambio de iluminación

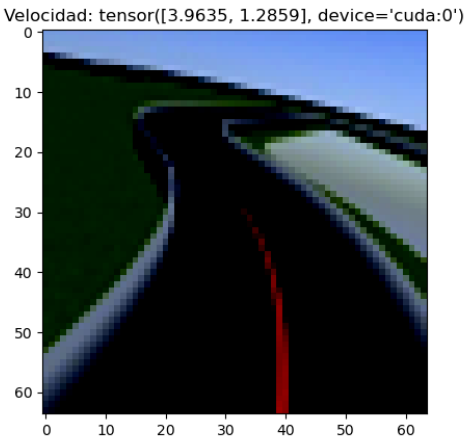


Figura 5.11: Imagen espejo

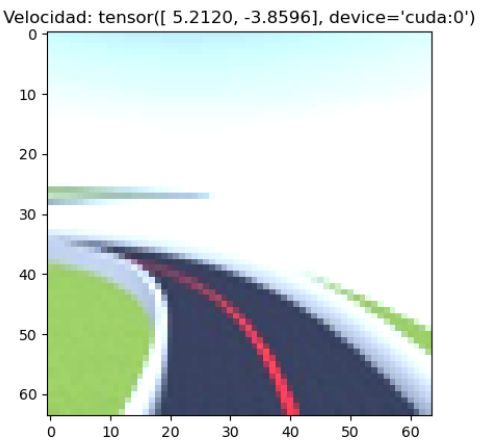


Figura 5.12: Saturación

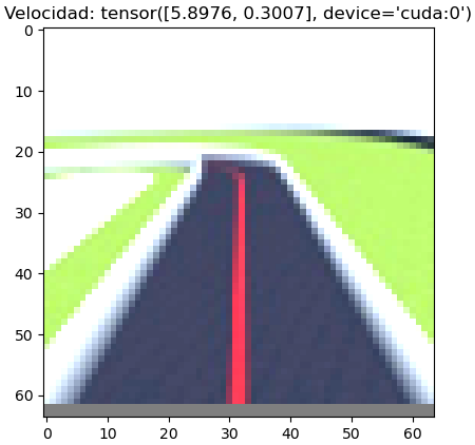


Figura 5.13: Desplazamiento

5.2.3. Estudio con diferentes *dataset*

Para evaluar la velocidad y posición, se registraron las transformaciones *(tf's)* en cada instante durante tres ejecuciones en cada circuito. La primera ejecución fue realizada por un piloto experto lento, priorizando la precisión de la posición para lograr un seguimiento perfecto de la línea, permitiendo así evaluar la desviación del dron respecto a la trayectoria ideal.

En la segunda ejecución, el mismo piloto experto que proporcionó el *dataset* realizó la vuelta, cuantificando su desviación y el tiempo necesario para completarla. Esto permitió una comparación directa con la tercera ejecución, llevada a cabo por el piloto neuronal.

Para calcular el tiempo por vuelta, se estableció un punto inicial y se esperó a que el dron regresara a ese mismo punto. Para evaluar el error de posición, se tomó como referencia la primera ruta y se comparó con las siguientes ejecuciones, calculando la desviación mediante el error medio entre cada punto.

Durante las pruebas en el circuito, se demostró que la red neuronal pudo terminar el circuito completo a pesar de no estar incluido en su *dataset* de entrenamiento, lo cual indica un entrenamiento satisfactorio obteniendo, siguiendo la siguiente evolución hasta conseguir finalmente el resultado esperado:

Dataset crudo

Como una primera aproximación para comprobar el funcionamiento de la red, se realizó un entrenamiento con un *dataset* desbalanceado. Esta prueba demostró

que sin un balance adecuado, la red no será capaz de completar el circuito. En la gráfica 5.14 se observa la distribución del *dataset* original, mientras que en la gráfica 5.15 se muestra la curva de entrenamiento.

- **Volumen del *dataset*:** 14152
- **Estado:** Crudo
- **Tiempo de entrenamiento:** 4 h
- **Pérdida final:** 0.86

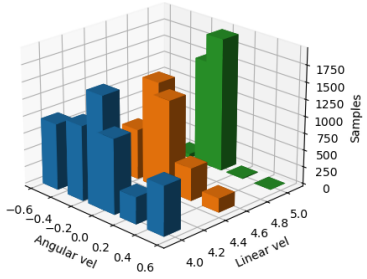


Figura 5.14: *Dataset* crudo

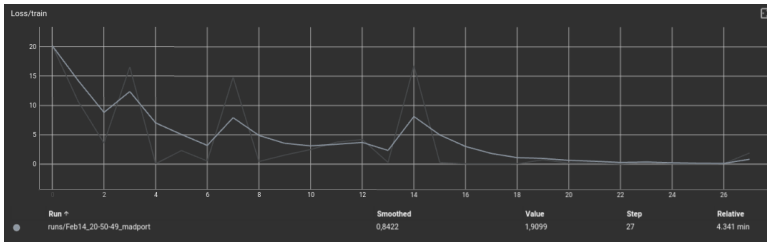


Figura 5.15: Gráfica de entrenamiento *dataset* crudo

Dataset totalmente balanceado

La siguiente prueba consistió en balancear totalmente el *dataset*, sin embargo el dron no consiguió completar el circuito, dando mejores resultados que la distribución anterior pero no suficientes. Se detectó una tendencia a la media entre todas las medidas en la ejecución de esta red, de aquí se llegó a la conclusión de que el modelo se podría estar sobrentrenado por lo que se añadió la copia de pesos en distintos periodos del entrenamiento para evitar este problema.

- **Volumen del *dataset*:** 34752
- **Estado:** Totalmente balanceado
- **Tiempo de entrenamiento:** 6 h
- **Pérdida final:** 0.91

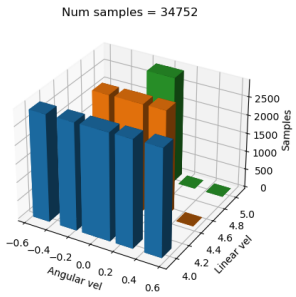


Figura 5.16: *Dataset* balanceado

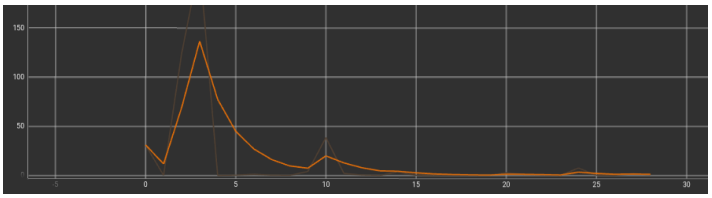


Figura 5.17: Gráfica de entrenamiento *dataset* balanceado

Aumento de labels

Se siguió el mismo procedimiento añadiendo un mayor número de etiquetas y utilizando un *dataset* semi-balanceado, donde se incrementó la cantidad de muestras de los casos más comunes mientras se introducían más situaciones inusuales. A pesar de estos ajustes, los resultados obtenidos no fueron satisfactorios. En la gráfica de entrenamiento 5.19 se observan varios intentos con diferentes distribuciones de datos; en este caso, los datos mostrados corresponden a la gráfica azul.

- **Volumen del *dataset*:** 26064
- **Estado:** Semi-balanceado
- **Tiempo de entrenamiento:** 6 h
- **Pérdida final:** 0.75

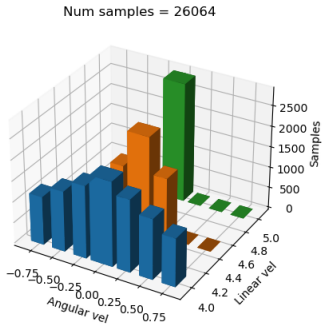


Figura 5.18: *Dataset* semi-balanceado

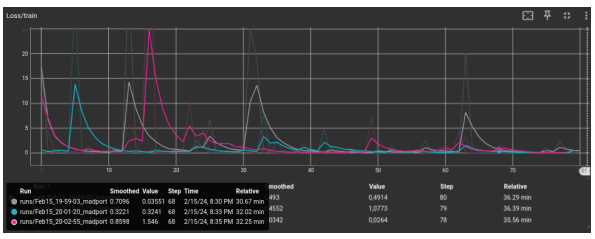


Figura 5.19: Gráfica de entrenamiento de aumento de labels y *dataset semibalanceado*

Dataset semi balanceado y ampliación de *dataset*

En las siguientes pruebas se grabó de nuevo un *dataset* con un mayor número de situaciones y se desarrolló una gráfica inferior para ver la distribución de la variación de velocidades, como se puede ver en la figura 5.20. Gracias a esto, se pudo visualizar como las velocidades quedaban distribuidas y así completar con situaciones que no estuvieran reflejadas en la gráfica. Podemos observar como, cuanto mayor es la velocidad lineal, menor es la velocidad angular. Esto se intuye gracias a que a cuanta mas velocidad lineal, mas concentradas están las velocidades entorno a 0 y que a cuanto menor velocidad lineal, mayor es la dispersión de la velocidad angular. Esto se traduce en que para ir en línea recta a máxima velocidad, tendremos que estar perfectamente alineados con la línea, pero para tener un giro brusco existe un número mas extenso de situaciones donde esta acción es necesaria. Gracias a esta distribución y a un *dataset* semibalanceado, se consiguió que el dron completara todos los circuitos.

