Proyecto Fin de Grado
Ingeniería de Tecnologías Industriales

Aplicación de visión por computador y machine learning al guiado de un robot móvil basado en Raspberry Pi

Autor: Luis Gallardo Gómez

Tutor: Carlos Vivas Venegas

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Proyecto Fin de Grado
Ingeniería de Tecnologías Industriales

Aplicación de visión por computador y machine learning al guiado de un robot móvil basado en Raspberry Pi

Autor:
Luis Gallardo Gómez

Tutor:
Carlos Vivas Venegas
Profesor titular

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2021

Proyecto Fin de Grado: Aplicación de visión por computador y machine learning al guiado de un robot móvil basado en Raspberry Pi

Autor: Luis Gallardo Gómez

Tutor: Carlos Vivas Venegas

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2021

El Secretario del Tribunal

Agradecimientos

En primer lugar, agradecer a toda mi familia, compañeros y amigos por hacer este duro camino algo más ameno y ayudar en todo lo posible siempre. Creo que mi éxito se debe en gran medida a las personas que me acompañan en el día a día ya que en ellas encuentro apoyo y consejo sincero sobre cada situación a la que me enfrento.

Por último, agradecer a D. Carlos Vivas Venegas, mi tutor en el trabajo, por su constante actitud proactiva hacia el desarrollo del proyecto y su consejo desde que le contacté por primera vez con mi propuesta.

Luis Gallardo Gómez

Sevilla, 2021

Resumen

En este proyecto se propone el desarrollo de la maqueta de un vehículo con capacidad de guiado autónomo basado exclusivamente en visión, y que emplea cierta señalización situada en el suelo para su orientación. En concreto, el cerebro del vehículo será un Red Neuronal Convolucional (CNN) que es entrenada para seguir, a partir de las imágenes capturadas por la cámara a bordo, un camino previamente marcado en suelo en forma de una cuerda, cordón o cinta adhesiva.

Se ha construido un pequeño coche mediante impresión en 3D de piezas diseñadas para este trabajo. Se usa una Raspberry Pi 4 como unidad de procesamiento y una cámara oficial de Raspberry Pi como medio de percepción. La red neuronal ha sido desarrollada mediante Keras y entrenada y ejecutada a través de TensorFlow. El entrenamiento de la red se realiza gracias a la herramienta proporcionada por Google denominada Google Colab que da acceso a hardware, como GPUs, en la nube de forma gratuita.

La idea original del proyecto consistía en desarollar un vehículo completamente autónomo capaz de procesar las imágenes a bordo del propio vehículo procesando las imágenes directamente en la Raspberry Pi (RP), pero debido a problemas de software no se ha llegado a poder poner en práctica. Como solución complementaria se implementa una comunicación PC-RP de manera que un PC en tierra realice este procesamiento de imágenes, devolviendo al vehículo las instrucciones para su guiado. El precio a pagar en este caso es una tasa de muestreo de imágenes inferior a la inicialmente deseada, que no ha supuesto sin embargo menoscabo en la capacidad de guiado del vehículo.

La red ha sido entrenada mediante un dataset formado por 6370 imágenes clasificadas en cuatro clases diferentes, 5096 para el entrenamiento y 1274 para test. La precisión obtenida sobre el set de prueba completamente desconocido para la red ha sido de un 92,86%. El principal problema en el entrenamiento de la red ha sido el conseguir cierta homogeneidad en el porcentaje de acierto para todas las clases. Finalmente se consiguió teniendo un 90% de precisión en la clase peor predicha.

El comportamiento final del coche se puede considerar satisfactorio ya que se cumple plenamente la idea implementada siguiendo la línea sin problemas. Se considera que podría haber un salto de calidad importante en términos de velocidad de respuesta si se consiguiera implementar la ejecución de la red directamente sobre la RP.

Abstract

This project proposes the development of a model of a vehicle with autonomous guidance capacity based exclusively on vision, and that uses certain signs located on the ground for its orientation. Specifically, the brain of the vehicle will be a Convolutional Neural Network (CNN) that is trained to follow, from the images captured by the on-board camera, a path previously marked on the ground in the form of a rope or adhesive tape.

A small car has been built by 3D printing parts designed for this project. A Raspberry Pi 4 is used as a processing unit and an official Raspberry Pi camera as a means of perception. The neural network has been developed using Keras and trained and executed through TensorFlow. The training of the network is carried out thanks to the tool provided by Google called Google Colab that gives access to hardware, such as GPUs, in the cloud for free.

The original idea of the project was to develop a completely autonomous vehicle capable of processing the images on board the vehicle itself, processing the images directly on the Raspberry Pi (RP), but due to software problems it has not been implemented. As a complementary solution, a PC-RP communication is implemented so that a PC on the ground performs this image processing, returning the instructions for its guidance to the vehicle. The price to pay in this case is an image sampling rate lower than the one initially desired, which has not, however, impaired the vehicle's guiding capacity.

The network has been trained using a dataset made up of 6370 images classified in four different classes, 5096 for training and 1274 for testing. The precision obtained on the completely unknown test set for the network has been 92.86%. The main problem in training the network has been to achieve a certain homogeneity in the percentage of success for all classes. Finally, it was achieved by having a 90% precision in the worst predicted class.

The final behavior of the car can be considered satisfactory since the idea implemented following the line without problems is fully met. It is considered that there could be a significant quality jump in terms of response speed if the network execution could be implemented directly on the RP.

Índice

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

1.1. Motivación

La idea principal con la que nace este proyecto es la de mostrar el potencial que nos ofrece la Inteligencia Artificial (IA) a la hora de atacar prácticamente cualquier tipo de problema ya sea relacionado con retos tecnológicos, médicos o de cualquier otro campo y así dar una explicación a la brutal disrupción que está experimentando esta tecnología hoy en día en todo el mundo.

En concreto, los problemas a los que se están enfrentando en la actualidad las grandes empresas de la industria de la automoción para lograr el completo desarrollo de coches autónomos motiva la elección de la temática de este TFG. La principal tecnología en la que se sustentan estos coches es la IA y en declaraciones como esta de Elon Musk, fundador de Tesla, se nos pone en contexto de la dificultad de los retos que están enfrentando:

Figura 1-1. Declaraciones de Elon Musk (Twitter)

Una traducción literal del tweet de Elon Musk sería 'La beta del FSD 9 se enviará pronto, ¡lo juro! La conducción autónoma generalizada es un problema difícil, ya que requiere resolver una gran parte de la IA en el mundo real. No esperaba que fuera tan difícil, pero la dificultad es obvia en retrospectiva. Nada tiene más grados de libertad que la realidad'. Estas declaraciones ilustran cómo de complicado es el conseguir desarrollar coches completamente autónomos y justifica en cierta medida la gran cantidad de promesas incumplidas hechas al respecto desde diferentes ámbitos del sector.

Aunque el coche autónomo parece aún algo lejano, la IA ya está presente en muchos de los automóviles que conducimos y su integración en los coches irá creciendo en los próximos años, pues es el elemento clave de los sistemas de seguridad, de la futura conducción autónoma y de muchos servicios relacionados con la movilidad. Actualmente hay coches en el mercado con sistemas de IA sencillos (sin capacidad de aprendizaje), empleados en asistentes digitales y en algunas funciones de los sistemas de seguridad ADAS (Sistemas Avanzados de Asistencia a la Conducción). Estos últimos ya ofrecen una visión artificial a través de la cámara que montan en el parabrisas, de otros sensores y de algoritmos de procesamiento de imágenes. Gracias a ello pueden reconocer el entorno, identificar situaciones de riesgo y detectar, por ejemplo, marcas viales, señales, peatones o ciclistas.

Se espera que el mercado de los vehículos inteligentes crezca de manera exponencial. El Parlamento Europeo

estima que el error humano está involucrado en aproximadamente el 95% de los accidentes de tráfico que se registran en las carreteras del continente. Por este motivo, la tecnología digital en los vehículos sin conductor se vislumbra como una solución para reducir la siniestralidad y las congestiones.

Existen varios niveles de coche autónomo y algunos de los prototipos iniciales ya circulan por la red de carreteras de nuestro país, por ejemplo. Es el caso de los llamados modelos de nivel 1, que incluyen los conocidos sistemas ADAS de ayuda a la conducción, y de nivel 2 que controlan movimientos laterales y longitudinales siempre bajo supervisión del conductor. Los niveles de autonomía 3 y 4 ya presentan características de detección de objetos y capacidad de respuesta en las que el conductor ya no es el protagonista. De este último nivel, se prevé que las ventas en Europa y China sean del 15% para el 2035.

Si bien los vehículos autónomos (nivel 5) tienen el potencial de remodelar el transporte y la sociedad, sus desarrolladores se enfrentan a un gran problema: su seguridad. Desde software con fallas hasta inteligencia artificial engañada -IA- y manipulación ambiental, hay muchas vulnerabilidades de seguridad que necesitan soluciones antes de que estos vehículos puedan considerarse seguros para su uso en nuestras carreteras.

Los vehículos autónomos necesitan percibir el camino mejor que el mejor conductor humano. Este desafío se reduce a perfeccionar la visión por ordenador, mejorar el sistema de percepción utilizando cámaras y radares, así como construir mapas altamente detallados del entorno para que el vehículo pueda saber qué le rodea. El segundo desafío es analizar cómo desarrollar un sistema de inteligencia artificial que pueda tomar decisiones razonables en el camino. Por ejemplo, cuándo cambiar de carril y a qué velocidad viajar.

Hay desarrollos prometedores que están acercando coches autónomos a la realidad, pero también hay problemas importantes que necesitan soluciones. Por una parte, hay que garantizar que la tecnología y el software sean robustos y seguros. Pero, además, los fabricantes de automóviles y los organismos de la industria deben saber cómo crear programas de IA que tomen las decisiones éticas correctas a pesar de la subjetividad de la cuestión.

1.2. Objetivos

Es prácticamente imposible realizar avances significativos en esta materia dado el difícil acceso a los equipos que estas empresas usan para la investigación en IA y el nivel técnico que requiere. Sin embargo, los coches autónomos pueden ser considerados la punta de un iceberg compuesto por un número casi infinito de soluciones tecnólogicas que se derivarán de los avances realizados para la consecución de este hito y a algunas ya se tiene acceso.

Al hilo de lo anterior, el problema que se propone estudiar y resolver es el obtener un pequeño vehículo autónomo que se dedique a seguir algún tipo de señalización en el suelo mediante una cámara. En concreto, construiremos un vehículo autónomo que se dedicará a seguir líneas representadas por una cuerda en el suelo mediante el análisis, en una red neuronal previamente entrenada, de las imágenes obtenidas por una cámara incorporada en el coche. Las respuestas que la red neuronal proporcione tras su análisis se utilizarán para darle órdenes al vehículo sobre hacia donde se debe dirigir para seguir el camino marcado por la cuerda.

El vehículo que se usará será un pequeño coche construido mediante piezas impresas en una impresora 3D y diseñadas para este trabajo. Se usa una Raspberry Pi 4(RP) como unidad de procesamiento y una cámara compatible con la RP como medio de percepción del entorno. La red neuronal será definida mediante Keras y ejecutada gracias a TensorFlow, todo gracias a su entrenamiento en el entorno Google Colab.

El problema en una primera instancia parece sencillo, pero nos presenta bastantes retos que se serán descritos a lo largo del trabajo de manera que proporcionemos una primera aproximación al contexto de desarrollo que se vive en la industria sobre IA en nuestros días. Cabe decir que, aunque es un desarrollo muy primario, una vez que se tiene resuelto este primer paso, la tecnología y metodología usada tiene un potencial de adaptabilidad a otro tipo de problemas bastante importante que podrá ser usado para futuras investigaciones o mejoras.

2 ESTADO DEL ARTE

2.1. Contexto histórico

Este apartado está basado principalmente en el artículo "Breve Historia de la Inteligencia Artificial" escrito por Fernando Sancho Caparrini [1].

2.1.1 Nacimiento de la Inteligencia Artificial

Ya desde 1946, el conocido como uno de los padres de las Ciencias de la Computación, Alan Turing, había publicado ensayos proponiendo y analizando la conexión existente entre la recién nacida Computación y la capacidad de crear respuestas inteligentes por métodos mecánicos. Y no era el único, podemos encontrar trabajos muy elaborados que establecían esta relación, aunque quizás sin proporcionar aproximaciones tan globales, como el que en 1949 propone Claude Shannon (el mismo que se considera padre de la Teoría de la Información) para la creación de un jugador automático de ajedrez.

Aunque hubo varios trabajos relacionando diversos juegos de estrategia con la recién nacida computación, el trabajo de Shannon presenta una particularidad que lo sitúa como uno de los trabajos esenciales en la historia de la IA, y es que propone un algoritmo nuevo basado en un trabajo de otra figura fundamental de la computación, John Vonn Neuman, que llamó Minimax, para buscar la respuesta más adecuada que debería dar la máquina por medio de una representación de las posibles respuestas como un espacio de estados a la vez que intenta dar un método uniforme de resolución para todos los casos. Este trabajo se convertirá en la metodología estándar de los trabajos futuros y, hoy en día, la combinación de representar adecuadamente el problema, y proporcionar mecanismos para buscar una solución en la nueva representación, sigue siendo la piedra angular de las metodologías que se siguen desarrollando y depurando para alcanzar los objetivos de la IA.

En estos mismos años, A. Samuel, ingeniero de la recién nacida IBM, propone un jugador automático de damas que presenta una particularidad que adelanta cuál será una de las líneas más fructíferas del futuro de la IA, y es que sobre el algoritmo Minimax de Shannon, Samuel añade un proceso de aprendizaje que permite a su programa mejorar a medida que va jugando con seres humanos, ayudándole a decidir qué movimiento es el más adecuado en función de los resultados obtenidos de partidas anteriores. Los resultados fueron tan buenos que se obtuvo un jugador artificial que suponía un contrincante interesante, quizás no contra jugadores expertos, y que pudo implementarse en la primera máquina comercial de IBM, el IBM 701, de 1956. Este mecanismo de aprendizaje, que el propio Samuel llamó machine-learning, será el detonante de una de las técnicas de IA que más frutos proporcionará en un futuro lejano.

Figura 2-1. More, McCarthy, Misky, Selfridge y Solomonof, asistentes a la Conferencia de Darmouth.

(Fotógrado: Joe Mchling. AI Magazine)

La expresión Inteligencia Artificial (IA) se acuñó por primera vez en la Conferencia de Dartmouth en 1956. Los asistentes a esta conferencia serán las figuras fundamentales de la disciplina en los siguientes años. Los primeros resultados fueron abrumadores provocando una euforia exagerada en las predicciones que se realizaban sobre el futuro de este campo. Esta euforia fue responsable, en parte, de la desilusión que llevará a lo que se conoce como el invierno de la IA, etapa de falta de resultados y abandono por parte de los centros de investigación que durará muchos más años de lo esperado.

2.1.2 Invierno de la IA

En esta época denominada invierno de la IA se siguen dando avances significativos en la materia, pero a un ritmo menor de lo esperado.

En 1958, A. Newell, J.C. Shaw y H. Simon crean el programa General Problem Solver con la intención de proporcionar un programa que resolviera cualquier tipo de problema que pudiera expresarse por medio de un lenguaje formal. Pese a que la idea era correcta, la explosión combinatoria que se producía en la búsqueda de soluciones de muchos problemas evitaba que el programa proporcionara soluciones con un ordenador de la época, algo que permitió en su momento destacar la importancia de la complejidad computacional en la resolución de problemas de manera efectiva, y no solo centrar el foco en la búsqueda de métodos de resolución formalmente válidos.

En los años posteriores, dentro de la década de los cincuenta, y llevados por el éxito obtenido en las damas, se vuelve a atacar el problema de ajedrez, mucho más complejo en cuanto a variedad y número de movimientos, haciendo uso esencialmente de la misma metodología Minimax comentada, pero introduciendo métodos de poda en el árbol de juego y heurísticas para decidir entre el elevado número de movimientos a elegir. Estos mecanismos para reducir el número de jugadas a explorar se convertirán en otro de los ejes de soporte de la mayoría de los algoritmos posteriores. Para poder comparar la dificultad de los dos juegos elegidos, damas y ajedrez, indicaremos que, mientras el mejor programa de ajedrez de la época perdía ante jugadores de nivel inicial, el de damas era capaz de ganar a humanos con un nivel de juego alto.

Además de hacer avanzar las metodologías que se convertirán en estándares para abordar muchos problemas con IA es importante el uso que se hace de los juegos de estrategia como representantes de la inteligencia humana, y que responde a dos razones principalmente: proporcionan un marco muy bien definido en el que comparar la capacidad de la IA y la humana, y permiten una representación del problema directa, formal, y sencilla.

En los años ochenta y noventa se viven breves primaveras gracias a desarrollos como los Sistemas Expertos que se basaban en reglas lógicas apoyándose en algoritmos de búsqueda y heurística similares a los anteriormente comentados para juegos de estrategia, estos avances proporcionaban herramientas útiles para la toma de decisiones en entornos complejos. No se consideran dispositivos inteligentes, pero en algunos casos de aplicación obtenían mejores resultados que expertos humanos.

A finales de los ochenta comienza a usarse el concepto de agente inteligente, ente artificial con cierta capacidad de percepción, razonamiento y acción, y con él, la Inteligencia Artificial Distribuida, un conjunto de estos agentes interactuando entre sí con el fin de resolver un determinado problema.

A pesar de que la neurona artificial aparece en los primeros años de la IA, la capacidad computacional de los ordenadores de las primeras décadas no permitía hacer uso de un número de neuronas artificiales que proporcionase resultados interesantes. Además, no se disponía de ningún mecanismo para obtener redes de neuronas que resolviesen problemas concretos, aunque había resultados teóricos que avalaban esta posibilidad. Tras la creación del algoritmo de propagación hacia atrás por Werbos en 1974, se empezaría a extender su uso en problemas concretos de optimización y aproximación de funciones, pero incluso en los noventa todavía no se consideran importantes para la IA, y tienen un uso limitado a la aproximación de ciertos conjuntos de datos de no demasiada complejidad en el mundo de la Ingeniería.

2.1.3 Resurgimiento de la IA hasta la actualidad

La mayor transformación sufrida por la IA en el S.XXI ha sido el peso que ha tomado el Machine Learning

(ML), y que acapara la casi totalidad de nuevas técnicas que están desarrollándose. Hasta el momento, la vía más común con la que se abordaban problemas de IA era con la construcción de algoritmos, más o menos generales, para encontrar la solución del problema recorriendo el espacio de posibles soluciones, como hacen algoritmos como el mencionado Minimax.

La aproximación que propone ML es muy distinta, y pasa por utilizar datos reales del comportamiento deseado para construir una máquina que sea capaz de reconocerlos y simularlos. En este sentido, el problema se traduce en utilizar una correcta representación de los datos junto a algún mecanismo de optimización para poder ajustar la máquina al comportamiento observado, que es lo que se conoce como entrenamiento. Al igual que en la vía tradicional se crean algoritmos genéricos de búsqueda en el espacio de soluciones, en ML también se crean algoritmos de optimización generales que, trabajando sobre representaciones adecuadas, sean capaces de ajustar correctamente la máquina al comportamiento observado en los datos, independientemente del dominio del problema.

Los algoritmos y modelos de ML más famosos son las nuevas variantes derivadas de las redes neuronales artificiales. Se sabía que la capacidad de adaptación de las redes neuronales depende de la estructura que tenga la red y que, cuanto más compleja es su estructura, más costoso es dar con el ajuste óptimo para asegurar que la máquina se ajuste al comportamiento de los datos hasta el punto de que el algoritmo de entrenamiento de propagación hacia atrás clásico se vuelve incapaz de devolver una buena red. Durante la segunda década del S.XXI, la tipología de las redes y los algoritmos de entrenamiento desarrollados han dado un salto cualitativo en su capacidad de ajuste, proporcionando redes capaces de aprender comportamientos a partir de datos cada vez más complejos (imágenes, textos, audio, video, etc.).

Estas mejoras vienen apoyadas por dos realidades tecnológicas. Por una parte, las nuevas tecnologías de la comunicación y almacenamiento han hecho posible disponer de una gran cantidad de datos de todo tipo de procesos de forma sencilla. Por otra parte, la aparición de máquinas de procesamiento paralelo a un precio asequible permite realizar cálculos que hasta hace poco eran computacionalmente inviables o estaban reservados a grandes centros tecnológicos.

Como muestra de los avances que el uso de ML ha proporcionado a los problemas clásicos de IA, tenemos uno de los resultados más llamativos del equipo Google DeepMind, que ha creado en un tiempo récord un programa, AlphaGo, capaz de ganar a los campeones mundiales de Go varias décadas antes de lo que los mejores pronósticos vaticinaban. Para su creación, este equipo ha hecho uso de técnicas clásicas –Minimax, heurísticas– junto con procesos de aprendizaje en los que, tal y como vimos en el pasado, la máquina se enfrenta a versiones de sí misma para poder aprender de millones de partidas en un tiempo mínimo –apoyado en la inmensa capacidad computacional de Google, claro.

Pero las aplicaciones de ML para la IA no se restringen al Go y cada día aparecen nuevos resultados que mejoran soluciones anteriores. Por ejemplo: el mismo equipo de Google ha generalizado su método para que la misma máquina pueda aprender a jugar a otros juegos adquiriendo niveles de maestro tras pocas horas de entrenamiento; el Procesamiento de Lenguaje Natural ha avanzado más en los últimos tres años que en los veinte años anteriores; los algoritmos de reconocimiento de imágenes son capaces de alcanzar ya cotas de fiabilidad similares a las de un ser humano y, mezcladas con técnicas de lenguaje, incluso son capaces de describir textualmente el contenido y acciones presentes en una imagen; la diagnosis de enfermedades y la generación de nuevos fármacos se ha visto potenciada de forma espectacular gracias al análisis de datos acumulados durante décadas por las empresas farmacéuticas y grupos de investigación médica, proponiendo nuevas vías en la creación de fármacos y en la detección de enfermedades; conseguir vehículos que se conducen solos y toman decisiones complejas en tiempo real para aumentar la seguridad de los viajeros está a la vuelta de la esquina; etc.

2.2. Redes Neuronales Artificiales (RNA)

La programación de redes neuronales artificiales (RNAs) es uno de los mayores hitos jamás logrados. En la programación convencional se dividen grandes problemas en muchas tareas pequeñas, definidas con precisión, dándole instrucciones claras a la computadora de manera que ésta las pueda realizar fácilmente. En cambio, las RNAs construyen una solución propia al problema en cuestión a partir de los datos proporcionados u observados.

2.2.1 Definición

Mientras que las computadoras actuales son muy rápidas en el procesamiento de información, existen tareas complejas como el reconocimiento y clasificación de patrones que les demandan mucho tiempo y esfuerzo resolverlas. Sin embargo, la experiencia nos muestra que el cerebro humano es muy eficiente en la resolución de ese tipo de tareas y en la mayoría de las ocasiones las resuelve sin aparente esfuerzo. Una Red Neuronal Artificial (RNA) es un modelo matemático inspirado en el comportamiento biológico de las neuronas y en cómo se organizan formando la estructura del cerebro humano.

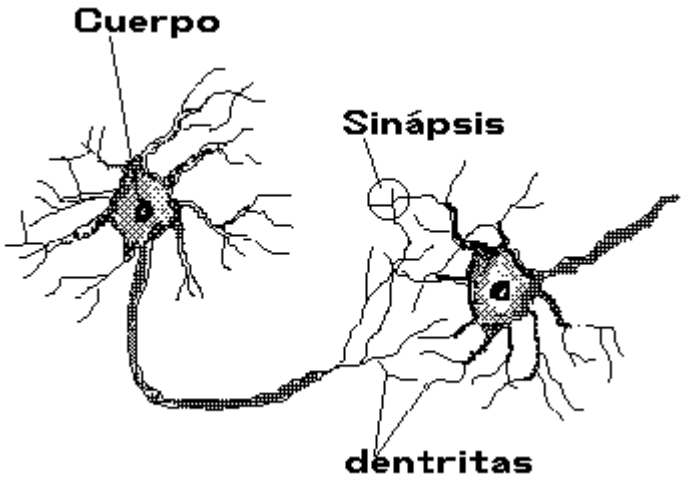

Figura 2-2. Estructura básica de neurona biológica

La imagen superior es un esquema simplicado del tipo más común de neurona biológica y se compone por las siguientes partes:

- El cuerpo central, llamado soma, que contiene el núcleo celular.
- Una prolongación del soma, el axón.
- Un conjunto de ramificaciones terminales, las dendritas.
- Zonas de conexión entre una neurona y otra, conocidas como sinapsis.

Las neuronas, al igual que las demás células del cuerpo, funcionan a través de impulsos eléctricos y reacciones químicas. Los impulsos eléctricos que utiliza una neurona para intercambiar información con las demás, viajan por el axón que hace contacto con las dendritas de la neurona vecina mediante las sinápsis. El cerebro es un conjunto de un gran número de neuronas interconectadas entre sí. Las RNAs son un intento de emular la manera en la que responde el cerebro ante estímulos del exterior y el aprendizaje que realiza.

2.2.1.1 Perceptrón de McCulloch-Pitts

El primer modelo matemático de una neurona artificial, creado con el fin de llevar a cabo tareas simples, fue presentado en el año 1943 en un trabajo conjunto entre el psiquiatra y neuroanatomista Warren McCulloch y el matemático Walter Pitts. Este modelo es conocido como perceptrón de McCulloch-Pitts y es la base de la mayor parte de la arquitectura de las RNAs diseñadas hasta la fecha.

El modelo puede ser fácilmente descrito con la ayuda de la siguiente figura:

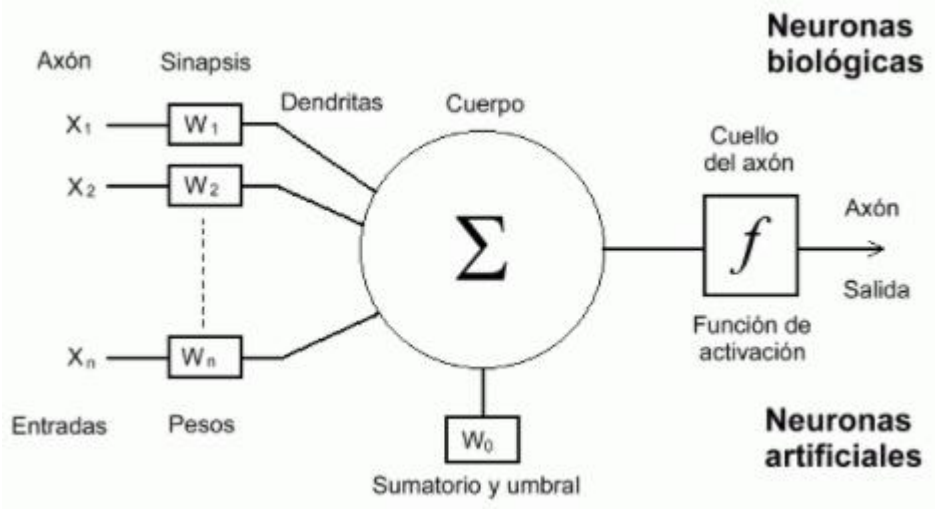

Figura 2-3. Neurona arfificial

La neurona consta de:

- Un conjunto de entradas x1, …xn.
- Los pesos sinápticos w1, …wn, correspondientes a cada entrada.
- Un umbral Wo, también denominado sesgo(b).
- Una función de agregación, Σ.
- Una función de activación, f.
- Una salida, Y

Las entradas son el estímulo que la neurona artificial recibe del entorno que la rodea, y la salida es la respuesta a tal estímulo. La neurona puede adaptarse al medio circundante y aprender de él modificando el valor de sus pesos sinápticos y su sesgo, y por ello son conocidos como los parámetros libres del modelo, ya que pueden ser modificados y adaptados para realizar una tarea determinada.

En este modelo, la salida neuronal Y está dada por:

$$Y = f(\sum_{i=1}^{n} w_i x_i + W_o)$$

Las funciones de activación empleadas pueden ser de diferente tipo y se eligen de acuerdo con la tarea que se vaya a realizar. Algunas veces son funciones lineales, otras funciones sigmoidales (p.ej. la tanh), y otras funciones de umbral de disparo (el percetrón inicial modelado era de este tipo, daba un resultado binario). Se ampliará sobre funciones de activación en apartados posteriores.

2.2.1.2 Perceptrón de multicalpa o RNA

El concepto de perceptrón multicapa (por razones históricas, a pesar de que las neuronas podrían ser p. ej. de tipo sigmoidal) o red neuronal artificial (RNA) consiste en un conjunto de las anteriormente definidas neuronas artificiales conectadas entre sí formando redes de neuronas. Aunque la forma en que se pueden agrupar es completamente libre y variada, el caso más común de agrupación se corresponde con una organización en capas ordenadas de manera que las salidas de las neuronas de una capa son las entradas de las neuronas de la capa siguiente.

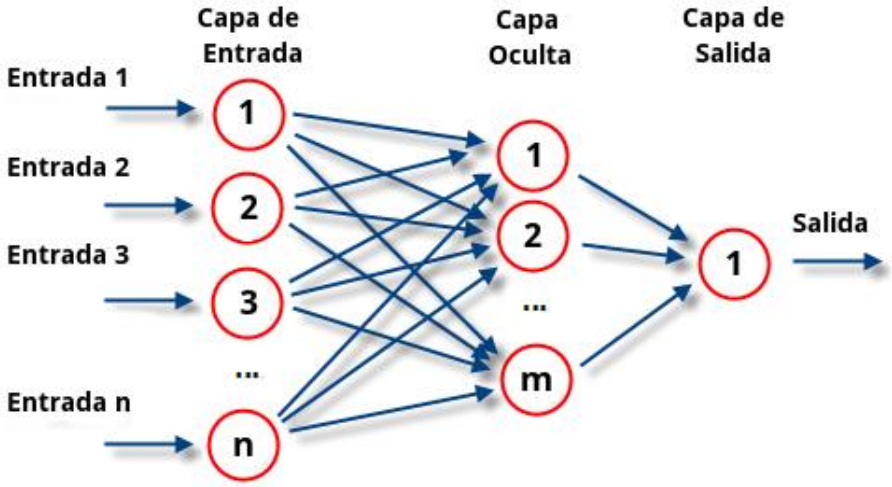

Figura 2-4. Red neuronal artificial

Así, tenemos la **capa de entrada** formada por las entradas de la información exterior a la red (que realmente no son neuronas), la **capa de salida** formada por las neuronas que constituyen la salida final de la red y de la que se obtiene el resultado del modelo, y las **capas ocultas** formadas por las neuronas que se encuentran entre los nodos de entrada y de salida, en estas capas se llevan a cabo la práctica totalidad de los cálculos que definen el comportamiento de la red y, por tanto, es donde realmente incide de manera efectiva el entrenamiento de la red. Una RNA puede tener varias capas ocultas o no tener ninguna de ellas. Generalmente, mientras más capas ocultas existan en la red, más profunda se dice que es y mejores resultados se suelen obtener, esto no se cumple siempre.

2.2.2 Tipos de RNA

Dada la total disrupción que sufre esta tecnología hoy en día y, como consecuencia, la gran investigación que se realiza es complicado encontrar una forma clara de clasificación de las RNAs que podemos usar.

En este trabajo vamos a clasificarlas según dos criterios: metodología de aprendizaje y arquitectura de la red. De esta forma pretendemos transmitir cierta claridad al concepto de RNA y justificar las decisiones que tomamos posteriormente.

2.2.2.1 Según método de aprendizaje

El criterio que se sigue a la hora de modificar los valores de los pesos de la red cuando se pretende que aprenda nueva información es un aspecto importante en el entrenamiento de esta. Se pueden distinguir principalmente dos métodos:

Aprendizaje supervisado

El aprendizaje supervisado se caracteriza porque el proceso de aprendizaje se realiza mediante un entrenamiento controlado por un agente externo que determina la respuesta que debería generar la red a partir de una entrada determinada. El supervisor controla la salida de la red y, en caso de que ésta no coincida con la deseada, se procederá a modificar los pesos de las conexiones con el fin de conseguir que la salida obtenida se aproxime a la deseada.

En este tipo de aprendizaje se suelen considerar, a su vez, tres formas de llevarlo a cabo, que dan lugar a los siguientes:

- Aprendizaje por corrección de error:

  Consiste en ajustar los pesos de las conexiones de la red en función de la diferencia entre los valores deseados y los obtenidos a la salida de la red, es decir, en función del error cometido en la salida.

  Un método que pertenece a esta clasificación es LMS Error (Least Mean Squared Error) o Regla del Mínimo Error Cuadrado que permite cuantificar el error global cometido en cualquier momento durante

el proceso de entrenamiento de la red y repartirlo entre las conexiónes de neuronas predecesoras para corregirlo. La Regla de Aprendizaje de Propagación Hacia Atrás o Backpropagation es una generalización de la anterior y fue la primera que permitió el entrenamiento de redes con capas ocultas, es la regla más utilizada, será descrita con más profundidad en apartados posteriores.

- Aprendizaje por refuerzo:

  Se basa en la idea de no disponer de un ejemplo completo del comportamiento deseado, es decir, de no indicar durante el entrenamiento exactamente la salida que se desea que proporcione la red ante una determinada entrada. Un supervisor se dedica a indicar mediante una señal de refuerzo si la salida obtenida en la red se ajusta a la deseada y, en función de ello, se ajustan los pesos basándose en un mecanismo de probabilidades.

- Aprendizaje estocástico:

  Consiste básicamente en realizar cambios aleatorios en los valores de los pesos de las conexiones de la red y evaluar su efecto a partir del objetivo deseado y de distribuciones de probabilidad. Si el comportamiento de la red se acerca al deseado, se acepta el cambio; si, por el contrario, se aleja, se aceptaría el cambio en función de una determinada y preestablecida distribución de probabilidades.

Aprendizaje no supervisado

Las redes con aprendizaje no supervisado no requieren influencia externa para ajustar los pesos de las conexiones entre sus neuronas. La red no recibe ninguna información por parte del entorno que le indique si la salida generada en respuesta a una determinada entrada es o no correcta.

Estas redes deben encontrar las características, regularidades, correlaciones o categorías que se puedan establecer entre los datos que se presenten en su entrada. Existen varias posibilidades en cuanto a la interpretación de la salida de estas redes, que dependen de su estructura y del algoritmo de aprendizaje empleado.

Se suelen diferencias dos tipos:

- Aprendizaje Hebbiano:

  Esta regla de aprendizaje es la base de muchas otras, la cual pretende medir la familiaridad o extraer características de los datos de entrada. El fundamento es una suposición bastante simple: si dos neuronas Ni y Nj toman el mismo estado simultáneamente (ambas activas o ambas inactivas), el peso de la conexión entre ambas se incrementa.

- Aprendizaje competitivo y comparativo:

  Se orienta a la clasificación de los datos de entrada. Como característica principal del aprendizaje competitivo se puede decir que, si un patrón nuevo se determina que pertenece a una clase reconocida previamente, entonces la inclusión de este nuevo patrón a esta clase matizará la representación de esta. Si el patrón de entrada se determinó que no pertenece a ninguna de las clases reconocidas anteriormente, entonces la estructura y los pesos de la red neuronal serán ajustados para reconocer la nueva clase.

2.2.2.2 Según arquitectura de la red

Existe un gran número de arquitecturas de redes neuronales diferentes entre las que se puede elegir, cuyo comportamiento y resultados varían según la aplicación para la que vaya a ser usada:

- Red neuronal Monocapa – Perceptrón simple:

  Son las redes neuronales más simples, constan de una capa de entrada y una capa de salida. Son capaces de representar funciones lineales. En las redes monocapa, se establecen conexiones entre las neuronas que pertenecen a la única capa que constituye la red. Un ejemplo muy sencillo de estas redes puede ser el de predecir la nota de un examen a partir de las notas de los parciales.

- Red neuronal Multicapa – Perceptrón multicapa:

  Gracias a las capas ocultas, el sistema es capaz de representar funciones no lineales. A medida que la

red neuronal va aprendiendo, la propia capa oculta es capaz de eliminar los enlaces que no considere relevantes. Normalmente, todas las neuronas de una capa reciben señales de entrada desde otra capa anterior, y envían señales de salida a una capa posterior. A estas conexiones se las denomina conexiones hacia adelante o feedforward. Sin embargo, en un gran número de estas redes también existe la posibilidad de conectar la salida de las neuronas de capas posteriores a la entrada de capas anteriores; a estas conexiones se las denomina conexiones hacia atrás o feedback.

- Red neuronal Convolucional (CNN):

  La principal diferencia de la red neuronal convolucional con el perceptrón multicapa viene en que cada neurona no se une con todas y cada una de las capas siguientes, sino que solo con un subgrupo de ellas (se especializa), con esto se consigue reducir el número de neuronas necesarias y la complejidad computacional necesaria para su ejecución.

  Su funcionalidad se centra en aplicaciones relacionadas con la visión por computador, como puede ser el reconocimiento y la clasificación de imágenes. Por ello, es el tipo de red elegida para nuestro trabajo y será descrita con más profundidad en apartados posteriores.

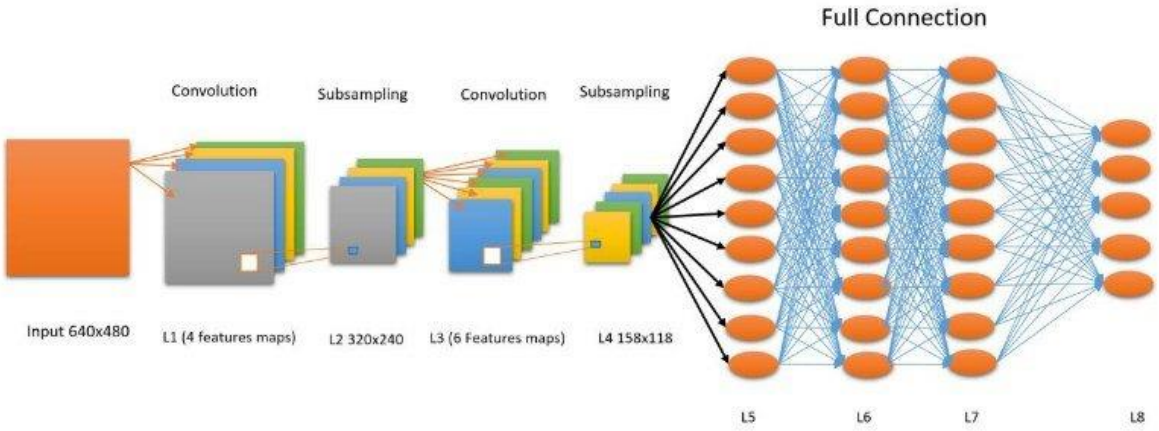

Figura 2-5. Red neuronal convolucional

- Red neuronal recurrente (RNN):

  Las redes neuronales recurrentes no tienen una estructura de capas, sino que permiten conexiones arbitrarias entre las neuronas, incluso pudiendo crear ciclos, con esto se consigue crear la temporalidad, permitiendo que la red tenga memoria. Los datos introducidos en el momento t en la entrada, son transformados y van circulando por la red incluso en los instantes de tiempo siguientes t + 1, t + 2, …

  Se emplean para aplicaciones en las que los datos de entrada son secuenciales, como el reconocimiento de sonidos para su transcripción o aplicaciones de traducción automática por computador.

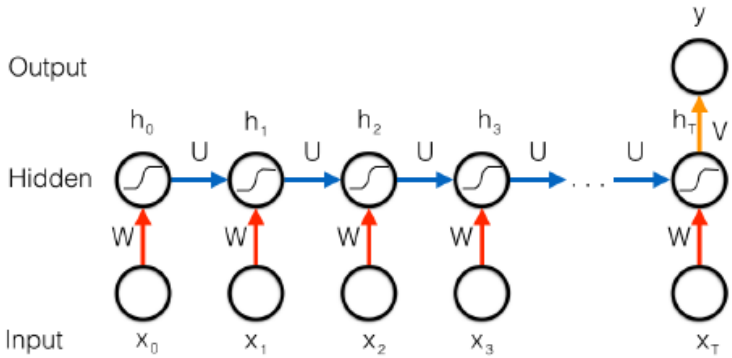

Figura 2-6. Red neuronal recurrente

- Red de base radial (RBF):

  Las redes de base radial calculan su salida en función de la distancia a un punto denominado centro. La salida es una combinación lineal de las funciones de activación radiales utilizadas por las neuronas individuales. Tienen la ventaja de que no presentan mínimos locales donde la retropropagación pueda quedarse bloqueada.