- **Volumen del *dataset*:** 23474
- **Estado:** Parcialmente balanceado
- **Tiempo de entrenamiento:** 5:30 h
- **Pérdida final:** 0.54

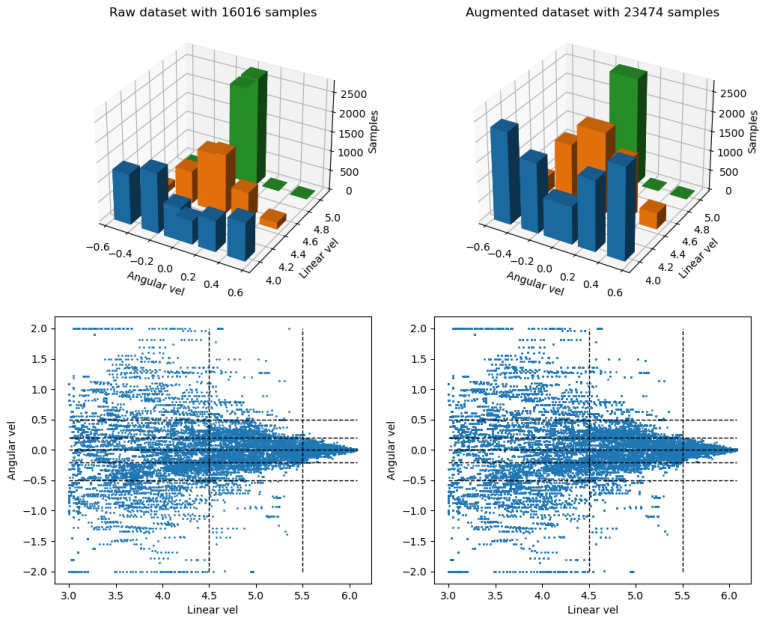


Figura 5.20: Distribución del *dataset* ampliado

5.3. Modelo neuronal y entrenamiento

Para esta aplicación se optó por utilizar el modelo *PilotNet*. Aunque otros modelos como *DeepPilot* son capaces de abordar el problema, se eligió *PilotNet* debido a su bajo coste computacional. A diferencia de *DeepPilot*, que tiene un modelo con una única salida y requiere entrenar la misma red tres veces para cada componente angular del dron, *PilotNet* está formada por una sola red neuronal que devuelve directamente las salidas necesarias.

Para el entrenamiento de *PilotNet*, se aprovechó la capa de abstracción proporcionada por aerostack2, que permite comandar directamente las velocidades lineales y angulares del dron en esta primera aplicación. Además siguiendo con la filosofía de automatizar todo lo posible la aplicación, una vez obtenidas las muestras normalizadas con el piloto algorítmico experto, se creó otro script complementario para iniciar posteriormente el entrenamiento y guardar copia de seguridad en su proceso.

5.3.1. Función de pérdida

Como criterio de pérdida se utilizó *MSELoss (Mean Squared Error Loss)*. Este criterio mide la media de los errores cuadrados entre los valores predichos por la red y los valores reales. La elección de *MSELoss* se debe a su simplicidad y efectividad en problemas de regresión, como el control de las velocidades del dron. El criterio de pérdida viene descrito por:

$$\text{MSELoss} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

donde y_i son los valores reales y $\hat{y}_i$ son los valores predichos por la red neuronal.

5.3.2. Función de optimización

Se utilizó `optim.SGD` *(Stochastic Gradient Descent)* ya que es un método de optimización sencillo y eficiente. Este optimizador actualiza los parámetros del modelo en la dirección del gradiente negativo de la función de pérdida, con el objetivo de minimizar dicha función. La elección de SGD se basa en su bajo coste computacional y su eficacia en la convergencia de modelos de red neuronal en

tareas de regresión y clasificación. La actualización de los parámetros en SGD se realiza según la siguiente fórmula:

$$\theta = \theta - \eta \nabla_\theta \mathcal{L}$$

donde:

- θ representa los parámetros del modelo,
- η es la tasa de aprendizaje,
- $\nabla_\theta \mathcal{L}$ es el gradiente de la función de pérdida respecto a los parámetros.

5.3.3. Criterio de finalización

El criterio de finalización que proporcionó los mejores resultados fue el de *n epochs*, en el que se establece un número determinado de *epochs* para el entrenamiento. Sin embargo, se observó que un mayor número de *epochs* no siempre garantizaba un mejor rendimiento de la aplicación, lo cual podría ser indicativo de un sobreajuste. Para abordar este problema, se implementó un script que guardaba varias configuraciones del modelo en intervalos regulares de tiempo durante el entrenamiento. Esto permitió evaluar diferentes etapas del entrenamiento y seleccionar la configuración que ofreciera los mejores resultados, minimizando el riesgo de sobreajuste y mejorando la generalización del modelo.

5.4. Validación experimental y evaluación

5.4.1. Validación experimental

Con cada prueba de cada dataset se graficó cada ejecución del dron para así poder ver su error medio y tiempo por vuelta en caso de completar el circuito. Viendo resultados como los del *dataset* crudo, mencionado en el capítulo 5.2.3 (fig. 5.21), o en el resultado del dataset totalmente balanceado, capítulo 5.2.3 en la figura 5.22, se aprecia como estas ejecuciones no consiguieron completar el circuito.

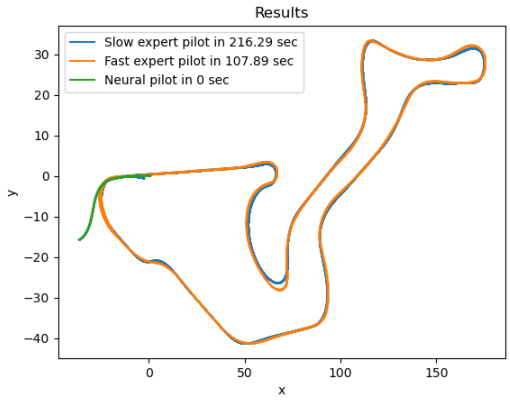


Figura 5.21: Resultado del *dataset* crudo

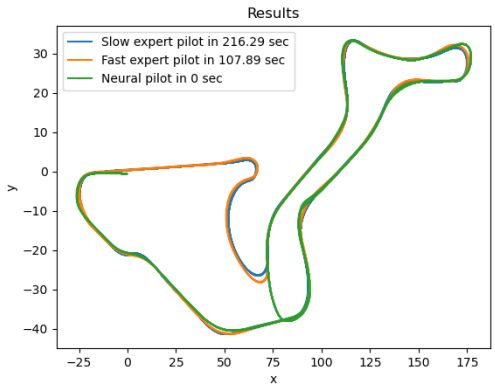


Figura 5.22: Resultado del *dataset* balanceado

Finalmente se consiguió el funcionamiento esperado del dron con el dataset ampliado semibalanceado explicado en el capítulo 5.3. Como podemos ver en la figura 5.23, los resultados fueron peores a los del piloto experto, con un error medio mayor de 0.53 m y con una diferencia de tiempo de 3.52 segundos. Pero a su vez se demostró que el piloto neuronal seguía por muy poco al piloto algorítmico experto, por lo que este resultado demostró que efectivamente esta aplicación se puede realizar con este sistema.

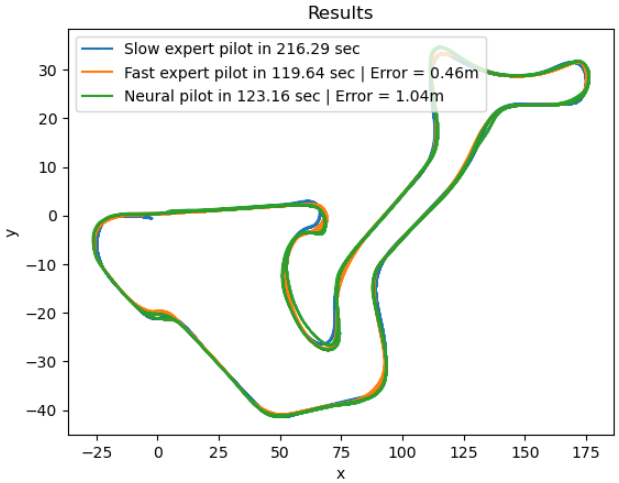


Figura 5.23: Resultados en el circuito de test

En el circuito de Montmeló (fig. 5.24), se observó que a pesar de tardar un segundo más que el piloto experto, la red neuronal fue capaz de seguir el circuito con una mayor precisión. Esto indica claramente que la red está aprendiendo eficazmente del piloto experto.

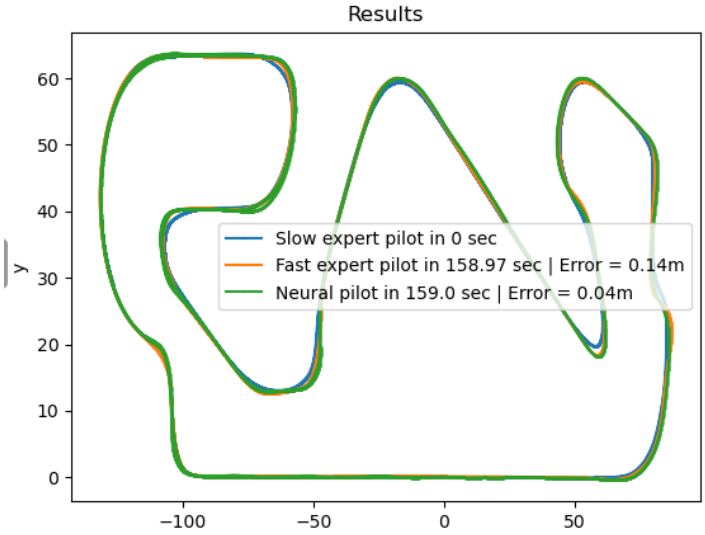


Figura 5.24: Resultados en el circuito de Montmeló

Para tener un análisis completo de que pesos de la red son los que mejor se ajustan para realizar la aplicación de manera adecuada, se creó un script capaz de probar las distintas configuraciones y comprobarlas. Este script se ejecutará después del entrenamiento y ejecutará en el circuito de tests cada red. Una vez terminado el circuito se guardará una imagen ilustrando el contenido mostrando el error medio y el tiempo de vuelta.