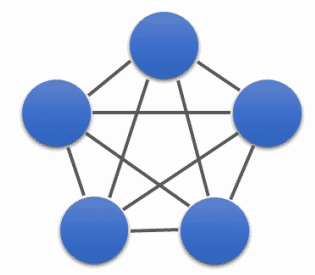

Figura 2-7. Red de base radial

- Redes híbridas:

  Para aplicaciones más complicadas, se crean combinaciones de los anteriores tipos de redes neuronales. Por ejemplo, la conducción de coches autónomos requiere de redes híbridas.

Los tipos de redes neuronales anteriormente explicados son los más generalizados. Realmente existen muchos más y derivaciones de los anteriores, todas estas arquitecturas surgen como consecuencia de aplicaciones cada vez más precisas y complejas.

2.3. Conceptos extra

2.3.1 Funciones de activación más usadas

Tanto en las redes neuronales artificiales como biológicas, una neurona no sólo transmite la entrada que recibe. La función de activación es análoga a la tasa de potencial de acción disparado en el cerebro. Esta función utiliza la suma ponderada calculada en la neurona con los datos de entrada a ella y la transforma dando lugar a la salida de la neurona. Cada una de las capas que conforman la red neuronal tienen una función de activación que permitirá reconstruir o predecir. En la gran mayoría de casos se usará una función no lineal debido a que le permite al modelo adaptarse para trabajar con la mayor cantidad de datos.

Se buscan funciones en las que las derivadas sean simples, para minimizar con ello el coste computacional. Tipos más usados:

- Sigmoid – Sigmoide: También conocida como función logística, la función sigmoide transforma los valores introducidos a una escala (0,1), donde los valores altos tienden a 1 y los valores muy negativos tienden a 0. Se usa en la última capa para clasificar datos. No está centrada en cero. Las redes neuronales de muchas capas se vuelven muy difíciles de entrenar dado el problema de desaparición de gradiente que esta función provoca. Su uso no está muy extendido actualmente.

$$f(x) = \frac{1}{1 - e^{-x}}$$

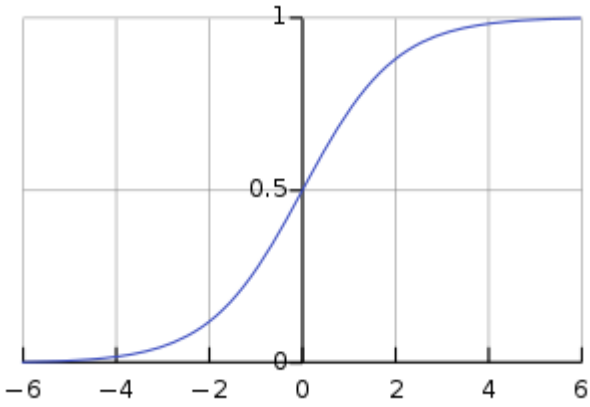

Figura 2-8. Función sigmoide

- Tanh – Tangent Hyperbolic – Tangente hiperbólica: Esta función es muy similar a la sigmoide pero centrada en 0. Presenta el mismo problema de desaparición del gradiente que la sigmoide, complicando el aprendizaje de la red. Tiene un buen desempeño en redes recurrentes.

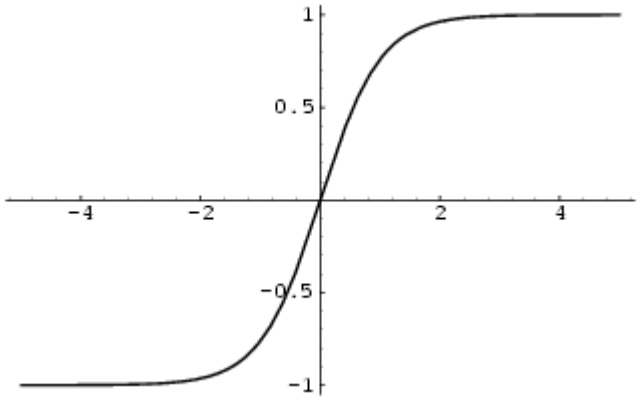

Figura 2-9. Función tangente hiperbólica

$$f(x) = \frac{2}{1 + e^{-2x}} - 1$$

- ReLU – Rectified Linear Unit: La función ReLU transforma los valores introducidos anulando los valores negativos y dejando los positivos tal y como entran. El gradiente de esta función será cero en el segundo cuadrante y uno en el primer cuadrante. Cuando se tiene que la función es igual a cero y su derivada también lo es, se genera lo que es la muerte de neuronas, a pesar de que puede ser un inconveniente en algunos casos, permite la regularización Dropout. Por esta razón, la función ReLu tiene variantes como Leaky ReLU o Parametric ReLU que previenen que existan neuronas muertas debido a la pequeña pendiente que existe cuando x<0. La mayoría de las redes neuronales de hoy en día utilizan ReLU o una de sus variantes debido al rápido aprendizaje que posibilitan. Se comporta bien con imágenes y da un buen desempeño en redes convolucionales.

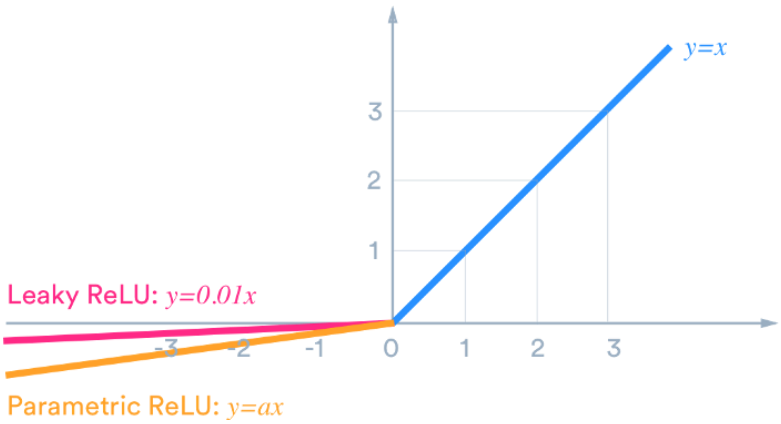

Figura 2-10. Funciones ReLU.

$$f(x) = \begin{cases} x & \text{if } x > 0, \\ ax & \text{otherwise.} \end{cases}$$

- Softmax: Esta función de activación devuelve la distribución de probabilidad de cada una de las clases soportadas en el modelo: transforma el vector de entrada a la función en una representación en forma de probabilidades, de tal manera que el sumatorio de todas las probabilidades de las salidas dé 1. Esta función es muy diferenciable. La fórmula calcula la exponencial del valor de entrada dado y la suma de los valores exponenciales de todos los valores en las entradas. Luego, la relación de la exponencial del valor de entrada y la suma de los valores exponenciales es la salida de la función Softmax. Por ejemplo: si el vector de entrada es [1, 2, 3, 4, 1, 2, 3], la función softmax retorna [0.024, 0.064, 0.175, 0.475, 0.024, 0.064, 0.175], de esta manera se facilita la clasificación multiclase normalizándola. Proporciona buen rendimiento en las últimas capas.

$$f(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

2.3.2 Regularización Dropout

Lo que se busca al entrenar una red neuronal es que ésta generalice para el máximo número posible de situaciones y no se sobreajuste al set de entrenamiento usado captando características de éste que no sean significativas. Para ello, normalmente los datasets se dividen en un 70-30% u 80-20% en set de entrenamiento y validación. El set de entrenamiento es usado puramente para entrenar la red mientras que el set de validación se usa para poder medir cómo de bien generaliza nuestra red. El set de validación es "nuevo" para la red ya que ésta fue entrenada con el de entrenamiento.

La regularización Dropout es un método que previene en cierta manera del sobreajuste y proporciona una manera de combinar de manera exponencial muchas arquitecturas de redes neuronales diferentes de manera eficiente. La traducción literal de "dropout" sería "abandonar" o "expulsar" unidades en una red neural. El abandono de una neurona supone su eliminación de la red junto con todas sus conexiones entrantes y salientes, reduciendo de manera notable en tamaño de nuestra red.

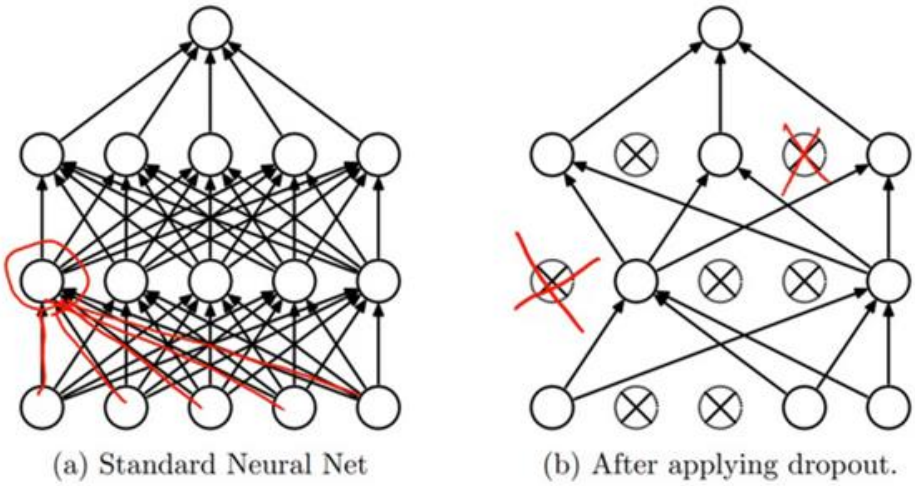

Figura 2-11. Regularización Dropout. (Medium.com)

El procedimiento es sencillo: por cada nueva entrada a la red en fase de entrenamiento, se desactivará aleatoriamente un porcentaje de las neuronas en cada capa oculta, acorde a una probabilidad de descarte previamente definida. Dicha probabilidad puede ser igual para toda la red, o distinta en cada capa. Lo que se consigue con esto es que ninguna neurona memorice parte de la entrada; que es precisamente lo que sucede cuando tenemos sobreajuste.

Una vez tengamos el modelo listo para realizar predicciones sobre muestras nuevas, debemos compensar de alguna manera el hecho de que no todas las neuronas permanecieran activas en entrenamiento, ya que al realizar predicciones ya fuera del entorno de entrenamiento, sí que estarán todas funcionando y, por tanto, habrá más activaciones contribuyendo a la salida de la red. Un ejemplo de dicha compensación podría ser multiplicar todos los parámetros por la probabilidad de no descarte.

2.3.3 Back-Propagation

La propagación hacia atrás de errores o retropropagación (del inglés backpropagation) es un método de cálculo del gradiente utilizado en algoritmos de aprendizaje supervisado usados para entrenar RNAs. Estos sistemas aprenden y se forman a sí mismos, en lugar de ser programados de forma explícita, y sobresalen en áreas donde la detección de soluciones o características es difícil de expresar con la programación convencional. El método emplea un ciclo propagación-adaptación de dos fases. Una vez que se ha aplicado un patrón a la entrada de la red como estímulo, éste se propaga desde la primera capa a través de las capas siguientes de la red, hasta generar una salida. La señal de salida se compara con la salida deseada y se calcula una señal de error para cada una de las salidas.

Las salidas de error se propagan hacia atrás, partiendo de la capa de salida, hacia todas las neuronas de la capa oculta que contribuyen directamente a la salida. Sin embargo, las neuronas de la capa oculta solo reciben una fracción de la señal total del error, basándose aproximadamente en la contribución relativa que haya aportado cada neurona a la salida original. Este proceso se repite, capa por capa, hasta que todas las neuronas de la red hayan recibido una señal de error que describa su contribución relativa al error total.

Esto permite que los pesos sobre las conexiones de las neuronas ubicadas en las capas ocultas cambien durante el entrenamiento. La importancia de este proceso consiste en que las neuronas de las capas intermedias se organizan a sí mismas de tal modo que las distintas neuronas aprenden a reconocer distintas características del espacio total de entrada. Después del entrenamiento, cuando se les presente un patrón arbitrario de entrada que contenga ruido o que esté incompleto, las neuronas de la capa oculta de la red responderán con una salida activa si la nueva entrada contiene un patrón que se asemeje a aquella característica que las neuronas individuales hayan aprendido a reconocer durante su entrenamiento.

Todo el proceso anterior se basa en el algoritmo de **descenso del gradiente**. El descenso del gradiente es un algoritmo de optimización iterativo para encontrar el mínimo de una función, en nuestro caso sería la función

del error con respecto a cada uno de los parámetros que deseamos ajustar. Para encontrar el mínimo local de una función usando el descenso de gradiente hemos de ir dando pasos proporcionales en la dirección del negativo del gradiente de la función en el punto en el que estemos.

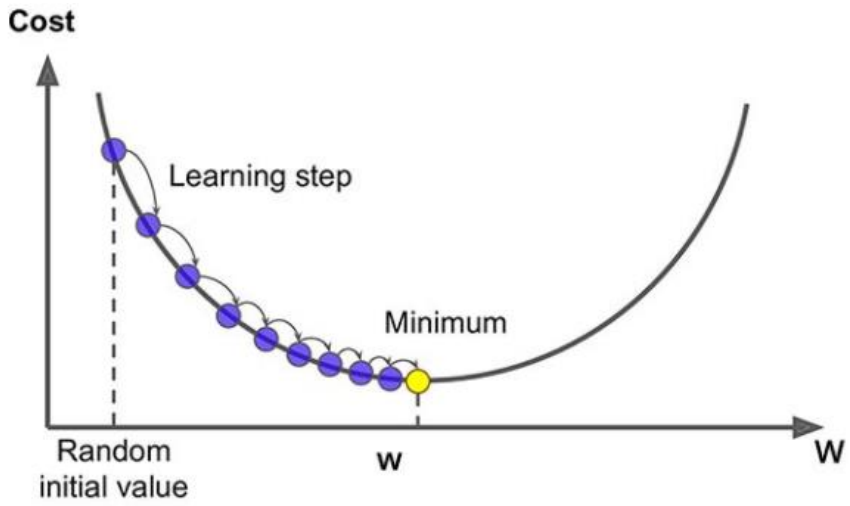

Figura 2-12. Gradient descent

2.3.3.1 Descripción matemática del algoritmo

La idea es minimizar la función de coste que nos da una magnitud del error que comete la red. Para una muestra i, la función de coste, por ejemplo, es:

$$J_i = (y_i - p_i)^2$$

En la expresión, y_i es la etiqueta (valor deseado de salida de la red) correspondiente a la muestra i , y p_i es el valor devuelto por la red para dicha muestra. Considerando que el comportamiento de la neurona de la capa L viene determinado por el peso del enlace, W^L, que la une con el valor devuelto por la capa anterior, a^{L-1} , y por el bias o sesgo a añadir, b^L :

$$p_i = f(W^L a^{L-1} + b^L)$$

Por comodidad, establecemos como Z^L:

$$Z^L = W^L a^{L-1} + b^L$$

Y, por tanto:

$$p_i = f(Z^L)$$

Valiéndonos de la regla de la cadena obtenemos la siguiente expresión para el cálculo de las derivadas parciales de la función del error con respecto al peso del enlace, W^L:

$$\frac{dJ_i}{dW^L} = \frac{dJ_i}{dp_i}\frac{dp_i}{dZ^L}\frac{dZ^L}{dW^L}$$

Una vez llegados a este punto, se puede observar que el cálculo de las derivadas parciales necesarias es trivial:

$$\frac{dJ_i}{dp_i} = 2 * (y_i - p_i)$$

$$\frac{dZ^L}{dW^L} = a^{L-1}$$

Y $\frac{dp_i}{dZ^L}$ es la derivada de la función de activación con la que estemos trabajando en la neurona.

El proceso anteriormente detallado explicado cómo se calcularía la derivada parcial de la función del error con respecto a cualquier peso de enlace de una RNA. El proceso con respecto a los sesgos sería completamente

análogo.

La expresión que refleja el mecanismo de iteración de este método de entrenamiento de los parámetros es:

$$W^L = W^{L-1} - \alpha \frac{dJ_i}{dW^{L-1}} \qquad b^L = b^{L-1} - \alpha \frac{dJ_i}{db^{L-1}}$$

Siendo α el ratio de aprendizaje ("learning rate") que controla la magnitud de cambio en cada iteración y, $\frac{dJ}{dW}$ y $\frac{dJ}{db}$ las derivadas respecto de la función coste de los parámetros de la red: peso sipnático(W) y sesgo(b).

En este enlace podemos encontrar una explicación muy aclaratoria de lo que sería el método de Backpropagation aplicado a una red neuronal multicapa. Sólo considera la actualización de los pesos sinápticos, pero es muy útil ya que para los sesgos sería un proceso análogo.

2.3.3.2 Elección de función de coste o pérdida

La función de pérdida, también conocida como función de coste, es la función que nos dice. durante el entrenamiento de la red, lo lejos que está en un momento dado lo que la red nos ofrece como salida y el resultado que nosotros consideramos que es el correcto o deseado. Esta es la función que optimizamos o minimizamos cuando realizamos el Back-propagation.

Existen varias funciones matemáticas que pueden usarse, la elección dependerá del problema que se esté resolviendo. Las más utilizadas son las siguientes:

- Error Cuadrático Medio ('Mean Square Error', MSE): Calcula la distancia 'geométrica' al cuadrado con respecto al valor objetivo. MSE se puede usar, por ejemplo, en problemas de regresión a valores arbitrarios y con una última capa sin función de activación.

$$(y_i - p_i)^2$$

- Entropía Cruzada Categórica ('Categorical Cross Entropy'): Es una medida de la distancia entre distribuciones de probabilidad. La entropía cruzada suele ser adecuada en modelos de redes cuya salida representa una probabilidad, como cuando hacemos una clasificación categórica con función de activación 'softmax'.

$$-(y \log(p) + (1 - y)\log(1 - p))$$

2.3.4 Optimizador Adagrad

Cuando se definen los hiperparámetros de una red neuronal, se le da un cierto valor a la tasa de aprendizaje (learning rate) y se supone que ésta permanece constante e igual para todos los pesos a reajustar durante el entrenamiento.

Supongamos que tenemos cierto dataset de entrenamiento en el que una pequeña parte de los datos contiene una característica, mientras que otra característica se encuentra en casi todos los datos. Asumimos que, al entrenar una red, ciertas características van ligadas a ciertas neuronas. Según la situación expuesta y el mecanismo de Backpropagation explicado, los pesos de las neuronas que correspondan con la característica más común en nuestro dataset se actualizarán más rápidamente, mientras que los relacionados con la característica que aparecía en pocos datos habrán tenido menos posibilidades de reajustarse ya que tienen la misma tasa de aprendizaje que las neuronas de la característica común, pero las oportunidades de actualización han sido mucho menores. Este hecho conduce a que nuestra red se entrene incorrectamente.

El algoritmo u optimizador Adagrad está pensado para paliar este problema ya que permite que las características de los datos para el aprendizaje de nuestra red ajusten automáticamente el tamaño de la tasa de aprendizaje. Formalmente se puede describir como:

$$m_t = g_t$$

$$V_t = \sum_{\tau}^{t} g_\tau^2$$

$$\theta_{t+1} = \theta_t - \eta \frac{m_t}{\sqrt{V_t}}$$

Donde Θ es el parámetro que se actualiza, g(t) es el gradiente del parámetro en el momento t y η es la tasa de aprendizaje. V(t) es el denominado impulso de segundo orden, resultado de la suma acumulada de los cuadrados del gradiente. Como para las características poco comunes en el dataset la actualización del parámetro será poco frecuente, la suma acumulada del cuadrado del gradiente de éste será relativamente pequeña, por lo que en la fórmula de actualización del parámetro se hará mayor la tasa de aprendizaje y, por lo tanto, esta característica será aprendida de una forma más rápida. Para evitar que el denominador anterior sea 0, a menudo se agrega un parámetro de término de suavizado ε quedando:

$$\theta_{t+1} = \theta_t - \eta \frac{m_t}{\sqrt{V_t + \epsilon}}$$

El contrapunto de esta solución es que provoca que la tasa de aprendizaje sea cada vez más insignificante conforme aumenta la cantidad de entrenamiento, por lo que se corre el riesgo de que llegue un momento en el que sea imposible actualizar de manera efectiva algún parámetro.