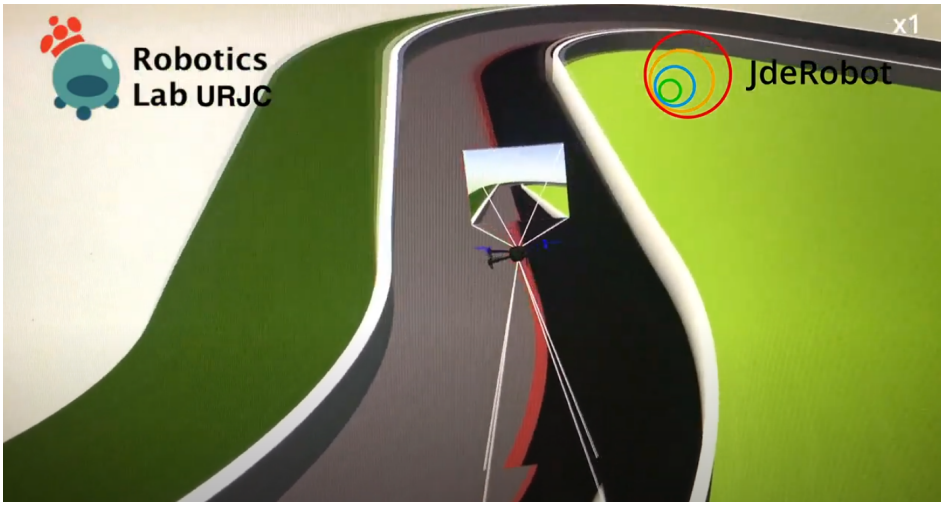


Figura 5.25: Validación experimental

En el siguiente vídeo [2] se puede ver el funcionamiento final del dron entrenado con el dataset semibalanceado(5.3) en el circuito de prueba. En este vídeo, se observa cómo el dron supera varias curvas cerradas, mantiene la estabilidad durante las rectas y ajusta su velocidad en función de la situación en la que se encuentra. Este comportamiento indica un buen desempeño en el entorno de prueba y demuestra la efectividad de la aplicación bajo las condiciones evaluadas.

5.4.2. Análisis de coste computacional

Analizando las frecuencias a las que itera la red, se descubrió un característica de este sistema, su bajo costo computacional. Como se observa en la gráfica 5.26, la frecuencia de iteración del piloto experto algorítmico alcanza un promedio de 125 Hz, mientras que la gráfica 5.27 del piloto neuronal muestra una frecuencia que ronda los 250 Hz, marcando un pico superior al anterior. Esto indica que, con la misma capacidad computacional, la red neuronal puede iterar significativamente más rápido que el piloto experto.

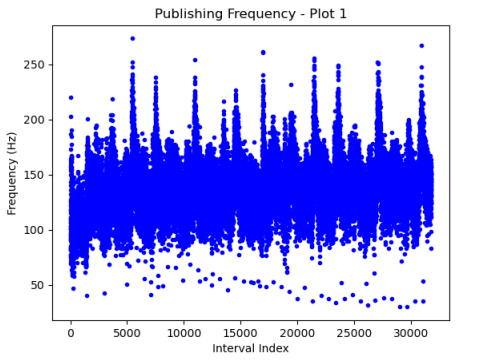


Figura 5.26: Freq. piloto experto

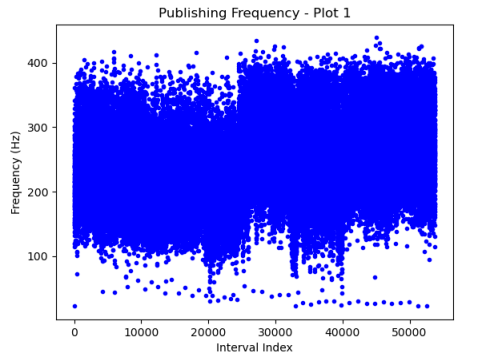


Figura 5.27: Freq. piloto neuronal

[2] https://www.youtube.com/watch?v=jJ4Xdin1gg4

6. Aplicación cruza ventanas

La segunda aplicación consistirá en que el dron sea capaz de seguir un circuito tridimensional cruzando ventanas. Estos elementos son muy utilizados en carreras de drones y suponen un desafío más complejo para su percepción que la aplicación del sigue líneas. En primera instancia, la aplicación se realizará con todas las ventanas a la misma altura utilizando *PilotNet*, para posteriormente tenerlas a diferentes alturas y con *DeepPilot* en el eje Z. Se seguirá un procedimiento similar al de la aplicación anterior, cambiando aquellas partes donde difieran ambas aplicaciones, mejorando así el rendimiento y el análisis objetivo de cada aplicación.

6.1. Generación de escenarios

A diferencia de la aplicación anterior, en esta no se contaba con escenarios 3D predefinidos. Por lo tanto, se crearon escenarios para la obtención del *dataset*. Se siguieron tres procedimientos:

En primer lugar, se generaron escenarios de forma manual, añadiendo las ventanas manualmente. Esto permitió la creación de circuitos complejos y situaciones poco comunes, enriqueciendo así la variedad del *dataset*. Esta metodología se utilizó en escenarios de testeo final y en escenarios de entrenamiento como el que se ve en la imagen 6.1 donde se muestra una combinación de figuras geométricas y ventanas colocadas con el fin de añadir variedad al *dataset*.

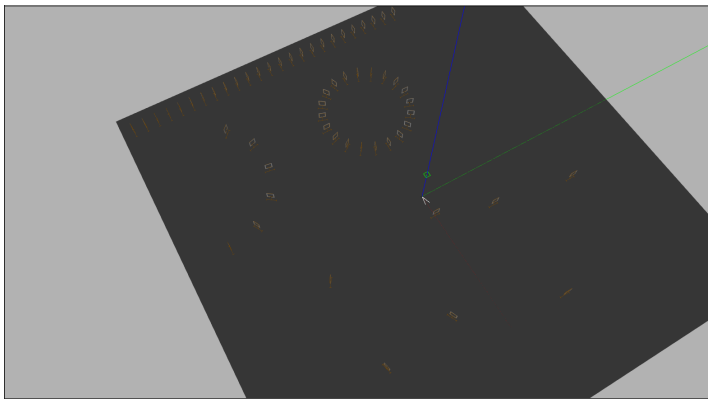


Figura 6.1: Escenario variado

Ya que esta tarea requería bastante tiempo, se generaron automáticamente circuitos con formas geométricas simples, como rectas, circunferencias y elipses como se puede ver en la imagen 6.2, para que el sistema pudiera aprender patrones básicos. Este enfoque complementó la diversidad de los escenarios manuales, asegurando que el sistema adquiriera tanto habilidades básicas como la capacidad de enfrentarse a escenarios complejos.

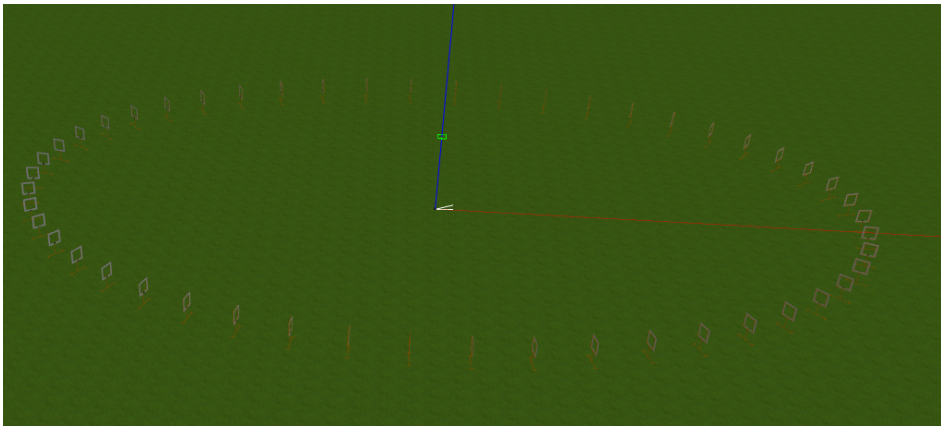


Figura 6.2: Escenario con forma de óvalo

Para añadir mas complejidad se creó un script de Python capaz de generar mundos aleatorios con n ventanas para poder generar un mayor número de escenarios y enriquecer el *dataset*.

Lo primero que se hizo en este generador de escenarios fue elegir aleatoriamente entre uno de los dos escenarios que Aerostack proporciona con Gazebo: *empty* o *grass*.

Dado un determinado número de ventanas que se generarán en el circuito, se creará el archivo `external_objects.yaml` donde se generarán las transformaciones *(TFs)* de cada ventana, nombrando genéricamente a cada una como **gate_n**.

El siguiente paso fue crear las ventanas de tal manera que el dron operara constantemente sin perder de vista la siguiente ventana, facilitando así la automatización en la grabación de *datasets*. Dado un número n_{gates}, un bucle iterativo generará cada ventana con referencia a la anterior. Primero se definirá el ángulo de ambas ventanas aleatoriamente, pero dentro de un umbral calculado empíricamente para que el dron no pierda de vista la ventana $n+1$ tras atravesar la ventana n. La ventana tendrá la posición:

$$x = x_{\text{prev}} - \text{distancia} \times \cos(\text{pos\_angle})$$
$$y = y_{\text{prev}} - \text{distancia} \times \sin(\text{pos\_angle})$$

Donde:

- x_{prev} y y_{prev} son las coordenadas x e y de la ventana anterior respectivamente.
- pos_angle es el ángulo seleccionado aleatoriamente para la nueva ventana.

Dado un conjunto de ventanas ubicadas en coordenadas (x_i, y_i)), la orientación de cada ventana se ajusta en función de los ángulos formados entre la ventana actual y sus ventanas vecinas, es decir, la ventana anterior y la ventana siguiente. Este ajuste se realiza mediante el cálculo del ángulo promedio entre estos segmentos de ventana.

$$\theta = \frac{\arctan 2\,(x_0 - x_{-1}, y_= - y_{-1}) + \arctan 2\,(x_1 - x_0, y_1 - y_0)}{2}$$

Para asignar alturas a las ventanas, se utiliza un valor aleatorio dentro de un intervalo definido. En este caso, la altura de cada ventana se determina mediante una distribución uniforme entre -1 y 4, el resultado final de este tipo de circuitos se puede ver en la imagen 6.3.