2.4. Redes Neuronales Convolucionales (CNN)

Una red neuronal convolucional (CNN) es un tipo de RNA con aprendizaje supervisado que procesa sus capas imitando al cortex visual del ojo humano para identificar distintas características en las entradas que, en definitiva, hacen que pueda identificar objetos y "ver". Para ello, la CNN contiene varias capas ocultas especializadas y con una jerarquía: esto quiere decir que las primeras capas pueden detectar lineas, curvas y se van especializando hasta llegar a capas más profundas que reconocen formas complejas como un rostro o la silueta de un animal.

Su principal ventaja es que a cada parte de la red se le entrena para realizar una tarea, esto reduce significativamente el número de capas ocultas, por lo que el entrenamiento es más rápido. Además, presenta invarianza a la traslación de los patrones a identificar.

2.4.1 Fundamento Biológico

A partir de 1959, se comenzó a comprender cómo funciona la corteza visual gracias, en gran medida, a un trabajo publicado por Hubel y Wiesel. En concreto, la publicación explicaba el papel de las células responsables de la selectividad de orientación y detección de bordes en los estímulos visuales dentro de la corteza visual primaria V1. Principalmente, dos tipos de células fueron identificadas para realizar dicha selectividad y ambas contenienen campos receptivos alargados de manera que tienen una mejor sensibilidad a estímulos visuales alargados como líneas y bordes:

- Células simples: Tienen regiones excitadoras e inhibitorias, ambas formando patrones elementales alargados en una dirección, posición y tamaño en particular en cada célula. Si un estímulo visual llega a la célula con la misma orientación y posición, de tal manera que ésta se alinea perfectamente con los patrones creados por las regiones excitadoras y al mismo tiempo se evita activar las regiones inhibitorias, la célula es activada y emite una señal.
- Células complejas: Operan de una manera similar. Como las células simples, éstas tienen una orientación particular sobre la cual son sensibles. Sin embargo, éstas no tienen sensibilidad a la posición.

Por ello, un estímulo visual necesita llegar únicamente en la orientación correcta para que esta célula sea activada.

Cabe destacar la estructura que las células forman en la corteza visual. Existe una jerarquía de manera que dicha corteza comienza por la región V1 pasando por V2 hasta llegar a IT, suponiendo el paso de una región a la siguiente un aumento en la complejidad de las características percibidas. Gracias a la alternación entre células simples y complejas, la activación de las células cada vez depende menos de la posición y tamaño de los estímulos iniciales.

2.4.2 Estructura de la red

Una red neuronal convolucional es una red multicapa en la que se pueden diferenciar dos fases:

Una primera fase de extracción de características que consta de capas convolucionales y de reducción alternadas. Según progresan los datos a lo largo de esta fase, se disminuye su dimensionalidad, siendo las neuronas en capas lejanas mucho menos sensibles a perturbaciones en los datos de entrada, pero al mismo tiempo siendo estas activadas por características cada vez más complejas. En esta fase, las neuronas artificiales tal y como las conocemos son sustituidas por neuronas más complejas que realmente son matrices con las que se hacen operaciones sobre los datos en 2D (p.e. imágenes).

La segunda fase se encarga de realizar la clasificación y se compone por neuronas de perceptrón más sencillas formando una red neuronal tradicional. Esta clasificación es de vital importancia ya que es la salida o respuesta final de nuestra red basada en las características extraídas de los datos de entrada en la fase anterior.

2.4.2.1 Capas convolucionales

La salida de cada neurona convolucional se puede expresar como:

$$Y_j = g\left(b_j + \sum_i K_{ij} \otimes Y_i\right)$$

Donde la salida de una neurona Yj de una neurona j es una matriz que se calcula por medio de la combinación lineal de las salidas Yi de las neuronas en la capa anterior cada una de ellas operadas con el núcleo de convolucional Kij correspondiente a esa conexión. Esta cantidad es sumada a una influencia b_j y luego se pasa por una función de activación g (·).

La convolución tiene el efecto de filtrar la imagen de entrada con un núcleo previamente entrenado. Cuando se habla de imagen de entrada nos referimos a la matriz de números que la define y con núcleo nos referimos a la matriz que define la función que realiza la neurona convolucional. Esto transforma los datos de tal manera que ciertas características (determinadas por la forma del núcleo) se vuelven más dominantes en la imagen de salida al tener estas un valor numérico más alto asignados a los pixeles que las representan. Estos núcleos tienen habilidades de procesamiento de imágenes específicas, como por ejemplo la detección de bordes. Sin embargo, los núcleos que son entrenados por una red neuronal convolucional generalmente son más complejos para poder extraer otras características más abstractas y no triviales.

2.4.2.2 Capas de reducción o muestreo

En este tipo de capas se realiza una reducción por muestreo del tamaño de los datos de entrada. La reducción se orienta de forma se eliminen zonas poco útiles destacando las que contienen las características buscadas. Al reducir la resolución, las mismas características corresponderán a un mayor campo de activación en la imagen de entrada. Con esta operación se consigue dotar a la red de cierta tolerancia ante pequeñas pertubaciones en los datos de entrada e invarianza ante traslaciones de los elementos característicos en la imagen.

Hoy en día, en la mayoría de los casos esta reducción se realiza mediante la operación de **Max-Pooling**, muy eficaz resumiendo características sobre una región. Esta operación encuentra el valor máximo entre una ventana de muestra y pasa este valor como resumen de características sobre esa área. Como resultado, el tamaño de los datos se reduce por un factor igual al tamaño de la ventana de muestra sobre la cual se opera. Además, existe

evidencia que este tipo de operación es similar a como la corteza visual puede resumir información internamente.

2.4.3 Ejemplo de funcionamiento

Preliminalmente consideramos que nuestra red va a analizar una imagen de tamaño 28x28 y en escala de grises. Las imágenes suelen venir con valores en sus píxeles acotados entre 0 y 255, estos valores habrá que normalizarlos de manera que queden entre 0 y 1 por facilidad de procesamiento para la red.

2.4.3.1 Primera convolución

Tal y como previamente se ha explicado, una convolución consiste en ir operando matemáticamente (producto escalar) una imagen mediante una pequeña matriz denominada kernel, filtro o núcleo (diferentes nombres para un mismo concepto). Este kernel recorre todos los píxeles que componen la matriz de la imagen y genera una matriz de salida del mismo tamaño de la imagen inicial. El kernel tendrá inicialmente valores aleatorios que se irán ajustando a su cometido mediante backpropagation. Cabe destacar que, si la imagen fuera a color, el kernel debería tener también 3 capas, es decir, un filtro de 3 kernels de tamaño, por ejemplo, 3x3.

En la siguiente ilustración se puede visualizar la operación de convolución en matrices:

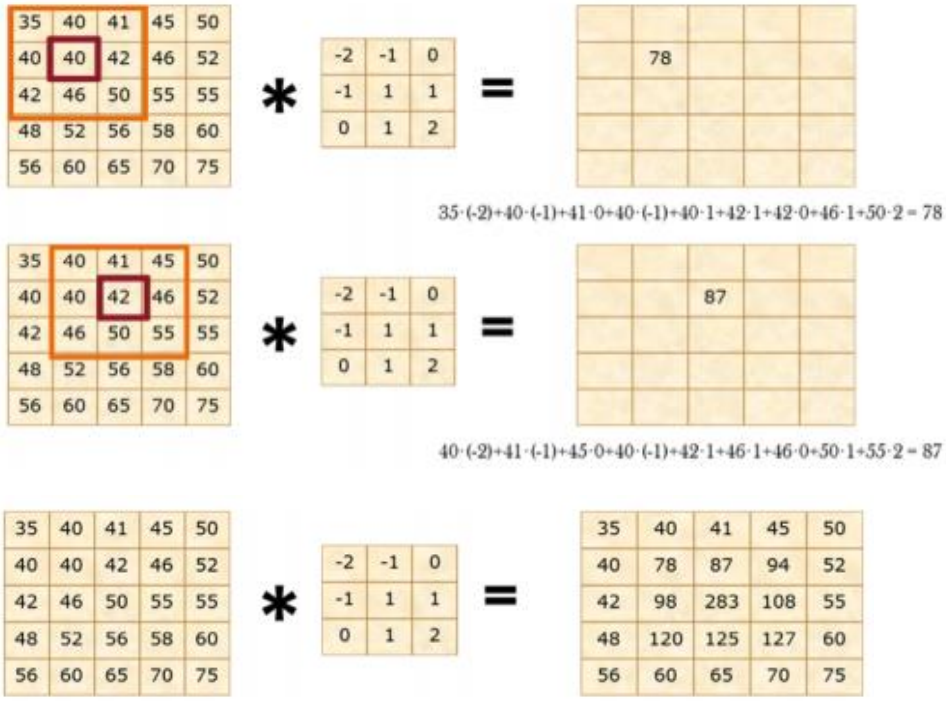

Figura 2-13. Ejemplo de convolución

La matriz que se genera se denomina mapa de características (o feature map). La operación de convolución se podría decir que "dibuja" ciertas características de la imagen original en una nueva, lo que ayudará a identificarlas en capas siguientes.

En la capa de convolución se suelen usar varios filtros kernel de manera que obtenemos tantas nuevas imágenes filtradas como filtros kernel hayamos usado. Tras la convolución se suele aplicar una función de activación sobre la imagen obtenida, en la mayoría de los casos será la función ReLU explicada en apartados anteriores.

Recapitulando el ejemplo, podemos suponer que hemos aplicados 32 fitros kernel de 3x3 sobre una imagen de 32x32 en escala de grises por lo que ahora tenemos 32 nuevas imágenes de tamaño 32x32.

Padding

Padding es un método que se usa a la hora de realizar las convoluciones sobre imágenes de manera que se rellena

con píxeles de valor cero alrededor de la imagen original.

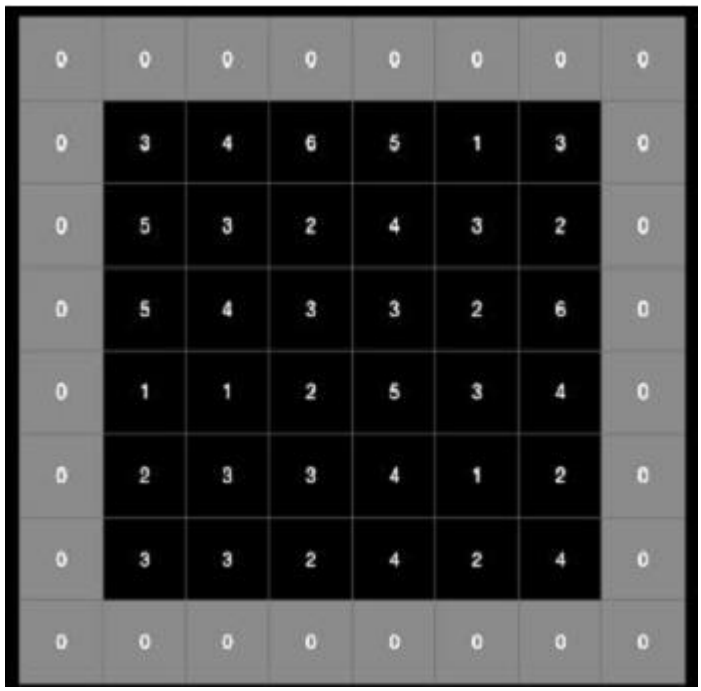

Figura 2-14. Padding (BootCampAI.Medium.com)

Esto se realiza con dos cometidos principalmente: permitir que la imagen generada mediante la convolución sea de igual tamaño que la imagen convolucionada y tener la información más relevante cerca del centro ya que, si esto no se hiciera, la información contenida en las esquinas de la imagen tendría menos influencia en la imagen generada porque al realizarse una convolución el filtro pasa más por el centro que por las esquinas.

2.4.3.2 Primera reducción mediante Max-Pooling

A este paso se le denomina de diferentes formas en el mundillo de la IA: reducción, muestreo, subsampling, etc. La idea es sencilla tal y como se ha explicado, se toma una matriz de cierto tamaño, por ejemplo, 2x2 y se va aplicando por todos los píxeles que componen a cada una de nuestras imágenes. Esta matriz se queda con el valor máximo de la región analizada y lo escribe sobre una nueva imagen. En la siguiente figura se visualiza con claridad la operación:

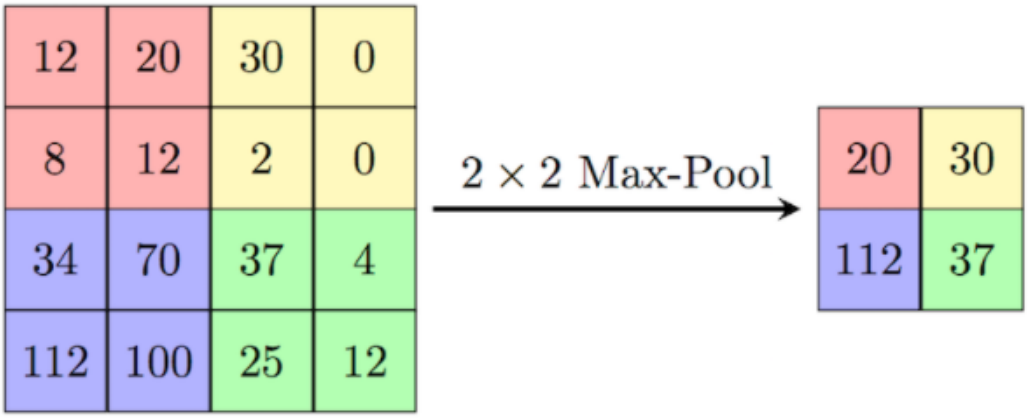

Figura 2-15. Ejemplo Max-Pooling (ComputerScienceWiki.org)

La siguiente imagen es trivial, pero quizás es útil para visualizar a grandes rasgos lo que se persigue. Al inicio se tenían 64 imágenes de tamaño 224x224 cada una y tras la reducción su tamaño será la mitad.

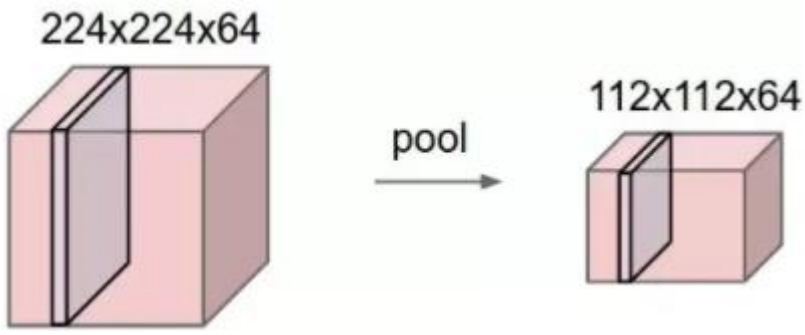

Figura 2-16. Ejemplo Max-Pooling sobre varias imágenes (ComputerScienceWiki.org)

Este paso es vital en las CNNs ya que a la vez que reducimos el tamaño de los datos que hemos de analizar, vamos resaltando las características dibujadas por los kernel en la convolución gracias a la forma en el que la técnica Max-Pooling reduce. En concreto, se consigue que cada vez se analice en mayor profundidad la imagen de manera que se van extrañendo características más complejas en los próximos pasos.

En nuestro ejemplo, esta reducción hubiera implicado pasar de tener 32 imágenes de tamaño 28x28 a que éstas fueran de tamaño 14x14.

2.4.3.3 Siguientes convoluciones

El proceso que hemos descrito en esta primera parte se denomina convolución (convolución+reducción). La primera convolución es capaz de detectar características primitivas como lineas ó curvas. A medida que hagamos más capas con las convoluciones, los mapas de características serán capaces de reconocer formas más complejas, y el conjunto total de capas de convoluciones podrá "ver".

2.4.3.4 RNA tradicional para clasificación final

Una vez que se ha terminado la fase de extracción de características tras las múltiples convoluciones, hemos de aplanar(flatten) los datos obtenidos en dicha fase. Esto supone pasar de tener, por ejemplo, un conjunto de matrices 3x3x128 a un vector de 1.152 elementos. Con este vector se alimentará a la red neuronal final que realiza la clasificación de las fotos.

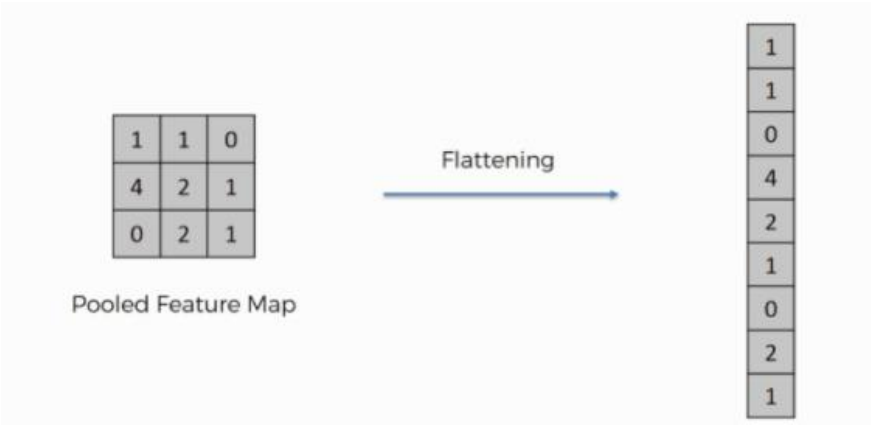

Figura 2-17. Flattening (SuperDataScience.com)

En la red neuronal se procesa el vector proporcionado a la entrada, de manera que nos expone ciertos resultados en su capa final. Evidentemente, esta capa final tendrá tantas neuronas como número de clases entre las que elegir haya en nuestro caso de clasificación. Si clasificamos perros y gatos, serán 2 neuronas. Si clasificamos coches, aviones ó barcos serán 3, etc.

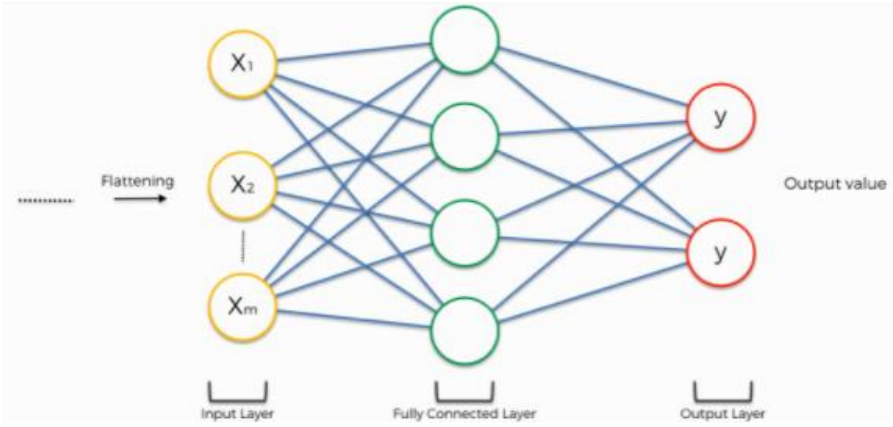

Figura 2-18. RNA de clasificación

Aplicamos la función "Softmax" sobre dichos resultados y nos proporcionará las probabilidades de que la imagen pertenezca a una categoría u otra. Por ejemplo, en un caso con 2 categorías una salida [0,2 0,8] nos indica 20% probabilidades de que sea una y 80% la otra.

Las salidas se proporcionarán en formato **One-hot-encoding** (anteriormente explicado) durante el entrenamiento. Por ejemplo, para coches, aviones ó barcos será [1,0,0]; [0,1,0]; [0,0,1]. La codificación One-hot es una forma de identificar las clases mediante un grupo de bits en la que hay un bit a 1 y todos los demás a 0. Este método se usa para evitar la relación de orden que conllevaría identificar nuestras categorías mediante enteros. En otras palabras, la codificación One-Hot nos permite asegurar que todas las categorías tienen igual valor y los errores de una clasificación errónea dentro de una categoría que no corresponde tengan el mismo valor de pérdida.

2.4.4 Entrenamiento

El proceso de entrenamiento de las CNNs es muy similar al usado para las redes tradicionales en las que tenemos una entrada y su respuesta deseada, y usando el algoritmo de Back-Propagation vamos reajustando el valor de los parámetros libres (pesos y sesgos) hasta conseguir el comportamiento perseguido. La fase de clasificación de las CNNs usa este método al ser una RNA tradicional.

En la fase de extracción de características es donde las CNNs varían algo respecto a lo anterior. En este caso, lo que se ajusta son los valores que componen cada filtro kernel que forman las capas de convolución. Esto presenta un punto a favor de este tipo de redes ya que los filtros kernel suelen tener un tamaño pequeño y, por lo tanto, el número de valores a ajustar será relativamente reducido. En nuestro ejemplo, tenemos 32 filtros kernel de tamaño 3x3 en la primera capa de convolución lo que significa 9 parámetros por filtro y 288 en la capa entera.