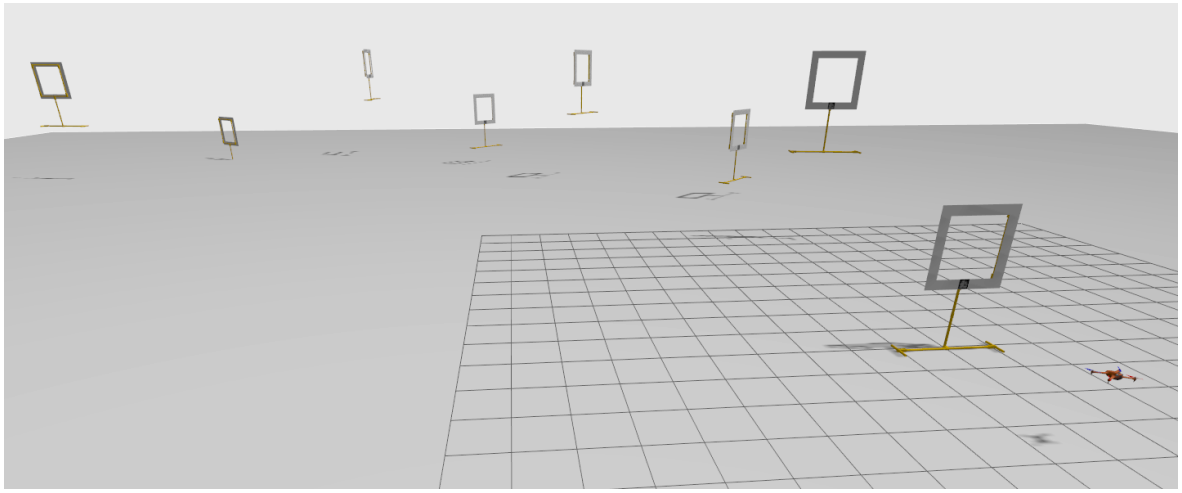


Figura 6.3: Escenario pseudoaleatorio

6.2. Piloto teleoperado

Una de las principales diferencias entre la aplicación de sigue líneas y la aplicación del cruza ventanas se basa en el método por el cual se obtiene el *dataset*.

Se llegó a la conclusión de usar un piloto teleoperado manualmente ya que permitiría analizar la capacidad de imitación de este sistema respecto a un piloto humano. Las acciones de un piloto humano, a diferencia de un piloto programado, no suelen estar guiadas por patrones estrictos y tienden más al error, pero a su vez poseen una mayor capacidad de adaptación ya que el comportamiento humano en el control de un dron incluye variaciones y correcciones constantes que un sistema automatizado puede no considerar. Además, esto muestra la eficacia del sistema en condiciones más cercanas a las del mundo real, donde las decisiones no siempre siguen un patrón predefinido y requieren ajustes dinámicos.

Como estación de control para el control del dron, se utilizó un mando de PS4[1]. Utilizando el driver `ds4drv`[2] junto al paquete ROS `ds4_driver`[3]. Utilizando estos paquetes se consiguió obtener los datos deseados del mando Bluetooth, dejando los controles de la siguiente manera: ver figura 6.4.

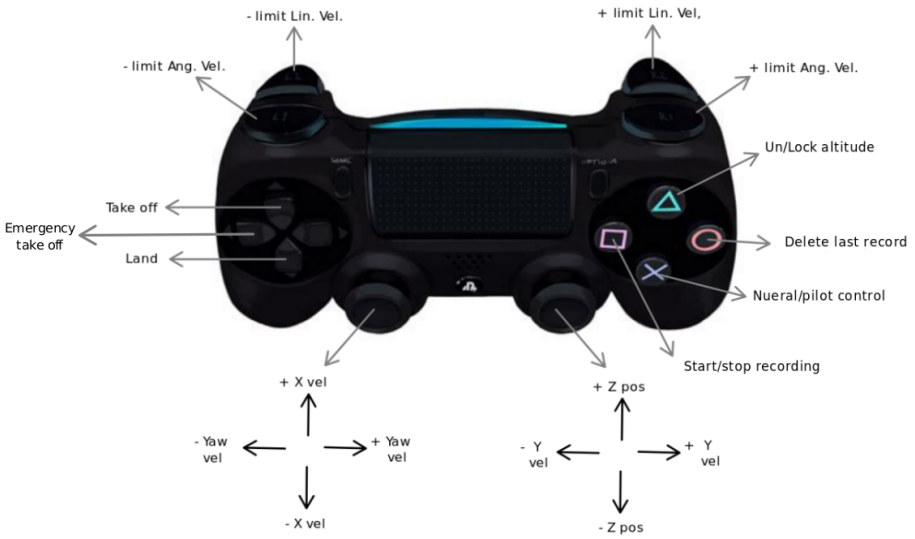


Figura 6.4: Distribución de controles del mando

6.2.1. Control general

Para el control general del dron, se eligió la cruceta como despegue y aterrizaje. La tecla superior se asignó para el despegue, la inferior para el aterrizaje y la izquierda para el aterrizaje de emergencia.

Por otro lado, se eligió la tecla X para que el piloto neuronal tomara el control

[1] https://www.playstation.com/en-us/accessories/dualshock-4-wireless-controller/
[2] https://github.com/chrippa/ds4drv
[3] https://github.com/naoki-mizuno/ds4_driver

total sobre el sistema. Para que el piloto recupere su control, simplemente deberá presionar el mismo botón, y la red neuronal dejará de inferir velocidades.

6.2.2. Control de velocidad

La velocidad lineal del dron se ajustará con el joystick izquierdo en su componente vertical y la componente de giro en *yaw* se ajustará en el eje horizontal, asimilándose al control en videojuegos y permitiendo al teleoperador un mejor control.

Adicionalmente, en estas dos componentes se podrán variar los límites de velocidad con los gatillos. El gatillo L1 se encargará de reducir la velocidad angular y el L2 se encargará de reducir la velocidad lineal. Asimismo, los gatillos R1 y R2 aumentarán respectivamente el límite de la velocidad angular y lineal, permitiendo al operador tener un mayor control sobre el dron.

Velocidad angular *(Yaw)*

En la gráfica inferior 6.5 podemos ver cómo la velocidad angular comandada, representada en azul, es totalmente proporcional a la entrada del joystick (representada en rojo). Esto se realizó de esta manera ya que se desea una respuesta rápida en la velocidad angular. Si nos fijamos a partir de la muestra 4000, la velocidad sube a escalones. Esto muestra que el piloto está variando la velocidad angular máxima, permitiendo ajustar el límite del PD.

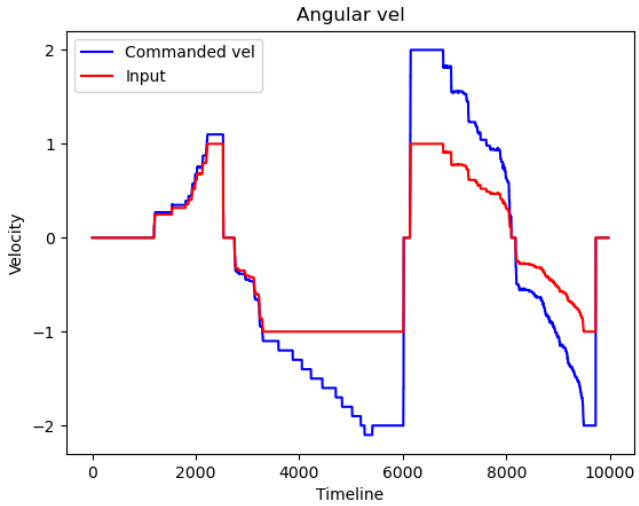


Figura 6.5: Control de velocidad angular

Velocidad lineal

Respecto a la velocidad lineal, se aplicó la técnica de un buffer de n medidas, asignando la media de estas medidas como velocidad de salida, para evitar movimientos exageradamente bruscos en el dron. El movimiento del joystick está representado en color rojo y la velocidad comandada en azul y como se puede ver en la gráfica 6.6, podemos notar que es ligeramente más suave y tiene menos saltos gracias a la implementación del buffer. Este controlador añade un pequeño *offset* en la práctica, sin embargo esto produce que el dron ladee menos la cámara y tenga una visión mas estable, comportamiento crucial para el correcto funcionamiento el sistema.

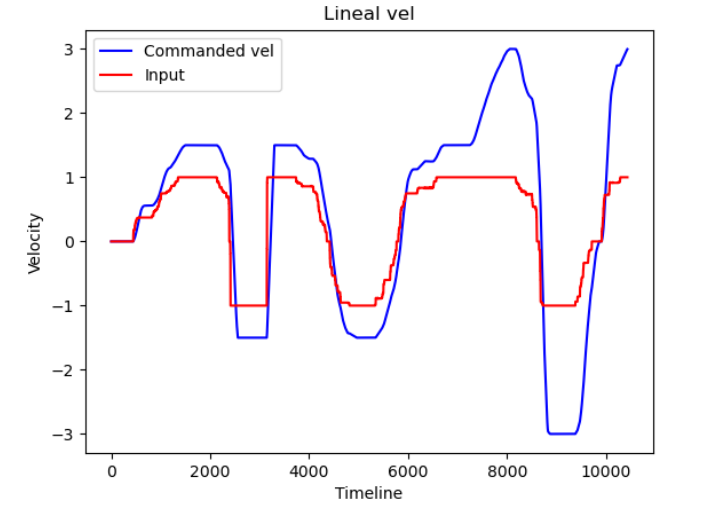


Figura 6.6: Control de velocidad lineal

Velocidad lateral

Con el joystick derecho en el eje horizontal se controla la velocidad en *roll*, aunque este no se utiliza para la aplicación, y en el eje vertical se controla la velocidad en Z, se implementó también el movimiento en este eje para posibles futuras aplicaciones.

Posición de altura

A diferencia de los controles en velocidad lineal y angular, que son respecto a la velocidad, el control en altitud estuvo enfocado en la posición.