Si se compara con lo que se necesitaría para una red tradicional, teniendo en cuenta que nuestra imagen de entrada era de tamaño 28x28 (784 elementos) y el tamaño de la segunda capa creada en la CNN es 14x14x32(6272 elementos), necesitaríamos prácticamente 5 millones de valores intercapa para unirlas todas. Lo anteriormente expuesto supondría un incuestionable alto coste computacional y eso que estamos considerando una imagen relativamente pequeña y en escala de grises.

2.4.5 Conclusión

Se puede decir que las CNNs suponen un gran avance en el estudio de todo tipo de datos donde estén distribuidos de una forma continua a lo largo del mapa de entrada, y a su vez sean estadísticamente similares en cualquier lugar de ese mapa. Por ello, son especialmente eficaces clasificando imágenes. También pueden ser usadas para la clasificación de series de tiempo o señales de audio usando convoluciones en 1D, o para clasificar datos volumétricos usando convoluciones en 3D.

Las operaciones que con tanta eficiencia realizan las CNNs serían muy difícil de imitar por redes tradicionales dado el alto coste computacional anteriormente expuesto que supondrían.

3 VEHÍCULO DISEÑADO

En este apartado se va a tratar el proceso de construcción y configuración del vehículo que vamos a usar como plataforma de pruebas para el desarrollo de nuestra red neuronal. Este coche evidentemente deberá llevar incorporado cierto hardware que nos permita darle instrucciones de movimiento y una cámara para tomar las fotos que analizará el algoritmo de control creado.

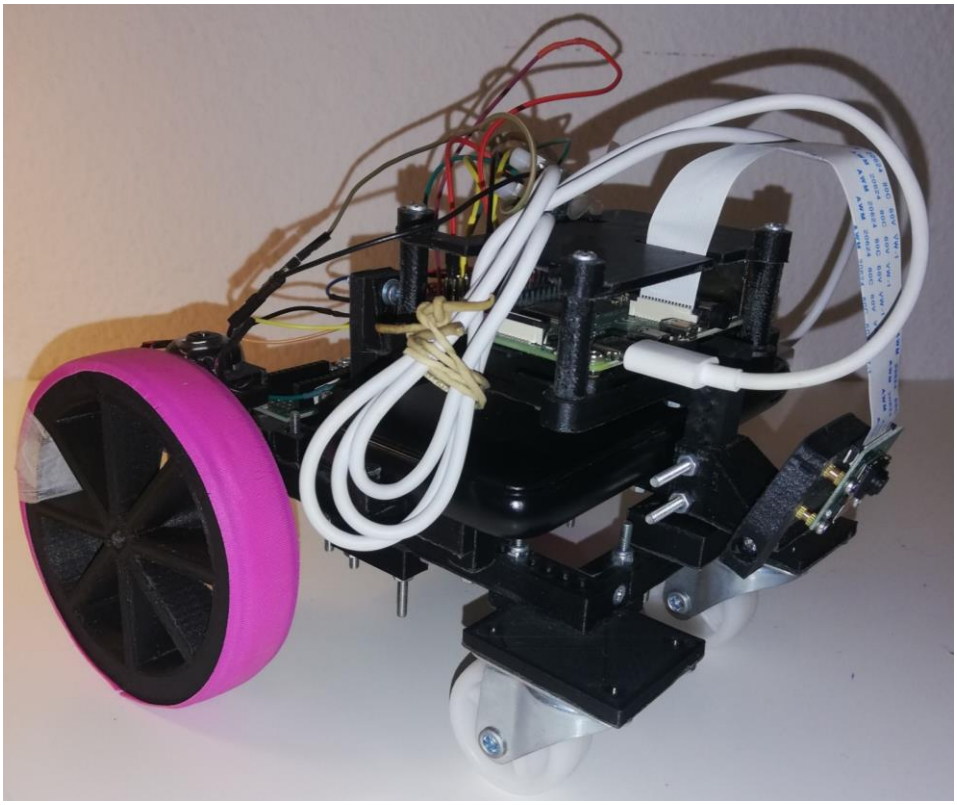

Figura 3-1. Aspecto final de vehículo diseñado 1

3.1. Construcción del coche

Se decide construir un pequeño coche completamente desde 0 por las ventajas que esto nos proporciona a la hora de poder decidir cada una de las características que éste tendrá. Se puede separar este proceso de construcción en dos partes: la parte de hardware para dotarlo de capacidad de movimiento y la parte de la estructura que le da forma al vehículo.

3.1.1 Hardware

La elección del hardware se hace con la principal idea de que éste proporcione el suficiente potencial de desarrollo tanto para el proyecto que se realiza como para tener margen para futuros experimentos que se deseen realizar sobre el tema.

El hardware que compone nuestro vehículo es:

3.1.1.1 Raspberry Pi 4 Modelo B

Raspberry Pi (RP) es básicamente un ordenador de bajo coste integrado en una placa de pequeño tamaño. Tiene la potencia suficiente para programar y compilar programas de manera que se ejecuten en ella. Su sistema

operativo oficial es de código abierto, llamado Raspbian, y también se pueden instalar otros SO como Linux o Windows ya que hay versiones específicas para la RP.

En nuestro trabajo tiene el papel de cerebro en nuestro coche ya que será la encargada de dar órdenes a los motores, hacer fotos, comunicarse con el PC, etc. Siempre que se plantea un proyecto con esta temática se plantea la duda de si usar una placa de Arduino o Raspberry Pi. En nuestro caso, se tomó esta decisión de utilizar la RP debido a razones como: proporciona una potencia de cálculo mayor a Arduino, cuenta ya de por sí con conectividad WiFi, etc.

Figura 3-2. Raspberry Pi 4 Model B

En concreto, algunas de las características de la Raspberry Pi 4 Modelo B 4GB usada son las siguientes:

- Procesador con velocidad de 1.5GHz
- 4GB de memoria RAM
- 2 salidas micro - HDMI
- Conectividad WiFi y Bluetooth
- 1 puerto para cámara Raspberry Pi
- 1 puerto para pantalla Raspberry Pi
- 4 puertos USB
- 40 pines de salida/entrada que pueden ser configurados en diferentes modos por el usuario.
- 1 puerto para SD Card

Tal y como reflejan sus características, la RP se presenta como la plataforma ideal sobre la que asentar nuestro trabajo ya que tiene capacidad para acogerlo tanto a éste como a futuras mejoras que se planteen. La idea final del proyecto en sí, más allá de este trabajo, es poder ejecutar la red neuronal que da órdenes de movimiento en la misma RP y es evidente que la RP tiene capacidad para ello.

3.1.1.2 Integrado L293D

El circuito integrado L293D es un puente H de tipo cuádruple de alta corriente el cual está diseñado para hacer girar dos motores de corriente continua en ambos sentidos o de forma bidireccional, es decir: en modo avance y retroceso. Los motores que puede hacer mover este circuito integrado van desde los 600 mA con voltajes de alimentación desde los 4.5 a 36V.

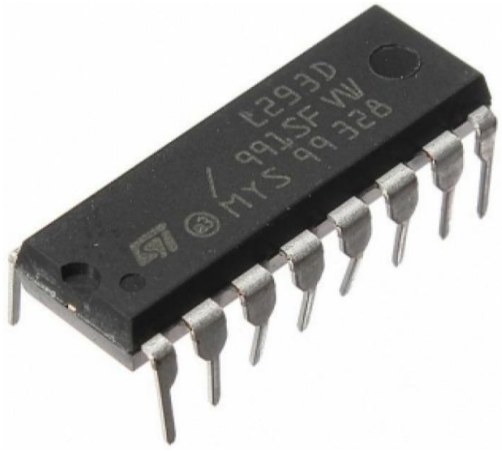

Figura 3-3. Circuito Integrado L293D

Este dispositivo es usado para controlar nuestros motores como paso intermedio entre la RP y los motores. El montaje y manejo será explicado a continuación.

3.1.1.3 Motores CC

Para el trabajo se han utilizado dos motores CC, uno para cada rueda trasera, como el que aparece en la siguiente figura. Tienen un rango de trabajo de 3 a 6 voltios.

Figura 3-4. Motor CC

Este modelo de motor de corriente continua es muy común en proyectos de este tipo por su buen rendimiento y precio asequible.

3.1.1.4 Batería

La batería elegida es la TOPK l1016 model que tiene una capacidad de 10.000mAh. Tiene dos salidas USB que proporcionan una tensión de 5V a 2.1A de máximo cada una.

A priori, la batería tiene unas características correctas para nuestro proyecto teniendo las dos fuentes de alimentación que necesitamos para nuestro coche, una para la RP y otra para los motores. Cabe descatar que, quizás, como mejora futura se podría plantear tener una fuente de alimentación independiente para los motores que proporcionara una tensión mayor a 5V, ya que hay momentos en los que estos 5V se ven algo justos para mover a nuestro vehículo.

3.1.1.5 Cámara

Lá cámara usada como medio de percepción en nuestro trabajo es una cámara oficial para Raspberry Pi de la marca LABISTS que tiene una resolución de 5MP pudiendo grabar vídeos a 1080p o sacar imágenes estáticas de hasta 2592x1944 píxeles.

Figura 3-5. Cámara Oficial para Raspberry Pi

La cámara elegida tiene unas características más que suficientes para el cometido del proyecto. Se podría haber usado cualquier otra cámara con una salida USB para conectarla a la RP. El motivo de la elección de ésta fue la comodidad que nos aportaba teniendo un puerto ya pensado para ella en la placa RP, el precio, sus características a nivel de resolución, poseer una librería definida para ella, etc.

Como resumen de la idea con la que se ha elegido cada uno de los componentes que conforman el hardware de nuestro vehículo, se puede decir que se ha buscado tener margen para mejoras futuras en cada uno de los elementos que definen a nuestro vehículo de manera que este trabajo sea tan solo el inicio de investigaciones futuras.

3.1.2 Estructura del vehículo

Tras buscar e incluso testar algún chasis de los disponibles en Internet que podría habernos servido, se llega a la conclusión de que la mejor opción es el diseño e impresión 3D de piezas para crear justamente el coche que necesitamos. La impresión de piezas en 3D nos proporciona una versatilidad prácticamente infinita a la hora de realizar modificaciones y posibles mejoras en el vehículo.

La idea con la que se construye la estructura es que sea completamente modular. El “esqueleto” del coche lo conforma este elemento rectangular con agujeros buscando que se puedan atornillar piezas a él con facilidad:

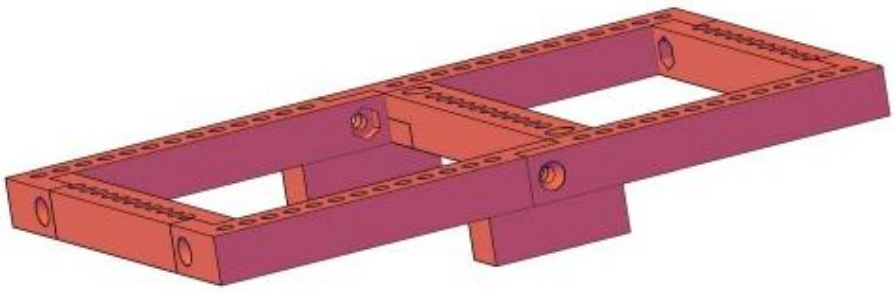

Figura 3-6. Chasis del coche

Partiendo sobre esta base, se crean las diferentes piezas necesarias para conformar nuestro vehículo, siendo las principales:

- Ruedas traseras, soporte para motores y pieza para unir las ruedas delanteras a la estructura:

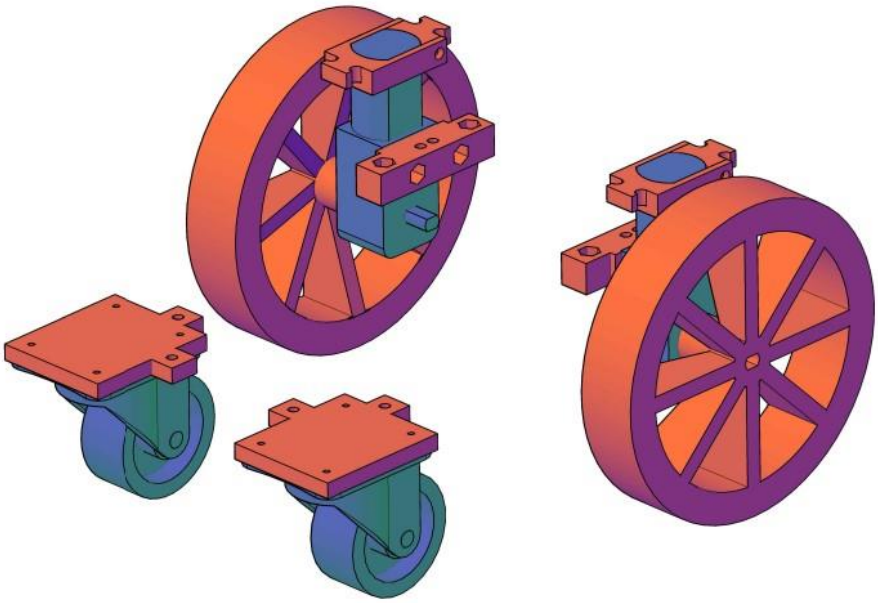

Figura 3-7. Piezas para movilidad del vehículo.

Como aclaración, las piezas en azul no han sido impresas (ruedas delanteras y motores) ya que son adquiridas externamente.

- Soporte para batería y Raspberry Pi:

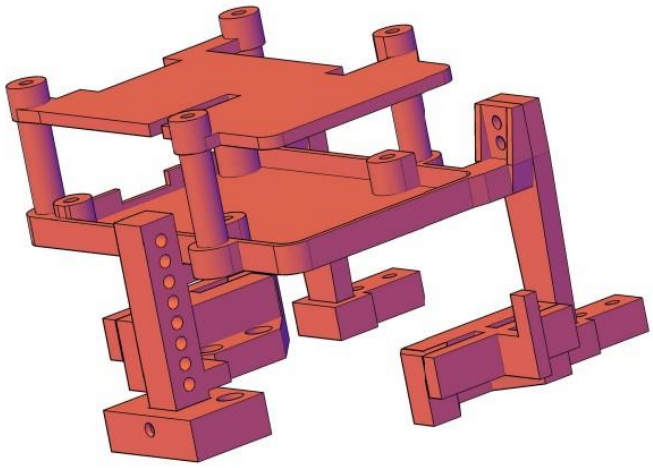

Figura 3-8. Soporte batería y Raspberry Pi

- Soporte para la cámara:

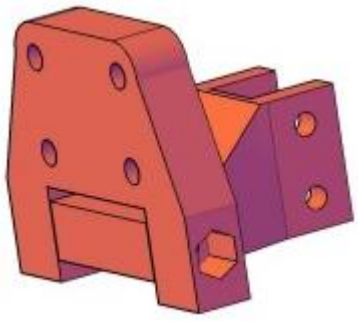

Figura 3-9. Soporte cámara

Cabe destacar que es regulable la orientación de la cámara, de manera que se ajuste a lo posición óptima.

Finalmente uniendo todo lo anterior, el coche tiene una apariencia tal que así:

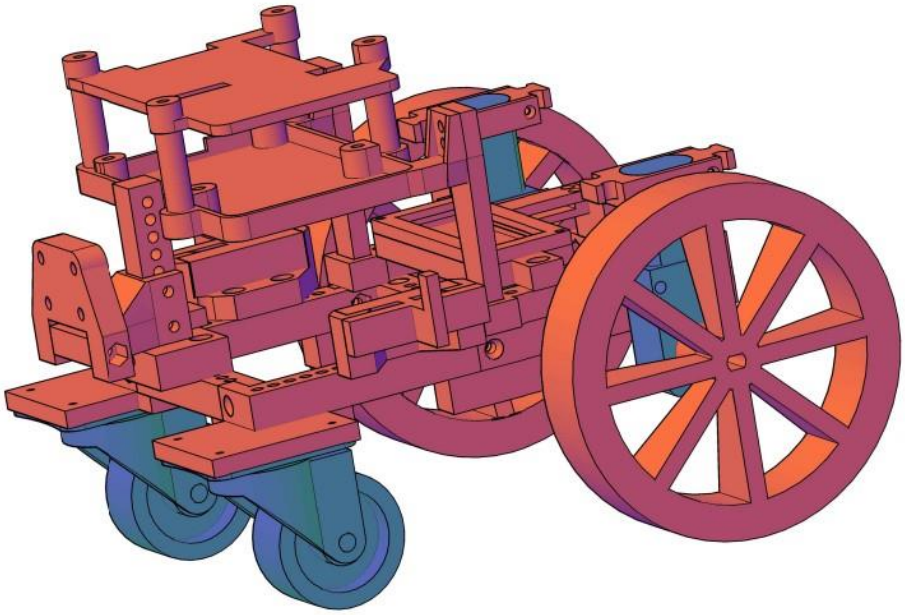

Figura 3-10. Estructura final de vehículo

Tras la impresión de las piezas y montaje completo del coche, este es el resultado:

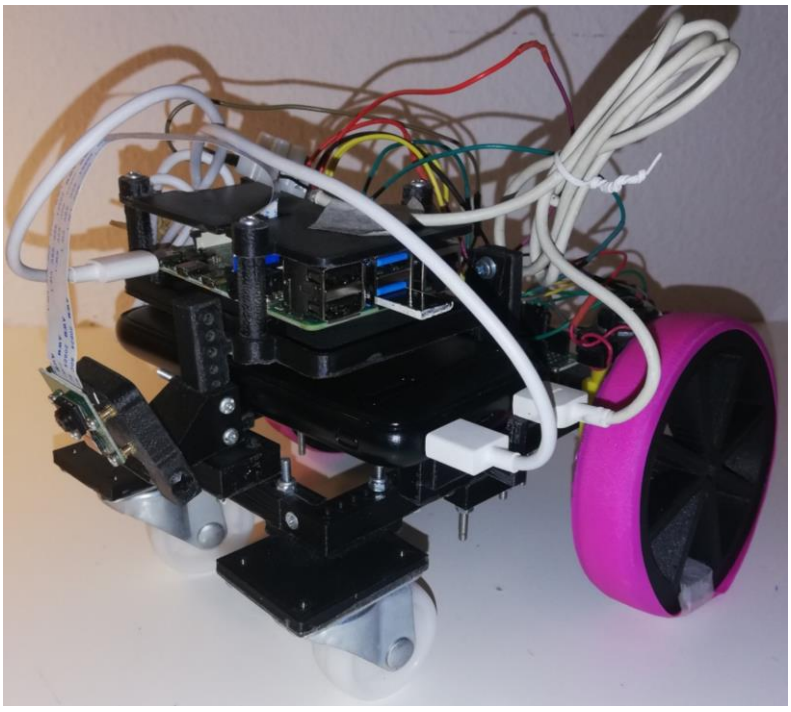

Figura 3-11. Aspecto final del vehículo diseñado 2

Se puede decir que la estructura diseñada para nuestro vehículo es un éxito ya que cumple con el objetivo inicial, se consigue un vehículo compacto que lleva a bordo todos los elementos necesarios para la autonomía y se mueve con agilidad. Como aspecto negativo se podrían mencionar los problemas de tracción que sufre el coche al concentrar todo su peso sobre el eje delantero que no es el que proporciona tracción, por lo que las ruedas del eje trasero resbalan con facilidad. Este problema se ha intentado paliar instalándole unas cintas de goma en las ruedas como “cubiertas” para tener un mayor agarre y cuando la superficie es muy deslizante se coloca algún elemento de peso sobre el eje trasero para mejorar la tracción.

3.2. Puesta en funcionamiento básico

Este puede considerarse un paso inicial clave para nuestro proyecto ya que, evidentemente, si lo que pretendemos en último lugar es controlar el coche mediante una red neuronal, necesitaremos saber cómo moverlo a través de nuestro PC. Serán formas análogas de darle órdenes, simplemente cambiará el origen de las órdenes de nuestro teclado por las predicciones hechas por la red. Además, es indispensable tener un programa que nos permita grabar vídeos desde el coche de una manera cómoda para facilitar la creación del dataset que se use en el entrenamiento de la red.