La entrada del joystick, cambia la posición deseada. A partir de esta posición se calculará el error entre la posición actual y la comandada, controlando este error con un PD adicional para dar una mayor reactividad y suavidad al movimiento. En esta componente, un controlador proporcional causaba rebotes en el control, por lo que fue necesario amortiguar estos mismos con un controlador PD, el resultado se puede ver en la gráfica 6.7. En caso de querer dejar la altura fija, presionando el botón triángulo, el dron se moverá en un plano 2D.

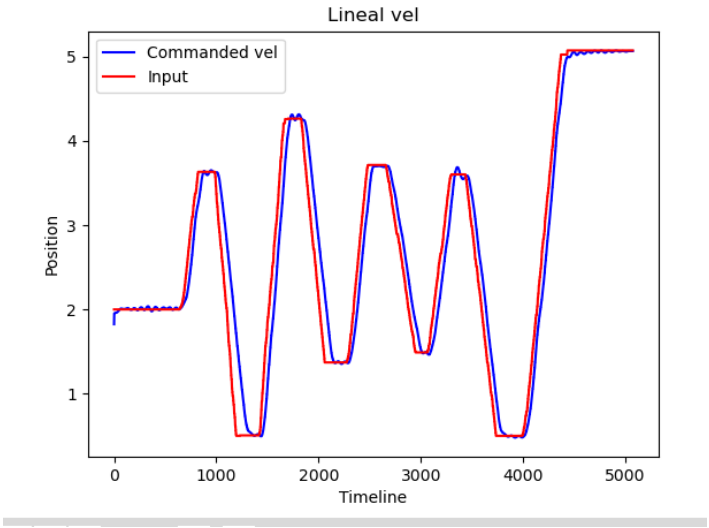


Figura 6.7: Control de posición en altitud

6.2.3. Grabación de rosbags

Para generar el *dataset*, necesitamos grabar las velocidades que se envían al dron. Grabar desde el despegue hasta el aterrizaje del dron podría ser perjudicial para la red, ya que habrá movimientos del piloto que pueden no ser claros para esta misma, como ir a una zona concreta teniendo un conocimiento del escenario completo. Para grabar el *dataset*, se utilizó el botón cuadrado para iniciar o parar la grabación y el botón círculo para borrar la última grabación. De esta manera, bajo el criterio del piloto, se pueden borrar ejecuciones o movimientos que hayan fallado, dejando en el *dataset* solo aquellas muestras que efectivamente sirvan para el entrenamiento de la red.

6.3. Piloto experto algorítmico

Para complementar al piloto teleoperado y enriquecer el *dataset*, se modificó el piloto experto del repositorio `project_crazyflie_gates`[2]. Este repositorio proporciona un ejemplo de uso de `Aerostack2`, donde un dron describe una trayectoria circular atravesando dos ventanas paralelas. El dron atraviesa estas ventanas utilizando el paquete `as2_external_object_to_tf`, el cual proporciona las transformaciones de las ventanas y permite posicionar al dron en el centro de las mismas.

Basándose en este ejemplo, se adaptó el dron para incluir una cámara y operar de manera similar a la simulación en Gazebo 11. Este piloto se desarrolló con Gazebo Fortress, lo cual no supone un problema, ya que al generalizar el *dataset*, se obtienen únicamente imágenes y velocidades, permitiendo el uso simultáneo de ambas opciones.

Después de esto, se realizaron ajustes en el `motion_controller_plugin` para que el dron tuviera una respuesta más reactiva.

Para grabar el *dataset*, se creó un nodo capaz de traducir las velocidades comandadas por el control de posición al control implementado en la aplicación anterior. Además, las velocidades solo se grabarán cuando superen un umbral determinado por una velocidad mínima, ya que al analizar el comportamiento del control por posición, estas muestras podrían afectar negativamente el rendimiento de la red.

Los scripts de automatización para esta aplicación consisten en un script que ejecuta en segundo plano la simulación, el sistema de Aerostack y el grabador de ROSbags, dejando en primer plano la misión del dron. Una vez finalizada la misión, se espera un tiempo determinado y se inicia una nueva ejecución con un nuevo circuito aleatorio.

Desafortunadamente, al utilizar otro modelo de dron, aunque los datos fueran revisados y parecieran similares, se descubrió que las muestras generadas con este piloto perjudicaban el funcionamiento del sistema tras combinarlas con el *dataset* recogido con el piloto teleoperado. Por lo tanto, se llegó a la conclusión de utilizar únicamente el piloto teleoperado para entrenar la red neuronal.

[2] https://github.com/Adrimapo/project_crazyflie_gates

6.4. Manejo y conjunto de datos

Se siguió el mismo proceso de estandarización que en la aplicación del sigue líneas, mencionado en el apartado 5.2. Respecto al entrenamiento de *PilotNet*, también se siguió el mismo procedimiento. Sin embargo, el uso de *DeepPilot* y el hecho de que el *dataset* fuera grabado por un piloto humano conllevó la necesidad de utilizar e implementar herramientas adicionales.

6.4.1. Selección de tomas

Se decidió añadir una verificación manual al *dataset* creado, tanto para detectar muestras no deseadas como para identificar posibles fallos del piloto, ya que al tener factor humano, el *dataset* es más susceptible a errores. Una vez que el *dataset* se haya transformado a un formato estándar, utilizando el visualizador de imágenes de Linux, podemos visualizar las fotos como si se tratara de un vídeo. Esto nos permite eliminar (tecla Supr.), a la misma velocidad con la que avanza la sucesión de fotos, cualquier muestra que no haya funcionado correctamente.

Posteriormente, utilizando un script de Bash, se eliminarán las velocidades múltiples asociadas a cada imagen dejando solo una, de esta manera se mantendrá la misma longitud en las muestras de imágenes y de velocidades, sin borrar información útil en ninguna de las dos.

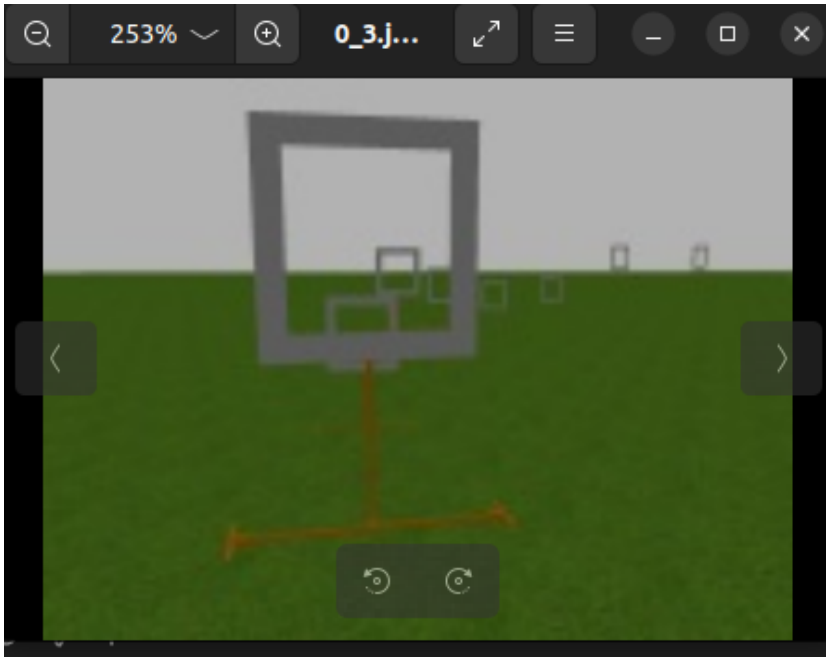


Figura 6.8: Visualización de imágenes *dataset*

6.4.2. *Dataset* con ventanas a altura constante

En primera estancia, se realizó la aplicación con altura fija, de esta manera se pudo comprobar si realmente *PilotNet* prestaba las condiciones necesarias para poder realizar la aplicación con éxito.

Primera aproximación

El primer *dataset* generado consistió en 5311 muestras. Con el aumento de datos, el volumen quedó en 9908. Tras el entrenamiento, se comprobó que no era un *dataset* suficiente, ya que no tenía ni las suficientes muestras ni la suficiente variedad (ver Figura 6.9). Al igual que en la aplicación anterior necesitamos encontrar un equilibrio entre situaciones poco comunes y situaciones comunes. Analizando las gráficas inferiores donde se muestra la distribución de velocidades, podemos comprobar que hay muy pocas situaciones contempladas y en las gráficas superiores donde se muestran el número de casos de las mismas vemos que las velocidades lineales bajas con velocidad angular negativa son las más comunes, esto es debido a que se entrenó el dron para que al no detectar puertas guiara en este sentido buscando una nueva.

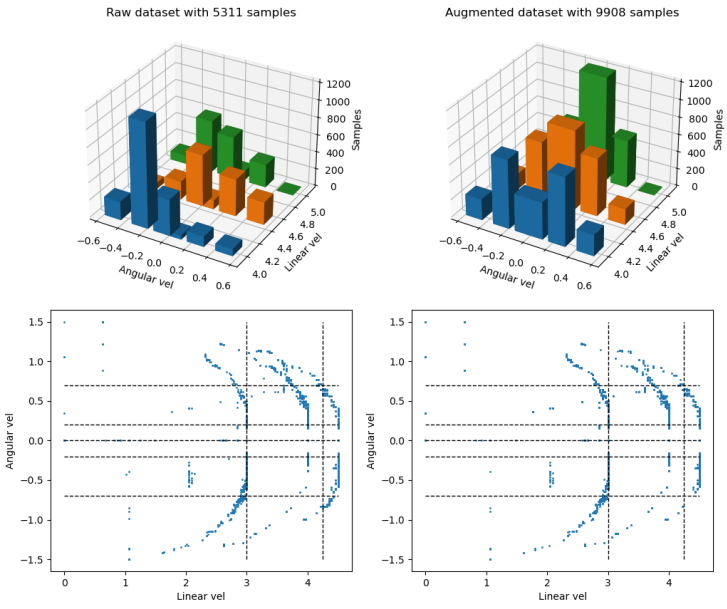


Figura 6.9: Primer *dataset*

Aumentado de *dataset* mixto

Posteriormente, se hicieron dos ampliaciones más del *dataset*: combinando el piloto experto de Aerostack y el piloto teleoperado (6.10). Sin embargo, aun con un mayor número de muestras (6.11), el resultado fue significativamente peor. Esto se debió a las diferencias entre el dron usado en la simulación y el pilotado neuronalmente, y el piloto experto de Aerostack, ya que se usaban versiones distintas de Gazebo. En este caso podemos ver los patrones conseguidos en el ejercicio anterior con el piloto teleoperado. También podemos ver una gran cantidad de velocidades generadas por el piloto algorítmico, estas están en el centro de la gráfica y muestran velocidades lineales medias-altas con una una velocidad angular baja.