Cabe destacar cierta complejidad inicial a la hora de poder lograr una puesta en marcha básica de nuestro vehículo dándole órdenes desde nuestro PC.

3.2.1 Configuración inicial de Raspberry Pi

Instalamos Raspbian en nuestra Raspberry Pi (RP), sistema operativo más común para este dispositivo. Este SO nos proporciona prácticamente todas las funcionalidades que tiene cualquier ordenador hoy en día. Tiene un entorno intuitivo y que facilita el trabajo.

No hay que hacer mucho más en la RP, simplemente instalar cada una de las librerías que se vayan a usar en Python desde la ventana de comandos, ya que este es el lenguaje que vamos a usar durante todo el trabajo. En el menú de configuración, habrá que habilitar la cámara para que pueda sacar fotos y grabar vídeos.

Cada vez que accedamos a la RP por primera vez tras encenderla, hemos de dar la orden "vncserver" para que se ponga en modo servidor de VNC, de forma que podamos ver el escritorio de la RP desde el ordenador, tal y como se va a explicar ahora.

3.2.2 Configuración inicial de PC

Para el uso diario de la RP, utilizamos principalmente tres programas en nuestro PC:

- Nmap – Zenmap GUI: Este programa escanea la dirección IP que le introduzcamos y nos dice los dispositivos conectados a ella y sus respectivas IPs.

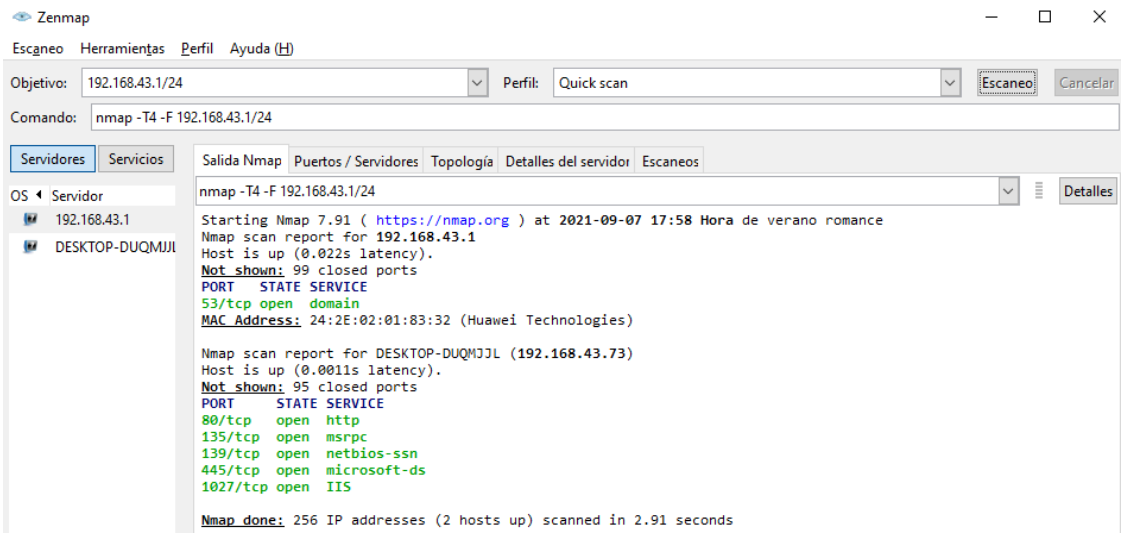

Figura 3-12. Nmap – Zenmap GUI

- PuTTY: Nos permite realizar una conexión tipo SSH de manera que nos da acceso a la ventana de comandos de la RP cuando se conoce su IP.

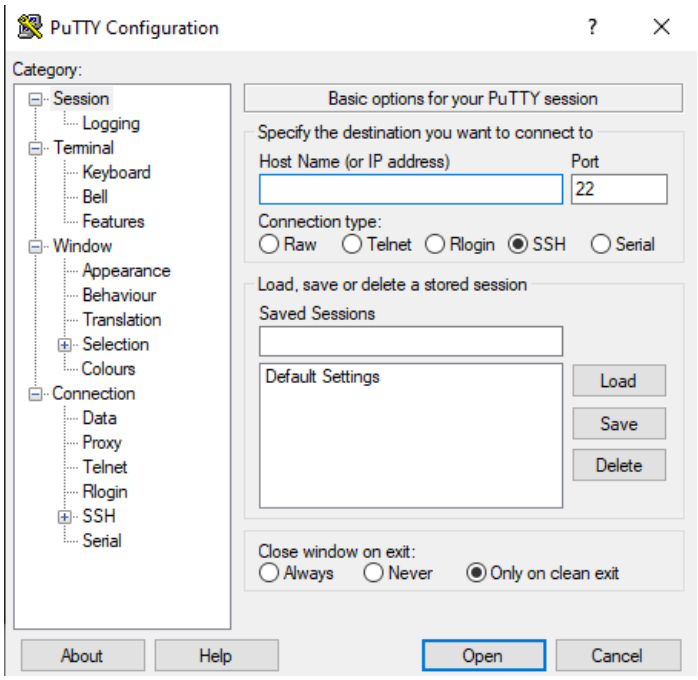

Figura 3-13. PuTTY

- VNC Viewer: Nos proporciona acceso al escritorio de la RP como si estuviéramos con una pantalla conectados a ella. Lo único que necesitamos es haber activado previamente el modo "vncserver" en la RP, esto se hace desde la ventana de comandos abierta en PuTTY.

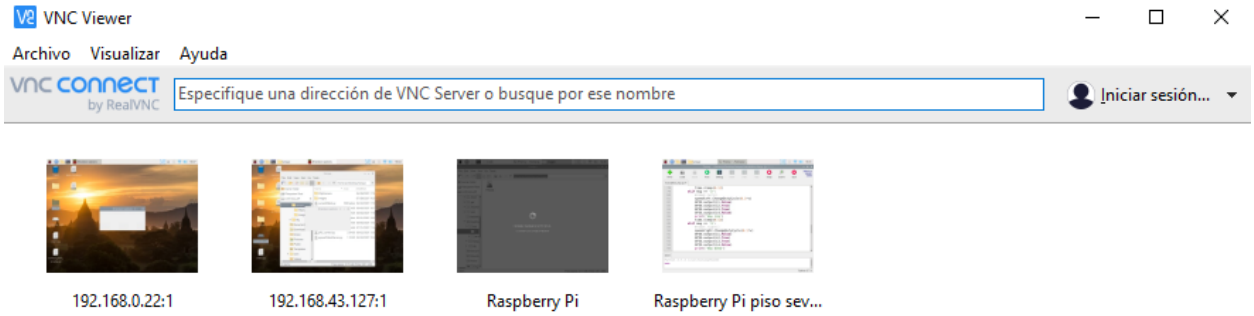

Figura 3-14. VNC Viewer

En resumen, los pasos a seguir serían escanear la IP de nuestra red de manera que conozcamos la IP de la RP (desde Nmap), comenzamos una conexión tipo SSH mediante la IP de la RP ya conocida desde PuTTY, la conexión SSH nos da acceso a la consola de comandos de la RP, ahí introducimos el comando "vncserver", vamos al programa VNC Viewer y con la IP de la RP ya podremos ver el escritorio de la RP como se vería desde una pantalla sólo para ella.

Es muy destacable la manera de interactuar con la RP que nos habilita el uso de estos tres programas ya que, de otro modo, deberíamos tener una pantalla independiente siempre que quisiéramos hacer cualquier acción sobre la RP. Con el método descrito podemos hacer todo desde nuestro PC.

Para la creación y ejecución de los programas Python en los que se basa el proyecto, usamos el paquete Anaconda que nos da acceso a estos dos entornos:

- Spyder: IDE interactivo para computación científica en Python. Este IDE permite escribir, editar y probar códigos ofreciendo visualización y edición de variables. Aporta un entorno muy intuitivo con diversas herramientas que facilitan la programación.

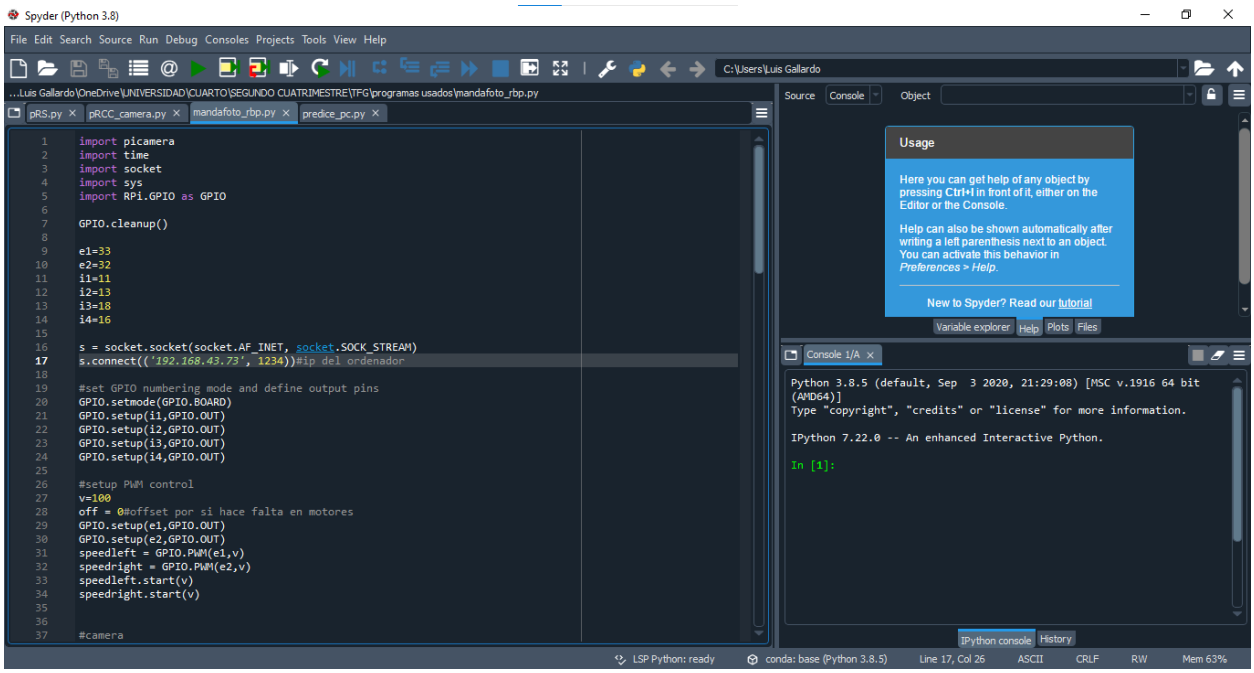

Figura 3-15. Spyder

- Anaconda Prompt: Consola de comandos desde la que se pueden ejecutar los programas de Python tal y como se haría desde la consola de comandos de Windows, con la diferencia de que esta Anaconda Prompt tiene ya preinstalada Python y una gran cantidad de librerías por lo que nos ahorramos mucho tiempo en configuración del sistema.

Figura 3-16. Anaconda Prompt

3.2.3 Circuitería implementada

En este apartado se va a explicar cómo se han realizado las conexiones entre los diferentes elementos de nuestro vehículo para poder manejar los motores.

Como aclaración, la Raspberry Pi será alimentada a través de su puerto USB tipo C mediante una de las dos salidas disponibles en nuestra batería mientras que los motores serán alimentados a través de la otra salida mediante un cable dividido en su positivo (rojo) y negativo (negro) cuyas conexiones serán aclaradas a continuación.

La siguiente imagen contiene un esquemático del integrado L293D en el que se da una primera aproximación a las conexiones que se realizan.

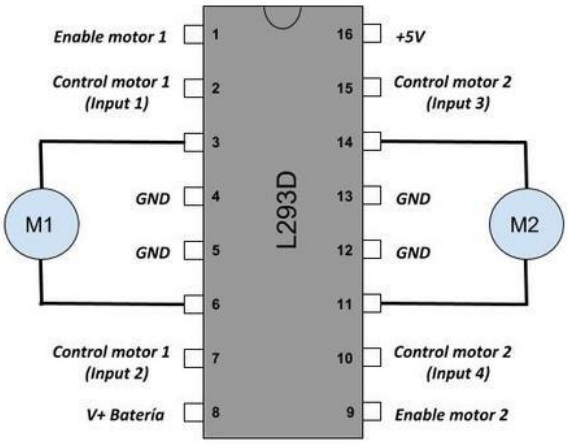

Figura 3-17. Esquemático del L293D (TalosElectronics.com)

A través de los pines "Enable" se controla la velocidad de cada uno de los motores ya que se usará una modulación de ancho de pulso (PWM) en la que se modifica la energía que se envía a la carga a través de una señal de voltaje, irá de 0 a 5V, siendo 5V la velocidad máxima del motor en la que se le proporcionará el máximo voltaje dado por la batería que puede ser otro a estos 5V. Esta modulación PWM se puede entender como una especie de "grifo" de voltaje.

Mediante los pines "Input" se puede definir el sentido de giro de cada uno de los motores. A estos pines se les proporcionará una señal de 0 o 5V, LOW o HIGH respectivamente en la siguiente tabla. Dependiendo del estado de cada entrada, el sentido de giro que tomará el motor será diferente tal y como podemos observar en la siguiente tabla de verdad:

Pin 2	Pin 7	Salida
LOW	LOW	Detenido
LOW	HIGH	Derecha
HIGH	LOW	Izquierda
HIGH	HIGH	Detenido

Figura 3-18. Tabla de verdad para Motor 1 (TalosElectronics.com)

A través del pin 16 de nuestro esquemático se proporcionará una señal de 5V desde nuestra Raspberry Pi. Esta señal se usará como referencia para que el L293D pueda conocer si los pines "Input" están en un estado u otro, y también como referencia para conocer el ancho de pulso transmitido a través de cada Enable.

El cable positivo de nuestra batería que proporcione alimentación a los motores deberá conectarse al pin 8 del L293D y el voltaje de éste podrá ser cualquiera dentro del rango que puedan soportar tanto nuestros motores (3-6V) como el L293D (4.5-36V). Mientras más voltaje se proporcione, mayor será la velocidad máxima que pueden alcanzar los motores.

En los pines output del integrado se conectarán los cables que alimentan los motores.

Una vez que se conoce el funcionamiento del integrado L293D, es fácil entender cómo se conecta la RP a él. Este es el esquema de los pines configurables por el usuario en nuestra Raspberry Pi:

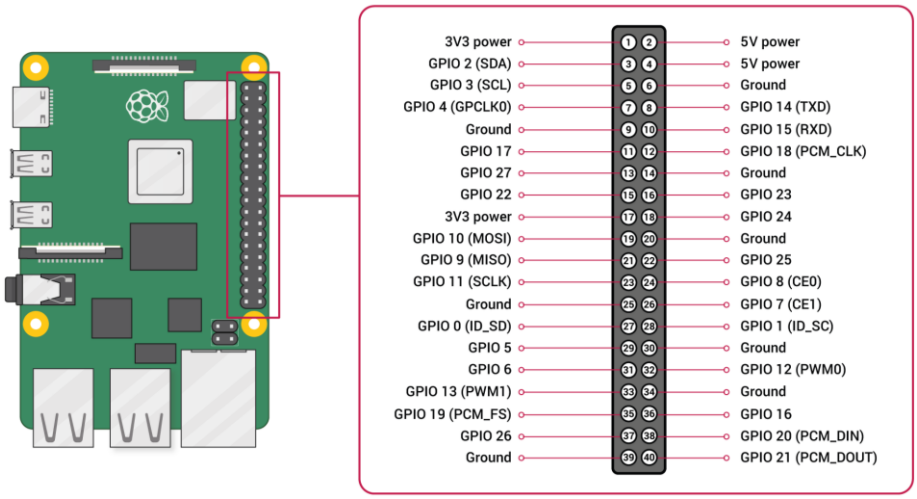

Figura 3-19. Esquema pines de Raspberry Pi (hwlibre.com)

Se puede ver que los GPIO 12 (pin 32) y 13 (pin 33) pueden ser configurados como PWM, éstos serán los usados para alimentar los "enable". En concreto, el pin 32 se conectará al enable 2, mientras que el 33 se conectará al enable 1.

Para los "input" puede usarse cualquier pin ya que simplemente reciben una señal LOW o HIGH. Para los inputs 1 y 2 se eligen los pines 11 y 13, mientras que para los inputs 4 y 3 se usan los pines 16 y 18 respectivamente.

La señal de referencia de 5V necesaria para el funcionamiento del L293D se proporcionará a través del pin 2 que, tal y como se ve en el esquemático, proporciona 5V.

Se ha creado esta figura en la que se detallan gráficamente las conexiones necesarias para este método de control de los motores:

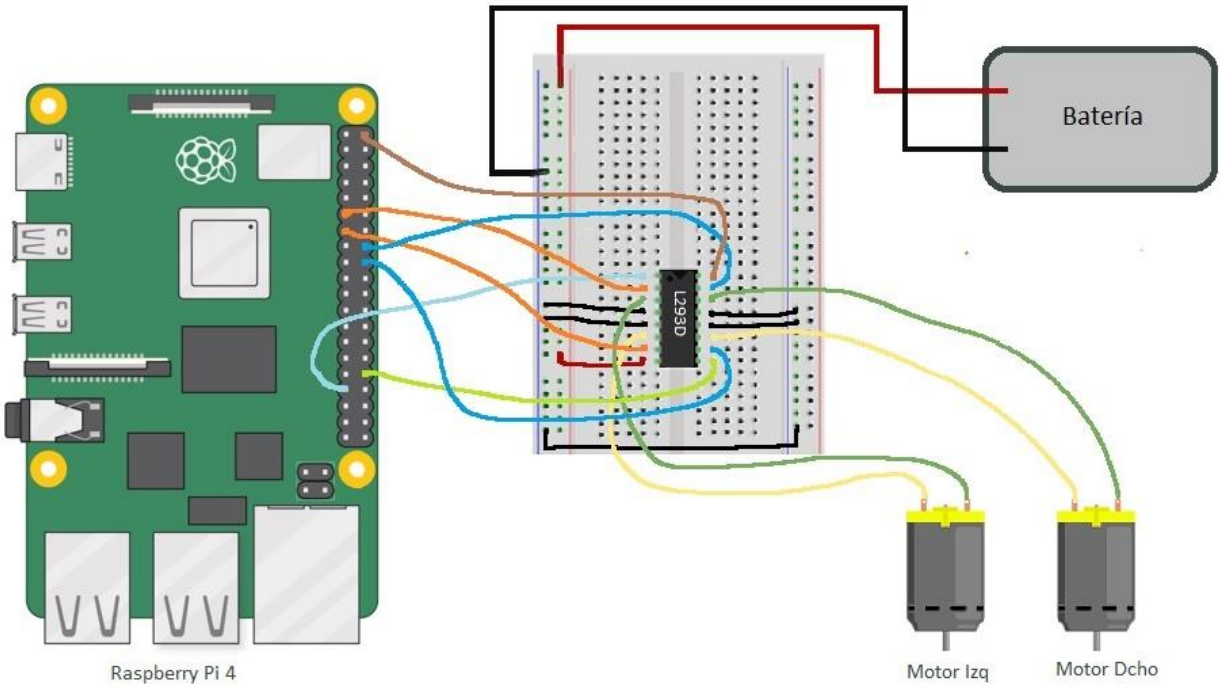

Figura 3-20. Circuitería del vehículo

3.2.4 Programas básicos de control del coche

Con dos programas conseguimos la comunicación de órdenes por parte del PC a la RP de manera que la RP es

capaz de moverse, pudiéndose variar su velocidad, grabar vídeos, etc.

3.2.4.1 Programa que manda órdenes desde PC

El código de programa se ha ubicado en el Anexo 1. Como se puede observar, se hace uso del protocolo TCP creando un socket de tipo conexión vinculado a la dirección IP del ordenador. Este socket de conexión se pone en modo escucha de manera que espera a que otro dispositivo (la RP) se conecte. Se crea una conexión en la que los dos mantienen una conversación hasta que se cierre por parte de uno de los dos u ocurra un error de conexión. Gracias a ello, ya tenemos una forma de enviar mensajes (órdenes) desde el PC hacia la RP y viceversa si fuera necesario.

Para leer cuando se pulsa el teclado del PC con el que pretendemos mover el coche, hacemos uso de la librería Pynput que permite monitorizar los inputs desde el teclado o ratón del PC. Se crea un programa que detecta si hay alguna teclada pulsada, mandando la orden correspondiente a la RP y, si no hay ninguna pulsada, se manda la orden de parar el coche. Si se pulsa la tecla Esc, se aborta la conexión y termina el programa.