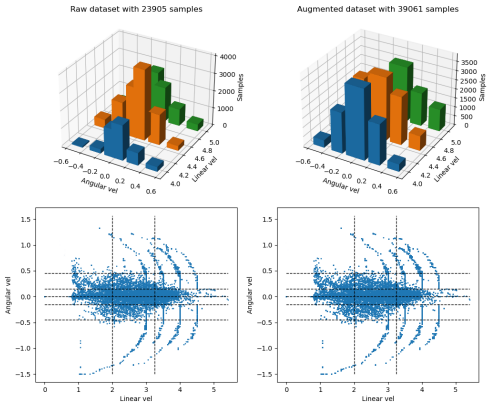


Figura 6.10: *Dataset* combinado 1

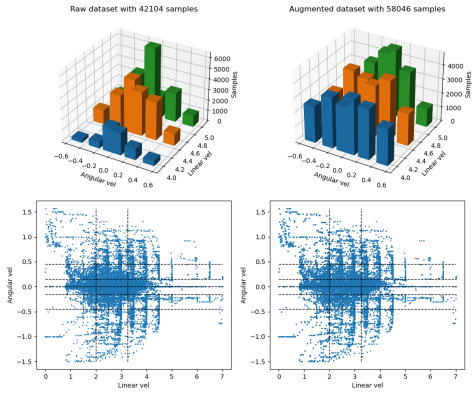


Figura 6.11: *Dataset* combinado 2

Dataset con piloto teleoperado

Por último, se decidió ampliar el *dataset* exclusivamente con el piloto teleoperado y eliminar las muestras tomadas con el piloto experto. Tras esto, el dron consiguió cruzar el circuito de test logrando comportamientos reactivos, pero con oscilaciones. El procedimiento para solventar este problema fue rebalancear el *dataset* de nuevo para dejar un menor número de muestras donde las velocidades angulares fueran altas, de modo que el dron se guiara por los casos donde las velocidades angulares fueran más bajas, los cuales son los más comunes. En este dataset hay una mayor variedad de velocidades y con un mejor equilibrio en la distribución de cada etiqueta. Por estos motivos este fue el conjunto de datos que mejor resultados dio.

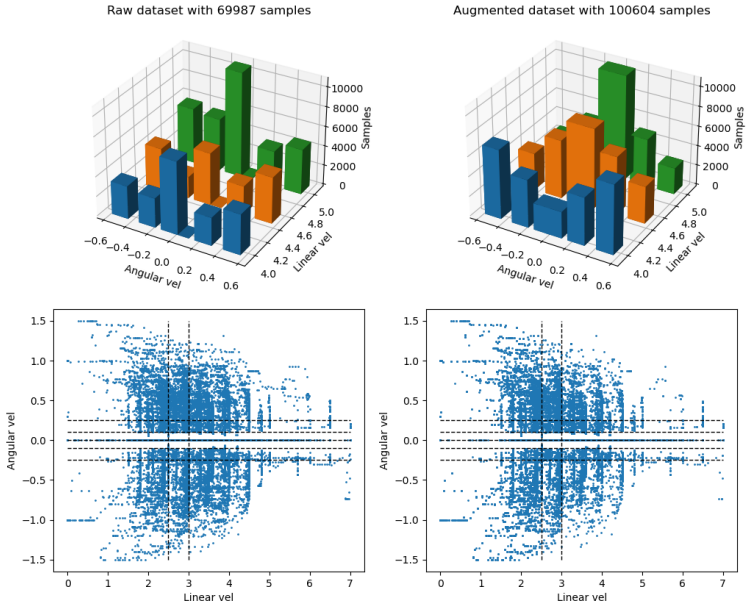


Figura 6.12: *Dataset* final

Dataset con imágenes procesadas

Con el *dataset* anterior se realizó una prueba adicional que consistió en aplicar un preprocesamiento a la imagen de entrada de la red. Este preprocesamiento consistió en un filtro de color junto con una dilatación, técnica similar al piloto experto de la aplicación de seguimiento de líneas (ver sección 5.1).

No se logró que el dron tuviera la robustez suficiente para completar el circuito sin chocarse. Sin embargo, un enfoque interesante que cabe destacar es que a medida que se aumentaba el tamaño de las ventanas mediante la dilatación, el dron chocaba más veces. Esto podría deberse a que el dron, al ser entrenado con ventanas más grandes, considera menos arriesgado pasar cerca de la zona descrita por la ventana. En contraste, cuando se reduce demasiado el tamaño de las ventanas, el dron opta por esquivar completamente la ventana buscando áreas donde no haya obstáculos, a veces pasando por fuera. El resultado más favorable se encontró en un compromiso entre ambas anchuras.

Figura 6.13: Ventanas dilatadas

Figura 6.14: Ventanas estándar

6.4.3. *Dataset* con ventanas a alturas variadas

Como luego se detallará, *DeepPilot* solo será utilizado para la componente vertical por lo que se grabó un *dataset* complementario mas centrado en la variedad de muestras con cambios de altitud.

Primer *dataset*

El primer *dataset* grabado constó de 51.068 muestras, pero este presentó un problema. No había suficientes casos donde el dron mantuviera una altura constante por lo que este tendía a ir hacia arriba. Por este motivo, en el siguiente *dataset* se añadieron muestras usadas en el entrenamiento de *PilotNet*, donde la velocidad vertical no variara.

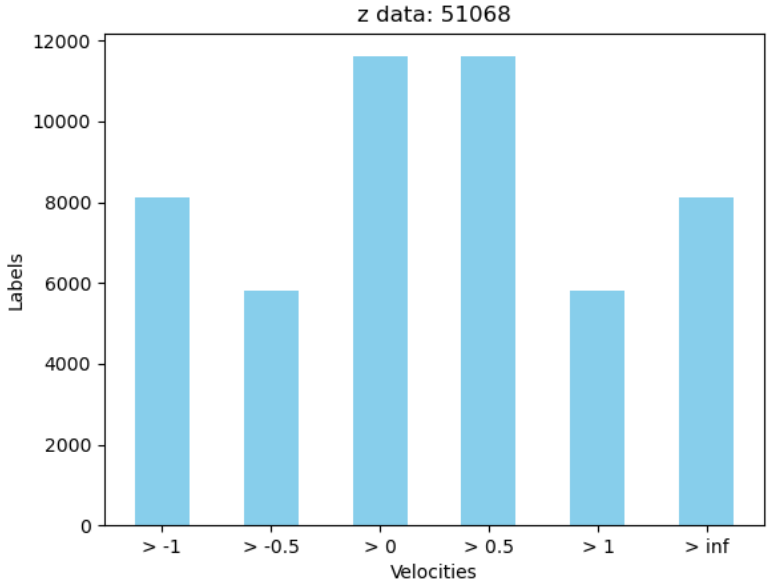


Figura 6.15: Primer *dataset* DeepPilot

Segundo *dataset*

Tras varias ampliaciones de *dataset*, se obtuvo un volumen de 42319 muestras en crudo 6.16 con un total de 76478 muestras con el *dataset* balanceado 6.17.

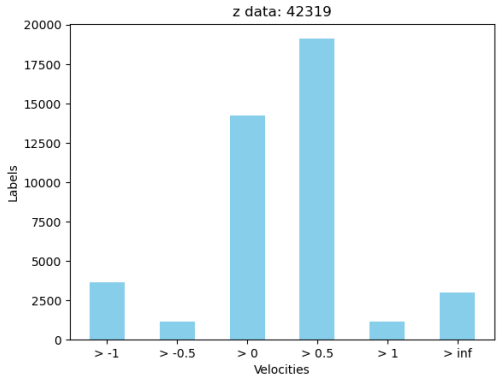


Figura 6.16: *Dataset* crudo

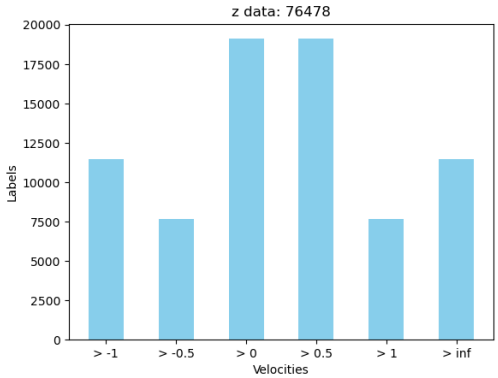


Figura 6.17: *Dataset* balanceado

En esta prueba los resultados se correspondieron con el rendimiento esperado, el dron fue capaz de pilotar eficazmente en el eje z, tanto con el piloto manual como con el piloto experto dejando un buen resultado en la combinación del uso de ambas redes neuronales.

6.5. Modelo neuronal y entrenamiento

6.5.1. Red *PilotNet*

Para la implementación del modelo de red, concretamente *PilotNet* se reutilizó la infraestructura de la aplicación anterior, mencionado en el apartado 5.3. De esta manera podremos comprobar si realmente este modelo es capaz de realizar ambas aplicaciones. Las únicas diferencias respecto a la aplicación anterior consisten en el cambio de un piloto experto a un piloto teleoperado, donde un operador humano es el que genera el *dataset* y que al tener ventanas a distinta altitud, el control en z dependerá de otra red neuronal, *DeepPilot*

6.5.2. Red *DeepPilot*

Para la implementación de *DeepPilot*, tomamos como referencia el Trabajo de Fin de Grado de Vanessa Martínez, *Conducción autónoma de un vehículo en simulador mediante aprendizaje extremo a extremo basado en visión* [4], y el artículo de Leticia Oyuki, *DeepPilot: A CNN for Autonomous Drone Racing* [3].

Se reutilizará la infraestructura utilizada con *PilotNet*, pero con las modificaciones necesarias requeridas por la red. En el caso de *DeepPilot*, esta red solo tendrá una salida en vez de dos como *PilotNet*. El tamaño de la imagen de entrada de la red será de $64 \times 64 \times 3$, manteniendo la imagen en formato RGB, dando una única salida que será la componente en z del dron.

6.5.3. Criterio de pérdida

Al igual que en la aplicación del sigue líneas con *PilotNet* 5.3, se utilizó `nn.MSELoss()` como criterio de pérdida. La función de pérdida MSE (Mean Squared Error) es ampliamente utilizada en problemas de regresión debido a varias ventajas:

- **Simplicidad**: Calcula el promedio de los cuadrados de las diferencias entre las predicciones y los valores reales, proporcionando una medida clara de cuán cerca están las predicciones del modelo de los valores reales.
- **Diferenciabilidad**: La función MSE es diferenciable, lo que es crucial para el proceso de optimización basado en gradientes. Esto permite que los

algoritmos de optimización ajusten los pesos de la red neuronal de manera eficiente.

- **Penalización de errores grandes**: MSE penaliza más los errores grandes debido al término cuadrático.

6.5.4. Función de optimización

Para la optimización, se utilizó `torch.optim.Adam`. Adam (Adaptive Moment Estimation) es un algoritmo de optimización que combina las ventajas de dos otros métodos populares: AdaGrad y RMSProp. Adam es ampliamente preferido en el entrenamiento de redes neuronales profundas debido a sus múltiples ventajas:

- **Adaptabilidad**: Adam ajusta las tasas de aprendizaje de cada parámetro de manera adaptativa, utilizando estimaciones de momentos de primer y segundo orden.
- **Convergencia rápida**: Gracias a sus mecanismos de corrección de sesgo, Adam puede converger más rápidamente que otros optimizadores estándar, lo cual es útil para entrenar modelos grandes y complejos.
- **Robustez**: Adam es robusto frente a la configuración de hiperparámetros, lo que facilita su uso en diversas aplicaciones sin necesidad de ajustes finos intensivos.
- **Eficiencia computacional**: Adam es eficiente en términos de memoria y tiempo de cómputo, lo que es crucial cuando se trabaja con grandes conjuntos de datos y modelos extensos.

6.6. Evaluación

6.7. Validación experimental y evaluación

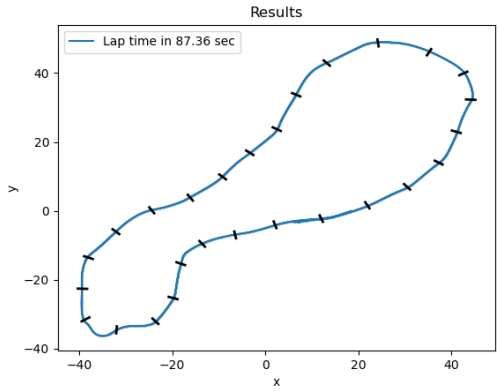


Figura 6.18: Resultado piloto neuronal

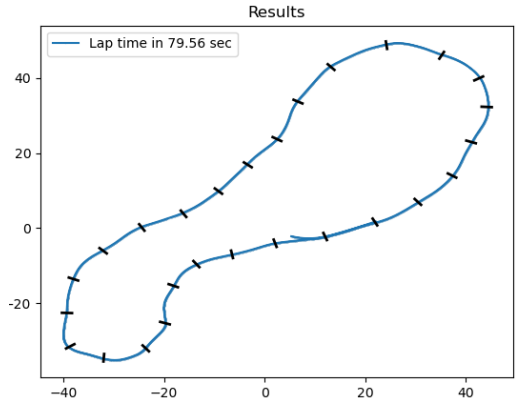


Figura 6.19: Resultado piloto teleoperado

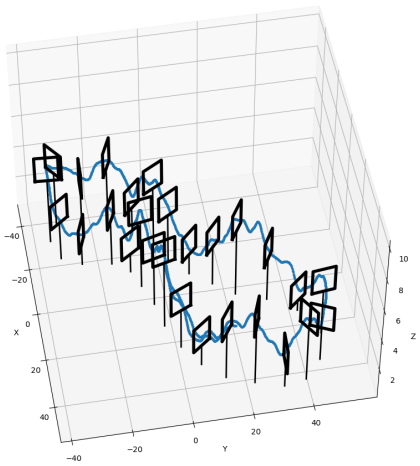


Figura 6.20: Gráfica 3D del recorrido 1

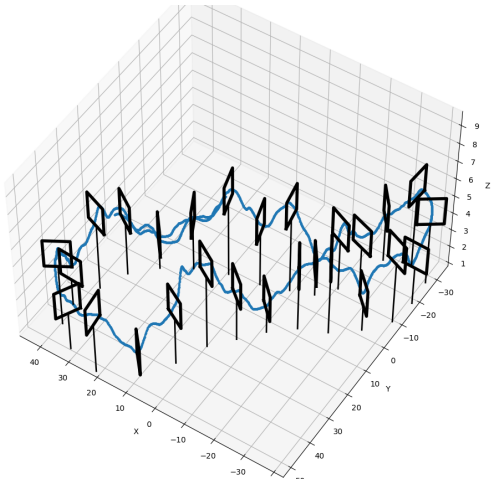


Figura 6.21: Gráfica 3D del recorrido 2

Finalmente combinando los modelos entrenados con los *datasets* mencionados anteriormente, en caso de *PilotNet* el *dataset* mencionado en el punto 6.4.2 y en el caso de *DeepPilot* 6.4.3 se consiguió completar la aplicación satisfactoriamente. Cabe destacar la capacidad que ha tenido *PilotNet* en la figura 6.18 para imitar el comportamiento humano en la figura 6.19 al realizar el circuito. Aunque el sistema no fue específicamente entrenado con este circuito particular, podemos ver en el

siguiente enlace un vídeo de la ejecución [3], que se ha logrado obtener buenos resultados. Esto no solo demuestra la capacidad de aprendizaje y adaptación de la red neuronal, sino también su habilidad para generalizar a nuevas situaciones tanto con altitud constante como con altitud variable.

Respecto a la ejecución con altitud variable, como podemos ver en las imágenes 6.20 y 6.21 o en el siguiente vídeo [4]. Se comprobó que en la práctica el control combinado de ambas redes neuronales da el resultado esperado y permite que el dron realice la aplicación completa llegando a imitar el comportamiento del piloto humano.

6.7.1. Análisis de coste computacional

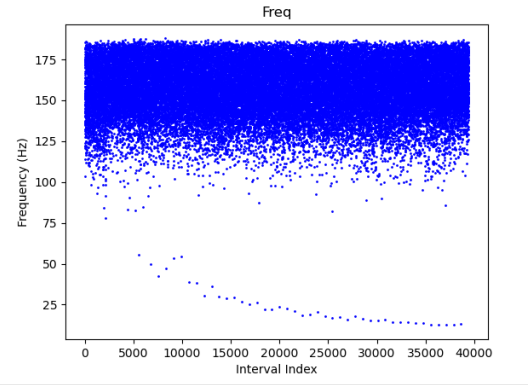


Figura 6.22: Frecuencia piloto teleoperado

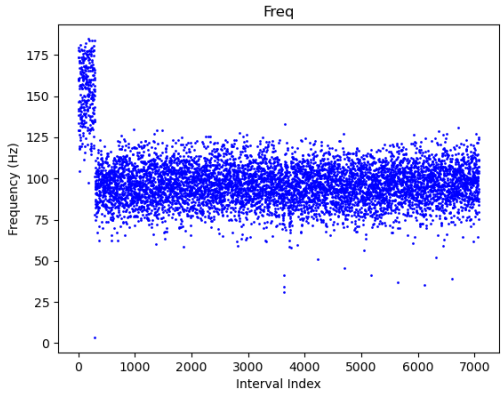


Figura 6.23: Frecuencia piloto neuronal

Ya que esta aplicación se desarrolló sin piloto experto, el análisis de coste computacional se enfocará de una manera distinta. El piloto teleoperado no cuenta con preprocesamiento de imagen ni operaciones pesadas, por lo que consideraremos como frecuencia máxima del sistema la frecuencia a la que itera el piloto teleoperado.

Cabe destacar que en esta aplicación se limitó la frecuencia de iteración. Se estableció un límite máximo de 200 Hz para la obtención de la imagen, lo cual evita que se publiquen velocidades más rápidamente de lo que se reciben imágenes de entrada, de esta manera no se infiere varias veces sobre la misma imagen.

[3] https://youtu.be/l-WyA2C-4I4
[4] https://youtu.be/Q1zBNXdW7Ns

Considerando que cada iteración con el piloto teleoperado 6.22 ocurre cada $\frac{1}{150}$ segundos (0,0067 segundos por iteración) y con el piloto neuronal basado en *PilotNet* (fig. 6.23) cada $\frac{1}{100}$ segundos (0,01 segundos por iteración), la diferencia de tiempo por iteración es de 0,0033 segundos. Esto implica un procesamiento bastante eficiente y un tiempo de inferencia bastante bajo, demostrando que el piloto neuronal puede ser mas eficiente computacionalmente.