3.2.4.2 Programa que ejecuta las órdenes en la RP

El código de programa se ha ubicado en el Anexo 2.

En primer lugar, se crea un socket que se conecta al creado para la IP del PC de manera que, desde ese momento, la conexión ya está en marcha entre PC y RP.

Se definen los pines que se usan en la RP para habilitar el movimiento del coche, tanto los enablers como los outputs para cada motor. Se hace su configuración inicial poniéndolos en modo Output y los enablers, que son por los que se marcará la velocidad del movimiento de cada rueda, se configuran en modo PWM. Todo ello se consigue mediante la librería RPi.GPIO creada justo con el propósito de poder manejar los pins de la RP.

La cámara, al estar concebida para RP, tiene una librería propia que nos facilita su uso. Se hace una preconfiguración de ésta de manera que giramos tanto por el borde ancho y alto la imagen que capta ya que la posición que tiene la cámara en el coche es totalmente del revés, hacemos esto para que las fotos se hagan como si la cámara estuviera derecha. Definimos la resolución y los fotogramas por segundo en los que graba (definidos de manera aleatoria) y parece que funciona bien para nuestro cometido.

Lo que es el núcleo del programa se basa en un bucle while que simplemente espera a recibir un mensaje por parte del PC, lo decodifica a números enteros y, según su contenido, mueve el robot de una manera u otra, o empieza o para de grabar vídeos, o sube o baja la velocidad, o termina el programa y la conexión.

4 PUESTA EN MARCHA DE RED NEURONAL

Una vez que tenemos claro los conceptos sobre redes neuronales y tenemos un vehículo al que sabemos darle instrucciones con el que podemos obtener fotos del entorno al que nos vamos a enfrentar, podemos crear la red neuronal que consideremos más apropiada para nuestro problema y comenzar a crear datasets y entrenarla para ver qué resultado obtenemos y si la idea inicial era adecuada.

4.1. Herramientas usadas

En este apartado se da información sobre las principales herramientas usadas para implementar la red neuronal.

4.1.1 TensorFlow

TensorFlow es una librería de código abierto orientada a la computación numérica mediante grafos de flujo de datos creada por Google y que permite un fácil procesamiento de matrices y tensores (2D), de ahí su nombre. En sus inicios, fue desarrollada por el equipo de Google Brain para uso interno en la compañía. La primera versión es de noviembre de 2015, no se liberó al público en general hasta febrero de 2017.

El inicio de Tensorflow fue facilitar la programación de redes neuronales dando acceso a una serie de librerías con funciones ya pre-hechas. Se puede usar dentro de cualquier programa desarrollado en Python (o C++), pero es de muy bajo nivel.

Si se pretende programar una nueva red neuronal (alguna que no esté inventada) o realizar cálculos muy costosos con matrices se deberá usar Tensorflow, pero si se quiere usar alguna red neuronal de las que ya existen, es mejor usar otras librerías más simples para su creación como Keras. En la mayoría de los casos, TensorFlow se usa como motor para ejecutar la red neuronal que hayamos diseñado mediante Keras y la entrenará.

Se da el hecho de que incluso se ha creado un nuevo hardware, por parte de Google, donde las operaciones de tensores más básicas estén implementadas directamente en el chip (TPUs), incrementando exponencialmente la rapidez de cálculo. Las aplicaciones de TensorFlow se pueden ejecutar en casi cualquier dispositivo que sea conveniente: una máquina local, un clúster en la nube, dispositivos IOS y Android, CPUs, GPUs o TPUs.

El software TensorFlow sigue actualizándose y se espera un gran crecimiento durante los próximos años. Se le considera uno de los protagonistas en el desarrollo del Machine Learning en el futuro. Muchas empresas de primer nivel lo utilizan para sus aspectos de investigación, como Bloomberg, Google, Intel, DeepMind, GE Health Care, eBay, entre otros. Está detrás de aplicaciones tan conocidas y ampliamente utilizadas como el traductor de Google, el Smart Reply de Gmail y también en aplicaciones de Airbnb, Uber o Twitter.

4.1.1.1 TensorFlow vs Pytorch

TensorFlow compite con un montón de otros marcos de Machine Learning. Librerías como Pytorch, Caffe, Theano o CNTK se consideran sus principales rivales, destacando Pytorch por el crecimiento en uso que está experimentando actualmente.

Cuando los investigadores quieren flexibilidad, capacidades de depuración y una formación de corta duración, eligen PyTorch. Funciona en Linux, macOS y Windows.

Gracias a su marco bien documentado y a la abundancia de modelos y tutoriales entrenados, TensorFlow es la herramienta favorita de muchos profesionales e investigadores de la industria. TensorFlow ofrece una mejor visualización, lo que permite a los desarrolladores depurar mejor y realizar un seguimiento del proceso de entrenamiento. PyTorch, sin embargo, solo proporciona una visualización limitada.

Por las razones ya explicadas y debido a que nuestro proyecto es una primera aproximación al Machine Learning, elegimos TensorFlow. La abundancia de documentación y modelos ha sido clave para el desarrollo de este trabajo.

4.1.2 Keras

Keras es una librería de alto nivel y código abierto que está diseñada para proveer una rápida implementación de redes neuronales profundas. Fue creada en 2015 por François Chollet, un ingeniero de Google. Esta librería se centra mucho en el minimalismo por el hecho de que puedes construir una red neuronal con muy pocas líneas de código. Se enfoca en ser modular, amigable al usuario y extensible. No permite computación a bajo nivel.

Keras necesita un backend, un motor computacional que corra debajo, como Tensorflow, Caffe, Theano o CNTK (y otros). Podríamos poner el símil de un programa PHP que debe acceder a una base de datos, el programa no cambiará por acceder a MySQL, PostgreSQL o SQL Server.

En 2017, Keras fue implementada en TensorFlow. Se puede acceder a ella mediante el módulo "tf.keras". En nuestro trabajo, las seguiremos considerando como librerías diferentes por dar claridad a los conceptos. TensorFlow y Keras se combinan casi a la perfección aunando potencia, sencillez de uso y rapidez de ejecución.

4.1.2.1 Primera aproximación a código

Con ayuda de este pequeño código se puede ilustrar la facilidad con la que se pueden crear redes neuronales mediante Keras:

```
from keras.models import Sequential
from keras.layers.core import Dense


model = Sequential()
model.add(Dense(8, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

Para tener acceso a las funciones usadas en el código anterior las tenemos que haber importado previamente, como se hace en el primer párrafo de código. Centrándonos en la red en sí (segundo párrafo):

En primer lugar, creamos un elemento de tipo "Sequential" al que denominamos "model", este elemento será nuestra red. El tipo "Sequential" implica que nuestra red tendrá una serie de capas secuenciales ("una delante de otra"), el tipo más común y que hemos visto.

En la segunda línea de código definimos las dos primeras capas de nuestra red: la capa de entrada que tendrá dos entradas ("input_dim=2") y la primera capa oculta que tendrá 8 neuronas. Se elige ReLU ("relu") como función de activación para estas capas. Estas capas se han definido de tipo "Dense", capas tradicionales compuestas por neuronas tal y como se ha venido explicando a lo largo del trabajo. Para llevar a cabo la adición de estas capas a nuestra red "model" hemos usado la función "add" proporcionada por la librería en cuestión, Keras.

Una vez se conoce lo anterior, es fácil intuir lo que se hace en la tercera y última línea de código: se añade una capa de tipo "Dense" con una sóla neurona, ésta será nuestra capa de salida y usará la función de activación "sigmoid".

En conclusión, en tres líneas de código somos capaces de definir la arquitectura de una red neuronal al completo.

4.1.3 Google Colab

Google Colab nació como un proyecto de investigación de Google creado para ayudar a difundir la educación e investigación sobre Machine Learning. Es una herramienta para ejecutar código Python y crear modelos de Machine Learning a través de la nube de Google, con la posibilidad de hacer uso de sus GPUs y TPUs. Es un entorno basado en Jupyter Notebook que no requiere configuración y se ejecuta completamente en la nube, lo que permite el uso de librerías como: Scikit-learn, PyTorch, TensorFlow, Keras y OpenCV. Es de uso totalmente gratuito, solo requiere una cuenta de Google.

La principal ventaja que ofrece esta herramienta es que libera a nuestra máquina de tener que llevar a cabo un

trabajo demasiado costoso tanto en tiempo como en potencia o incluso nos permite realizar ese trabajo si nuestra máquina no cuenta con recursos suficientemente potentes.

Otro de los beneficios que tiene lo indica el propio nombre, «Colaboratory», es decir, colaborativo, nos permite realizar tareas en la nube y compartir nuestros cuadernos si necesitamos trabajar en equipo. Todo lo que produzcas lo puedes almacenar directamente en Google Drive o en Github.

Aunque tiene algunas limitaciones que pueden consultarse en su página de FAQ, es una herramienta ideal, no sólo para practicar y mejorar nuestros conocimientos en técnicas y herramientas de Data Science, sino también para el desarrollo de aplicaciones de machine learning y deep learning, sin tener que invertir en recursos hardware o del Cloud.

Dado todo lo anterior, este entorno es el elegido para el diseño y entrenamiento de la red neuronal en la que se basa este proyecto.

4.2. Código implementado para la red

Una vez que conocemos toda la teoría sobre redes neuronales y tenemos todas las herramientas necesarias, sólo falta ponerlo en práctica. En este apartado vamos a describir la arquitectura de la red de tipo CNN (Convolutional Neural Network) que se usa en el trabajo para clasificar las fotos realizadas desde la Raspberry Pi.

Esta descripción se va a realizar directamente sobre el código con el que se hace funcionar la red en Google Colab mediante Keras y TensorFlow (Backend). La gran mayoría de los conceptos teóricos aplicados en la red han sido ya previamente explicados, el resto se detallarán a continuación cuando sea necesario.

La red explicada es la que corresponde a la CNN final elegida para dar instrucciones a nuestro coche. Durante el proceso de diseño se ha probado con ligeras variaciones de ésta que serán nombradas, la descrita es la que mejor resultados ha proporcionado.

Importar librerías

En primer lugar, como en cualquier programa en Python, se comienza importando las librerías que vamos a usar en nuestro código:

```
import numpy as np
import os
import re
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

import keras
from keras.utils.np_utils import to_categorical
from keras.models import Sequential,Input,Model
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.layers.normalization import batch_normalization
from keras.layers.advanced_activations import LeakyReLU
```

Esta podría ser una pequeña explicación de la función para la que se usa cada una de las librerías importadas:

- Numpy: Librería de Python que ofrece funciones matemáticas de alto nivel para trabajar con vectores, matrices o cubos (objetos de la clase array) de una forma eficiente. Un array es una clase de objeto que permite representar datos de un mismo tipo en varias dimensiones.

  Para una lista de valores se crea un array de una dimensión, también conocido como vector. Para una lista de listas de valores se crea un array de dos dimensiones, también conocido como matriz. Para una lista de listas de listas de valores se crea un array de tres dimensiones, también conocido como cubo. Y así sucesivamente. No hay límite en el número de dimensiones del array más allá de la memoria disponible en

el sistema.

- os: Módulo de Python que nos proporciona una manera versátil de usar funcionalidades dependientes del sistema operativo como leer o escribir en un archivo '`open`', manipular rutas '`os.path`', etc.
- re: Módulo de Python que proporciona operaciones de coincidencia de expresiones regulares. En nuestro programa sólo es usado en la búsqueda de las fotos para el dataset.
- Sklearn: Su nombre completo es Scikit-learn, es una librería de código abierto de Python que nos da acceso a una gran variedad de algoritmos diseñados para el aprendizaje automático como el preprocesamiento de datasets, validación cruzada, etc. En concreto, para el proyecto se importan las funciones: '`sklearn.model_selection.train_test_split`' que nos permite dividir un dataset en dos bloques, normalmente serán los destinados al entrenamiento y a la validación, y '`sklearn.metrics.classification_report`' que genera un informe en el que se muestran los principales indicadores estadísticos del entrenamiento realizado para poder realizar una fácil evaluación de éste.

Cargar dataset

A la hora de ejecutarlo, cabe destacar que hemos de estar en el directorio que contiene a la carpeta en la que esté nuestro dataset ya que, de otra forma, no funcionaría el siguiente fragmento de código. Preliminarmente se aclara que nuestro dataset en sí es una carpeta que contiene cierto número de subcarpetas en su interior cuyos nombres son los nombres de las clases en las que queremos clasificar imágenes y que estas subcarpetas contienen las imágenes que componen a nuestro dataset.

Así quedaría el código para la carga del dataset:

```
dirname = os.path.join(os.getcwd(), 'dataset')
#obtenemos la ruta a la carpeta del dataset
imgpath = dirname + os.sep
#le añade separador a la ruta, quedaría 'rutadataset/'

images = []
directories = []
dircount = []
prevRoot=''
cant=0
i=1

print("leyendo imágenes de ",imgpath)

for root, dirnames, filenames in os.walk(imgpath):
#os.walk genera toda esta información sobre el directorio que apunta al dataset:
#todo lo siguiente son listas
#rutas de las subcarpetas que hay en carpeta analizada(root)
#nombres de las subcarpetas dentro de la carpeta(dirnames)
#nombres de los archivos dentro de las subcarpetas(filenames)

    for filename in filenames:#se recorren los filenames para cada root
        if re.search("\.(jpg|jpeg|png|bmp|tiff)$", filename):
        #busca las imágenes anteriormente identificadas
            cant=cant+1#crea una cuenta para saber cuantas imágenes hay por clase
            filepath = os.path.join(root, filename)#ruta a la imagen
            image = plt.imread(filepath)#devuelve lista con la imagen
```

```
            images.append(image)#suma la imagen a la lista de las imágenes
            b = "Leyendo..." + str(cant)
            print (b, end="\r")
            if prevRoot !=root:#cuando se cambia de subcarpeta analizada
                cant=cant-1
                if i==1:#si es la primera vez
                  prevRoot=root
                  directories.append(root)#lista con rutas subcarpetas
                  cant=1
                  i=0
                else:
                  print(prevRoot, cant)
                  prevRoot=root
                  directories.append(root)#lista con rutas subcarpetas
                  dircount.append(cant)#lista con cantidad de imágenes en carpeta
                  cant=1

#para la última clase
print(root, cant)
dircount.append(cant)

print('Directorios leídos:',len(directories))
print("Imágenes en cada subcarpeta", dircount)
print('Suma total de imágenes en subcarpetas:',sum(dircount))
```

Con la ayuda de los comentarios resulta evidente la forma de proceder del código, es genérico para cualquier dataset y nos proporciona las características por pantalla. Para nuestro dataset muestra lo siguiente:

```
leyendo imágenes de  /content/dataset/
/content/dataset/recf 2168
/content/dataset/atrasf 1430
/content/dataset/derf 1533
/content/dataset/izqf 1239
Directorios leídos: 4
Imágenes en cada subcarpeta [2168, 1430, 1533, 1239]
Suma total de imágenes en subcarpetas: 6370
```

Etiquetar las imágenes

El concepto de etiquetar se puede traducir en indicar a qué clase pertenece cada imagen de nuestro dataset.

```
labels=[]
indice=0
#recorremos cada clase etiquetándola
for cantidad in dircount:
    for i in range(cantidad):
        labels.append(indice)
    indice=indice+1
print("Cantidad etiquetas creadas: ",len(labels))

direcciones=[]
indice=0
```

```
for directorio in directories:
    name = directorio.split(os.sep)#extraemos el nombre de la clase de su ruta
    print(indice , name[len(name)-1])
    direcciones.append(name[len(name)-1])
    indice=indice+1
```

Creamos un vector con las etiquetas de manera que a la primera clase la denominamos como un 0, la segunda un 1, etc. Para ello, usamos la lista que tenemos con las cantidades de imágenes disponibles para cada clase denominado "dircount". La segunda parte es para visualizar cada clase con su número asignado. Obtenemos lo siguiente por pantalla:

```
Cantidad etiquetas creadas:  6370
0 recf
1 atrasf
2 derf
3 izqf
```

Pasar de elementos de tipo lista a array (Numpy)

```
#convierto de lista a numpy
y = np.array(labels)
X = np.array(images, dtype=np.uint8)
#en esta ultima linea, adicionalmente, pasamos los valores de las imágenes a un rango de enteros de 0 a 255
#la 'X' serán las imágenes y la 'y' serán las etiquetas

classes = np.unique(y)#devuelve array con los valores diferentes que hay en 'y'
nClasses = len(classes)
print('Número total de salidas : ', nClasses)
print('Clases de salida : ', classes)
```

El tipo array de Numpy es necesario para el posterior procesamiento de los datos por el resto de las funciones.

```
Número total de salidas :  4
Clases de salida :  [0 1 2 3]
```

División en set de entrenamiento y de test

```
train_X,test_X,train_Y,test_Y = train_test_split(X,y,test_size=0.2)
#se divide en 80% imágenes para entrenamiento y 20% para tests futuros
#la división se hace tanto sobre las imágenes, como sobre las etiquetas evidentemente

#mostramos las características de estos sets creados
#muestra el tamaño de cada una de las dimensiones de los arrays
#por orden: número de fotos, tamaño anchura, tamaño altura, número de capas (imágenes en RGB)
#eso sería en el caso del set de imágenes, las etiquetas sólo tienen una dimensión
#muestra el número de etiquetas
print('Características set de entrenamiento : ', train_X.shape, train_Y.shape)
print('Características set de test : ', test_X.shape, test_Y.shape)
```

El código es claro.