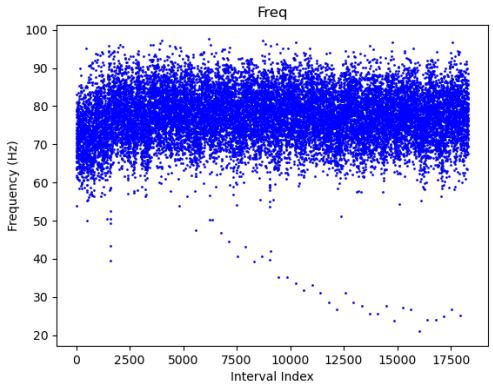


Figura 6.24: Frecuencia *DeepPilot*

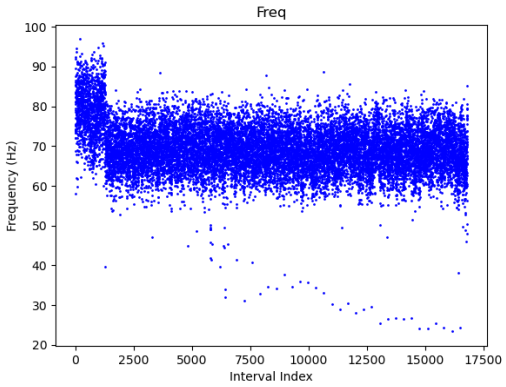


Figura 6.25: Frecuencia *DeepPilot* + *PilotNet*

Podemos ver en la gráfica 6.24 como *DeepPilot* es una red con un mayor coste computacional, esto es debido a que posee una arquitectura más compleja, la frecuencia media de inferencia torna una media de 82Hz lo que es un equivalente a 1 iteración cada 0.012 segundos. Respecto a ambas juntas, como se puede ver en la gráfica 6.25, la inferencia total torna los 66 Hz lo que equivale a una iteración cada 0.015 segundos, restando ambas cantidades, calcularemos que *PilotNet* tiene un tiempo de inferencia de 0.003 segundos, como en la aplicación anterior.

Comparando *DeepPilot* con los resultados anteriores tiene un tiempo de inferencia de 0.009 segundos, el triple que *PilotNet*. Si utilizáramos *DeepPilot* para las 3 componentes tendríamos una interacción cada 0.028 segundos, lo que equivale a 35 Hz. Esto nos muestra que *DeepPilot* es un modelo ligero que nos puede brindar muy buenos resultados.

7. Conclusiones

En este capítulo se expondrán las conclusiones obtenidas de este trabajo y las líneas futuras del mismo.

7.1. Objetivos conseguidos

A lo largo de esta memoria se ha demostrado que una red neuronal es capaz de pilotar un dron de manera eficiente, imitando tanto el comportamiento de un piloto humano como el de un piloto experto algorítmico. Para alcanzar estos objetivos, se han utilizado diversas herramientas que permitieron desarrollar la aplicación con un nivel de abstracción adecuado, facilitando la implementación y validación de la red. El objetivo principal de este trabajo es mostrar que un piloto controlado por una red neuronal puede producir resultados comparables o superiores a los de un piloto humano o programado de manera clásica en varias aplicaciones mediante el uso de *imitation learning*. Tanto en la aplicación de seguimiento de líneas como en la de cruce de ventanas, se ha validado que redes neuronales como *PilotNet* o *DeepPilot* puede aprender eficazmente de ambos tipos de comportamiento. Debido a las similitudes entre ambas aplicaciones y su desarrollo se obtuvieron las siguientes conclusiones conjuntas:

- **Tratamiento del dataset** Una de las conclusiones más importantes después de trabajar en un proyecto de *machine learning* es que sin un dataset de calidad, la aplicación no va a funcionar de manera adecuada. La mayor parte del tiempo se dedicó a probar distintas distribuciones de datasets y ver cómo esto afecta al sistema, por lo que se aprendieron técnicas como el aumentado de datos para llegar a un mejor rendimiento de la aplicación.
- **Entrenamiento de la red neuronal e implementación:** Se aprendió sobre varias redes neuronales, adquiriendo también conocimientos prácticos sobre estas mismas y su entrenamiento. Aunque la solución converja y el error de entrenamiento sea muy bajo, no significa que la aplicación vaya a funcionar correctamente. En otras ocasiones, con cierto error, el dron funciona mucho mejor que con un error más pequeño. Por ello, se creó un script que hiciera varias copias durante el entrenamiento y luego se automatizara la selección del mejor piloto.

- **Evaluación de los resultados:** En esta fase se utilizaron mayoritariamente gráficas para observar los resultados y mejorarlos, permitiendo un desarrollo y evolución correcta de la aplicación y un mayor conocimiento sobre el sistema desarrollado.

Respecto a cada subobjetivo individual se consiguieron las siguientes conclusiones:

7.1.1. Aplicación sigue líneas

La primera fase consintió en el estudio del estado del arte en *imitation learning*. Esto nos permitió tener una visión más clara de cómo realizar el proyecto y aportó varias ideas que se expondrán más adelante en trabajos futuros. Cabe destacar la importancia de los trabajos de fin de grado de otros compañeros, que proporcionaron conocimientos sobre redes neuronales y sistemas aéreos, permitiendo combinar ambos. Posteriormente, en el ámbito de simulación, gracias al equipo de desarrolladores de Aerostack2, quienes a través de varios *issues*, ayudaron en la implementación de los *drivers*, dejando la base de la aplicación preparada. Gracias a esta capa de abstracción, pudimos centrarnos en la aplicación en sí y en el rendimiento de las redes neuronales.

Durante el desarrollo del piloto experto algorítmico, se utilizaron diversos conocimientos obtenidos durante el grado, como técnicas de visión artificial para el reconocimiento de la línea, y técnicas de control para que el dron se mueva de manera suave y sin cambios bruscos.

Respecto a la automatización de la generación del dataset, esta parte del proyecto permitió aplicar conocimientos en Linux. Estos conocimientos no solo fueron útiles para la generación, sino que en el tratamiento del dataset, también fue un elemento crítico tener herramientas que nos permiten agilizar tareas que manualmente son más tediosas y requieren mucho tiempo. La combinación de todas estas herramientas acabó en la finalización con éxito de este subobjetivo.

7.1.2. Aplicación cruza ventanas

Inicialmente, junto al contexto obtenido en el ejercicio anterior, se encontró mas información de esta aplicación, debido a que es más común el cruza ventanas a la

hora de comprobar el rendimiento de este tipo de aplicaciones en drones.

Las mayores diferencias entre esta aplicación y la anterior consistieron en la implementación del control remoto, aquí se consiguió un control preciso con un mando externo permitiendo manejar el dron de una manera cómoda y sencilla. La implementación de *DeepPilot* también demostró la importancia de la generalización del código, ya que al tener programada la infraestructura de *PilotNet* su implementación y uso fue bastante más sencillo.

Al igual que en el subobjetivo anterior, se completaron todas las tareas con éxito y se consiguió implementar un piloto neuronal capaz de atravesar ventanas con diferentes alturas, cumpliendo así todos los objetivos y subobjetivos esperados de este trabajo de fin de grado.

7.2. Trabajos futuros

La infraestructura desarrollada en este proyecto sienta las bases para futuras investigaciones y aplicaciones en el área de pilotaje autónomo de drones mediante redes neuronales. A continuación, se enumeran y explican algunas de las posibles líneas de trabajo futuro:

1. **Implementación y validación en drones reales:**
    - *Descripción:* El primer paso futuro más directo es trasladar y validar el sistema en drones reales. Hasta ahora, todas las pruebas se han realizado en simulación, y es crucial verificar que el sistema se comporta de manera similar en un entorno real.
    - *Objetivo:* Comprobar la robustez y eficacia del sistema en condiciones reales, identificando posibles problemas y ajustando parámetros según sea necesario.
2. **Investigación en otros formatos de imagen:**
    - *Descripción:* Explorar el uso de diferentes formatos y resoluciones de imagen para mejorar el rendimiento del sistema.
    - *Objetivo:* Evaluar si otros formatos de imagen, como imágenes térmicas o cámaras de profundidad, pueden proporcionar información adicional que mejore la precisión y fiabilidad del sistema.
3. **Exploración de otras arquitecturas de redes neuronales:**

- *Descripción:* Investigar y probar otras arquitecturas de redes neuronales que puedan ser más eficientes o adecuadas para esta aplicación.
- *Objetivo:* Identificar y comparar el rendimiento de diferentes arquitecturas de redes neuronales para determinar cuál es la más eficaz para el pilotaje autónomo de drones.

4. **Desarrollo de aplicaciones prácticas:**
    - *Descripción:* Extender el uso del sistema a aplicaciones prácticas más allá de las pruebas iniciales.
    - *Objetivo:* Aplicar el sistema en escenarios como la localización y seguimiento de personas para rescate, la supervisión de infraestructuras críticas, y otras tareas que puedan beneficiarse de un dron autónomo.

Bibliografía

[1] Pilotnet: A deep learning framework for autonomous driving. 2016. URL https://arxiv.org/abs/1604.07316.

[2] End-to-end deep learning for autonomous driving: A review. 2019. URL https://arxiv.org/abs/1903.11582.

[3] Deeppilot: A cnn for autonomous drone racing. 2020. URL https://www.mdpi.com/1424-8220/20/16/4524.

[4] Conducción autónoma de un vehículo en simulador mediante aprendizaje extremo a extremo basado en visión. 2019. URL https://gsyc.urjc.es/jmplaza/students/tfm-deeplearning_autonomous_navigation-vanessa-2019.pdf.

[5] Infraestructura de programación de robots aéreos y aplicaciones visuales con aprendizaje profundo. 2022. URL https://gsyc.urjc.es/jmplaza/students/tfm-drones-followperson-pedro_arias-2022.pdf.

[6] A review of quadrotor unmanned aerial vehicles: Applications, architectural design and control algorithms. 2021. URL https://link.springer.com/content/pdf/10.1007/s10846-021-01527-7.pdf.

[7] Diseño del modelado y control de un quad-rotor. 2023. URL https://upcommons.upc.edu/handle/2117/28592/browse?type=dateissued.

[8] Data augmentation in python: Everything you need to know. 2023. URL https://neptune.ai/blog/data-augmentation-in-python.

[9] Conducción autónoma de un robot con visión mediante aprendizaje por refuerzo. 2020. URL https://gsyc.urjc.es/jmplaza/students/tfm-reinforcementlearning-conduccion_autonoma-ignacio_arranz-2020.pdf.

[10] Autonomous drone racing: Time-optimal spatial iterative learning control within a virtual tube. 2023. URL https://arxiv.org/pdf/2306.15992.