```
Características set de entrenamiento :  (5096, 72, 128, 3) (5096,)
Características set de test :  (1274, 72, 128, 3) (1274,)
```

Normalización de los sets de imágenes

```
train_X = train_X.astype('float32')
test_X = test_X.astype('float32')
train_X = train_X / 255.
test_X = test_X / 255.
```

En primer lugar, pasamos los elementos a tipo flotante de manera que puedan tener parte fraccionaria y, a continuación, los pasamos a un rango de 0 a 1 diviendo entre 255. Esto se denomina normalización y fue explicado anteriormente.

One-hot enconding para etiquetas

```
#Pasamos de tipo categórico a one-hot
train_Y_one_hot = to_categorical(train_Y)
test_Y_one_hot = to_categorical(test_Y)

#Visualizamos un ejemplo del cambio
print('Etiqueta original:', train_Y[0])
print('Después de la conversión a one-hot enconding :', train_Y_one_hot[0])
```

El código es simple y el concepto ya ha sido explicado. Ejemplo que devuelve:

```
Etiqueta original: 1
Después de la conversión a one-hot enconding : [0. 1. 0. 0.]
```

Creación de set de entrenamiento y de validación

```
#Paso análogo al hecho al crear sets de entrenamiento y testing, pero, en
#este caso, también se mezclan todas las imágenes entre sí para que su orden sea compl
etamente aleatorio
#Mezclar todo y crear los grupos de entrenamiento y validación
train_X,valid_X,train_label,valid_label = train_test_split(train_X, train_Y_one_hot, t
est_size=0.2, random_state=13)

#visualizamos características de los sets creados
print(train_X.shape,valid_X.shape,train_label.shape,valid_label.shape)
```

Características de los sets:

```
(4076, 72, 128, 3) (1020, 72, 128, 3) (4076, 4) (1020, 4)
```

Creación de la CNN

```
#definimos los hiperparámetros
INIT_LR = 1e-3 # Valor inicial de learning rate
epochs = 8 # Cantidad de iteraciones completas al conjunto de imágenes de entrenamient
o
```

```
batch_size = 40 # tamaño de los lotes que se propagarán a la vez por la red

red = tf.keras.models.Sequential()
#capa de convolución con 275 filtros kernel de 3x3
red.add(Conv2D((275), kernel_size=(3,3),activation='linear',padding='same',input_shape
=(72,128,3)))
#input_shape es el tamaño de nuestras fotos que serán la entrada de la red
#padding='same' significa que sea hará padding de manera que en las convoluciones la
#salida sea de igual tamaño a la entrada
red.add(LeakyReLU(alpha=0.1))#función de activación LeakyReLU
red.add(MaxPooling2D((2, 2),padding='same'))#capa de reducción
red.add(Dropout(0.5))#regularización dropout

red.add(Flatten())#se aplanan los datos de las características extraídas
red.add(Dense(40, activation='linear'))#capa tradicional para clasificación de 100 neu
ronas
red.add(LeakyReLU(alpha=0.1))
red.add(Dropout(0.5))
red.add(Dense(nClasses, activation='softmax'))
#capa de salida con tantas neuronas como
#numero de clases haya ya que será el número de salidas. Se le aplica la función "soft
max" que
#proporcionará la probabilidad de ser una clase u otra

red.summary()#nos muestra tabla con las características de la red

#crea la red definiendo la función de pérdida, optimizador y el tipo de datos que va
#a proporcionar para el análisis de su comportamiento (metrics)
red.compile(loss=tf.keras.losses.categorical_crossentropy, optimizer=tf.keras.optimize
rs.Adagrad(lr=INIT_LR, decay=INIT_LR / 100),metrics=['accuracy'])
```

Esta es la tabla que se genera con la función summary, un resumen de la arquitectura de nuestra red proporcionándonos el número de parámetros que se van a entrenar:

```
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_6 (Conv2D)            (None, 72, 128, 275)      7700
_________________________________________________________________
leaky_re_lu_12 (LeakyReLU)   (None, 72, 128, 275)      0
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 36, 64, 275)       0
_________________________________________________________________
dropout_12 (Dropout)         (None, 36, 64, 275)       0
_________________________________________________________________
flatten_6 (Flatten)          (None, 633600)            0
_________________________________________________________________
dense_12 (Dense)             (None, 40)                25344040
_________________________________________________________________
leaky_re_lu_13 (LeakyReLU)   (None, 40)                0
_________________________________________________________________
dropout_13 (Dropout)         (None, 40)                0
_________________________________________________________________
dense_13 (Dense)             (None, 4)                 164
=================================================================
Total params: 25,351,904
Trainable params: 25,351,904
Non-trainable params: 0
```

Los datos que aparecen en la tabla concuerdan plenamente con lo explicado en la teoría sobre las CNNs. En total, se entrenan alrededor de 25 millones de parámetros conteniéndose la gran mayoría en la capa de clasificación que es una red neuronal de tipo tradicional. Cabe destacar que la capa de extracción de características devuelve unos 633.600 elementos con información para realizar la clasificación de las imágenes por lo que no parece una cifra descabellada los 25 millones de parámetros a ajustar en la capa final.

Entrenamiento de la red

Una vez que hemos construido la red que pretendemos, el último paso sería entrenarla para ver qué rendimiento nos proporciona. Esto se hace mediante la función "fit":

```
red_entrenada = red.fit(train_X, train_label, batch_size=batch_size,epochs=epochs,verb
ose=1,validation_data=(valid_X, valid_label))
```

Esta operación nos va devolviendo por pantalla en tiempo real los resultados obtenidos por la red en cada iteración de entrenamiento. Estos datos están separados por epoch y nos proporcionan: duración de la epoch, pérdida "loss" y precisión "accuracy" sobre set de entrenamiento, pérdida "val_loss" y precisión "val_accuracy" sobre set de validación. Este último parámetro es el que nos interesa principalmente ya que nos da una idea de cómo de bien generaliza nuestra red sobre el set de validación (no es usado en el entrenamiento, sólo se usa para dar idea de cómo generaliza la red ante datos desconocidos). Los datos que ha proporcionado el entrenamiento de nuestra red final son los siguientes:

```
Epoch 1/8
102/102 [==============================] - 17s 157ms/step - loss: 1.0068 - accuracy: 0.5726 - val_loss: 0.6570 - val_accuracy: 0.8304
Epoch 2/8
102/102 [==============================] - 15s 150ms/step - loss: 0.6415 - accuracy: 0.7679 - val_loss: 0.4693 - val_accuracy: 0.9069
Epoch 3/8
102/102 [==============================] - 15s 150ms/step - loss: 0.5010 - accuracy: 0.8221 - val_loss: 0.4111 - val_accuracy: 0.8676
Epoch 4/8
102/102 [==============================] - 15s 150ms/step - loss: 0.4545 - accuracy: 0.8469 - val_loss: 0.3394 - val_accuracy: 0.9343
Epoch 5/8
102/102 [==============================] - 15s 149ms/step - loss: 0.4135 - accuracy: 0.8548 - val_loss: 0.2988 - val_accuracy: 0.9059
Epoch 6/8
102/102 [==============================] - 15s 150ms/step - loss: 0.3898 - accuracy: 0.8651 - val_loss: 0.2948 - val_accuracy: 0.9137
Epoch 7/8
102/102 [==============================] - 15s 150ms/step - loss: 0.3676 - accuracy: 0.8717 - val_loss: 0.2467 - val_accuracy: 0.9422
Epoch 8/8
102/102 [==============================] - 15s 150ms/step - loss: 0.3428 - accuracy: 0.8884 - val_loss: 0.2510 - val_accuracy: 0.9402
```

Estos datos son realmente útiles para conocer si nuestra red no realiza "overfitting" (sobreajuste) sobre el set de entrenamiento, "overfitting" signidica que la red se centra en detalles no deseados del set de entrenamiento para mejorar sus predicciones y no aprende el patrón que se le pretende mostrar. En nuestro caso, el porcentaje de acierto sobre nuestro set de validación es superior al 94% mientras que sobre el set de entrenamiento es del 90, por lo que se puede afirmar que nuestra red generaliza muy bien. Esta situación se considera ideal en el entrenamiento de redes neuronales.

Guardar la red en archivo .h5py

Si al entrenar una arquitectura de red conseguimos unos resultados que son adecuados para nuestro propósito y queremos guardarla en un formato que pueda ser reconocido por un programa en Python para realizar predicciones con ella, simplemente tenemos que ejecutar la siguiente línea de código dándole el nombre que deseemos:

```
red.save("red_entrenada_final_2.h5")
```

Con todo el código anteriormente explicado sería suficiente para conseguir una red neuronal válida para implementarla en el dispositivo que deseemos.

4.2.1 Aspectos necesarios para que funcione en Google Colab

En este punto se van a explicar unos cuantos tips claves para poder entrenar redes neuronales fácilmente en Google Colab.

4.2.1.1 Subir nuestro .zip que contega nuestro dataset

El dataset lo subimos a Google Colab desde nuestro local en formato .zip y se descomprime, todo ello se hace mediante el código que aparece en la captura. De otra manera, sería muy laborioso subir carpeta por carpeta y, además, el .zip tiene menor tamaño al estar comprimido por lo que la subida se realiza más rápido. Esta subida tendrá que hacerse cada vez que se inicia Google Colab ya que todos los archivos se borran de una sesión a otra.

Figura 4-1. Subir .zip a Google Colab

4.2.1.2 Conectar nuestro Google Drive

Para poder tener un lugar donde guardar archivos de manera permanente, como el de nuestra red neuronal, hemos de conectarnos a nuestro Google Drive mediante las siguientes sentencias:

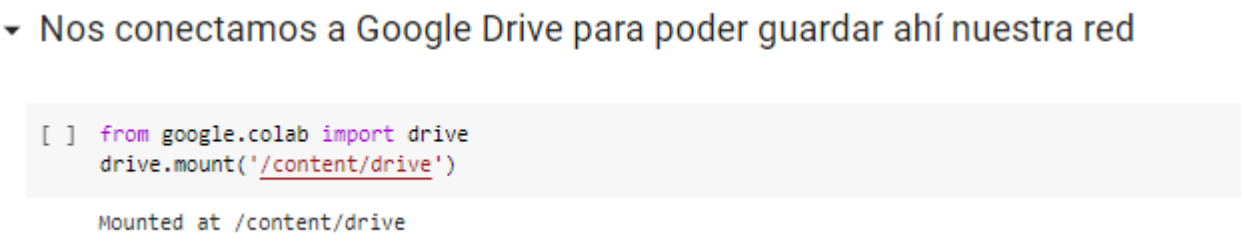

Figura 4-2. Conexión Google Drive

Una vez que estemos conectados, nuestro Google Drive aparecerá como una carpeta más en el menú de directorios y podremos guardar archivos en él o usar archivos que estén contenidos ahí.

4.2.2 Código extra para analizar performance de la red

Mediante la función "evaluate" le introducimos el dataset de test y nos evalúa la red dándonos el porcentaje de acierto obtenido.

```
test_eval = red.evaluate(test_X, test_Y_one_hot, verbose=1)
```

Resultados:

```
40/40 [==============================] - 2s 50ms/step - loss: 0.2534 - accuracy: 0.9286
```

Para poder tener una visión más clara de cómo actúa nuestra red, evaluamos este dataset por clases mediante el siguiente código:

```
#obtenemos las probabilidades dadas para cada clase según cada imagen
predicted_classes2 = red.predict(test_X)

#obtenemos la clase con mayor probabilidad cada imagen
predicted_classes=[]
for predicted_class in predicted_classes2:
    predicted_classes.append(predicted_class.tolist().index(max(predicted_class)))
predicted_classes=np.array(predicted_classes)

#comprobamos que el número de clases obtenido es el mismo que de fotos evaluadas
predicted_classes.shape, test_Y.shape
```

Efectivamente, es correcto:

```
((1274,), (1274,))
```

Con estas dos últimas sentencias mostramos por pantalla, gracias a la función "classification_report" de sklearn la precisión de la red especificada por clases:

```
target_names = ["Class {}".format(i) for i in range(nClasses)]
print(classification_report(test_Y, predicted_classes, target_names=target_names))
```

Se obtiene:

```
              precision    recall  f1-score   support

     Class 0       0.94      0.97      0.96       468
     Class 1       0.94      0.81      0.87       268
     Class 2       0.90      0.95      0.92       287
     Class 3       0.93      0.96      0.95       251

    accuracy                           0.93      1274
   macro avg       0.93      0.92      0.92      1274
weighted avg       0.93      0.93      0.93      1274
```

4.3. Dataset

La obtención del dataset se puede considerar uno de los factores clave en el éxito de este trabajo ya que el dataset marca completamente el comportamiento de nuestra red y es definido en su totalidad por nosotros.

El dataset con el que se ha entrenado finalmente la red se compone de 6370 fotos. Se ha intentado compensar el número de fotos entre todas las clases para asegurar un entrenamiento en el que no se tomaran ciertos patrones erróneos debido a que una característica fuera más común que otra. En concreto, esta es la lista de las clases y cantidad de fotos para cada una de ellas:

```
leyendo imágenes de  /content/dataset/
/content/dataset/recf 2168
/content/dataset/atrasf 1430
/content/dataset/derf 1533
/content/dataset/izqf 1239
Directorios leídos: 4
Imágenes en cada subcarpeta [2168, 1430, 1533, 1239]
Suma total de imágenes en subcarpetas: 6370
```

Captura extraída del momento de la carga del dataset en Google Colab. En los puntos posteriores se explican las diferentes decisiones que se han tenido que ir tomando para darle forma a nuestro dataset.

4.3.1 Procedimiento para obtener el dataset

En primera instancia, no se suele tener en cuenta el cómo obtener las imágenes que van a componer nuestro dataset, pero no resulta un procedimiento tan sumamente trivial y ha requerido cierta investigación.

Para su obtención, no se toman fotos porque se considera una manera poco eficiente de obtener la cantidad necesaria para crear un dataset útil. La conclusión a la que se llega es grabar vídeos y obtener los fotogramas de éstos para poder conseguir fotos de manera masiva.

Se graban vídeos con el programa explicado anteriormente (en el punto 3) moviendo manualmente el coche a lo largo de un "circuito" hecho con una cuerda con la cámara apuntando a ella, lógicamente. El vídeo grabado se divide en cortos según las direcciones que debería tomar el coche para cada orientación de la cuerda grabada.

Estos vídeos que se graban en h264(así graba la cámara de raspberry pi), se pasan de h264 a mp4, se extraen los fotogramas del mp4 en jpg y se pasan los lotes de fotos en jpg al tamaño que busquemos.

4.3.2 Definición de las clases

Al principio comencé intentando dividir mi dataset en 7 clases (direcciones que podría tomar mi robot): izquierda, un poco a la izquierda, recto, un poco a la derecha, derecha, parado y hacia atrás. Como idea inicial resulta maravilloso un robot que sea capaz de detectar tal sensibilidad al ver la cuerda, pero en la puesta en práctica no se ajusta mucho a la realidad. Dadas las indecisiones que se experimentaron a la hora de dividir los vídeos grabados sobre la cuerda para clasificarlos en una dirección u otra, se llega a la conclusión de que la red también iba a obtener patrones confusos a la hora de identificarlas.

A la hora de definir este tipo de datasets para clasificación hay una regla muy fácil de poner en práctica, si el humano duda a la hora de hacer la clasificación entre las clases que él mismo define, la red tampoco va a ser capaz de dar buenos resultados en ello.

La idea inicial no fue ni siquiera puesta en práctica a la hora de entrenar la red con estas clases ya que crear el dataset era muy complicado y generaba dudas. Se decide reducir las clases a 4 más simples: izquierda, de frente, derecha y hacia atrás. El patrón que se siguió para realizar la clasificación era el siguiente:

Si la cuerda estaba en esta orientación, iba hacia la derecha:

Figura 4-3. Ejemplo patrón clase derecha

Si la cuerda estaba en esta orientación, iba hacia la izquierda:

Figura 4-4. Ejemplo patrón clase izquierda

Si la cuerda no se consideraba claramente ni de izquierdas o derechas, se iba hacia delante:

Figura 4-5. Ejemplo patrón hacia delante 1

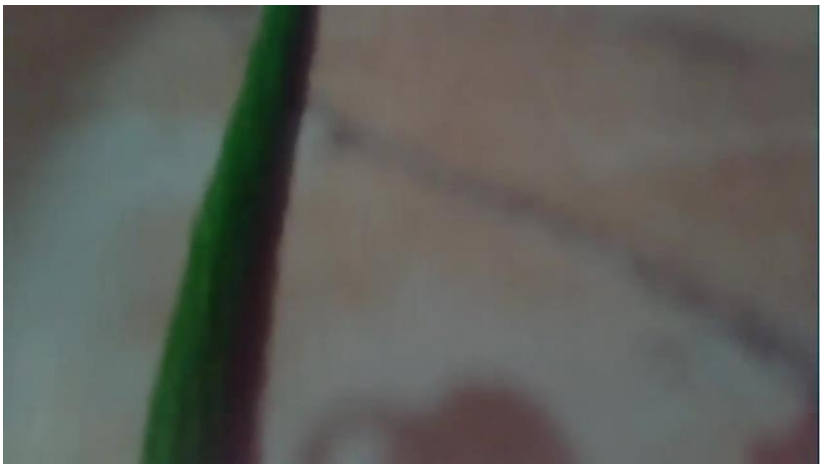

Figura 4-6. Ejemplo patrón hacia delante 2

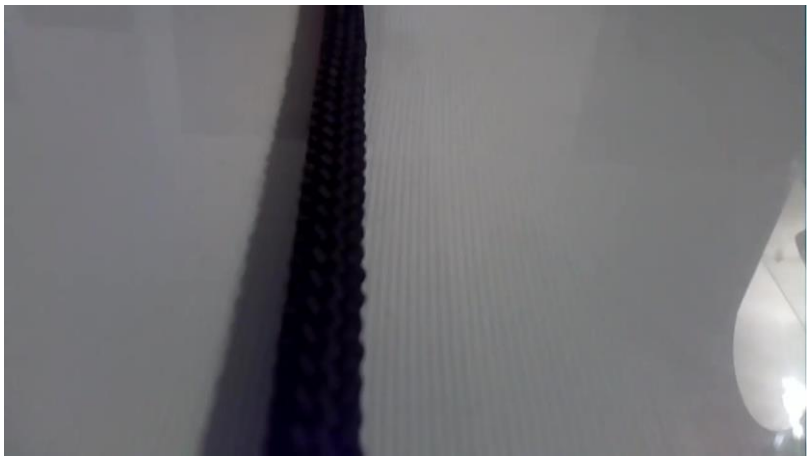

Figura 4-7. Ejemplo patrón hacia delante 3

Si no se encontraba cuerda en la imagen analizada, se iba hacia atrás:

Figura 4-8. Ejemplo patrón hacia atrás

Esta clasificación tan básica se pensó para que fuera muy fácil para nuestra red detectar el patrón que le deseábamos transmitir.

Dado que nuestro problema a clasificar no era un todo o nada, no es clasificar entre perros y gatos, es ir hacia la izquierda o hacia delante, por ejemplo. Aunque nuestro robot fuera hacia delante en situaciones en las que con nuestra primera idea de clasificación hubiera ido hacia la izquierda, nos aprovechamos del hecho de que no por no ir hacia la izquierda el robot va a salirse del recorrido necesariamente. Puede que vaya hacia delante en el comienzo de una curva a izquierdas, pero justamente en la siguiente foto que eche ya verá claramente que tiene

que ir hacia la izquierda y logrará corregir esa primera decisión volviendo al circuito. De hecho, la clase marcha atrás también se pensó con este propósito si, por ejemplo, llega a esa curva a izquierdas, sigue hacia delante y ya en la siguiente foto no detecta ninguna cuerda, el coche ha de volver hacia atrás de manera que en la siguiente foto ve la cuerda de nuevo y tras varias o ninguna corrección tomará la dirección correcta.

Esta idea explicada en el anterior párrafo resultó correcta y en la práctica el robot seguía con éxito la cuerda que le señalaba el camino.

4.3.3 Tamaño de las imágenes

Al principio del entrenamiento, se alimentaba a la red con imágenes de tamaño 32x18 píxeles por una decisión meramente arbitraria. En cierto modo se buscaba un entrenamiento rápido de la red y que pudiera procesar las imágenes que le llegaran con cierta facilidad.

Tras varios entrenamientos en los que cada vez conseguíamos afinar más nuestro data set de manera que las imágenes de éste representaran patrones bien definidos para cada clase, se llegó a cierto punto en el que la red ya no mejoraba sus predicciones, acertaba alrededor del 70% del set de validación, y se consideraba que la representatividad de las imágenes del dataset era poco mejorable. Tras ello, se decidió probar a aumentar el tamaño de las imágenes para ver qué suponía en mejora de predicciones y en términos de ralentizar el entrenamiento de la red.

Se entrenó la red con un dataset formado por las mismas imágenes que el anterior, pero pasando éstas de un tamaño de 32x18 a 128x72 píxeles, con este mero cambio de tamaño se consiguió pasar de una predicción del 70% de aciertos a sobrepasar el 80%, un completo éxito. Además, no supuso una ralentización notable en el tiempo de entrenamiento debido a que se usa el entorno Google Colab, o sea, se entrena mediante las GPUs de Google que son muy eficientes. Se podría afirmar casi con total seguridad que este cambio no podría haber sido realizado si se hubiera hecho el entrenamiento de la red con la CPU de mi PC.

4.3.4 Variedad necesaria en el dataset

Cuando se probaba el funcionamiento de nuestro coche para que siguiera una cuerda mediante la red entrenada, su comportamiento variaba dependiendo de la hora del día (que marcaba la luz que había), del tipo de suelo en el que se movía, las sombras, etc. Por todo ello, el dataset necesitaba ser lo más variado posible tomando fotos en diferentes suelos, a diferentes horas, con diferentes cuerdas, etc. De esta forma, le mostrábamos un patrón claro a nuestra red sobre lo que de verdad importaba que era la orientación de la cuerda que detectara.

Este es otro de los aspectos que inicialmente puede que ni se tuviera en cuenta al inicio pero que ha sido clave en el éxito del comportamiento de nuestro robot.

4.4. Modificaciones en la red buscando mejor rendimiento

Una vez que ya se considera que el dataset es lo sufientemente bueno y que no hay más margen de mejora mediante modificaciones en el mismo, se comenzó a retocar los hiperpárametros y a aumentar la complejidad de la red pasando de 32 a 100 filtros kernel en la convolución y el mismo cambio en la capa de clasificación pasando de 32 a 100 neuronas. Se aumentó el número de epochs al tenerse que entrenar un mayor número de parámetros y se redujo el tamaño de los lotes para hacer más intenso y largo el entrenamiento.

Nuestro mayor problema se concentraba en que la red proporcionaba una precisión que rozaba el 90% sobre el set de validación, pero al analizar esta precisión por clases se veía claramente como los fallos que cometía se concentraban en la clase que ordenaba “ir hacia atrás”, se acertaba sólo el 76% de los casos mientras que el resto de las clases superaban el 90. Este hecho repercutía en el comportamiento real del coche ya que había veces en las que se quedaba confundido moviéndose hacia atrás cuando no era necesario. Tras los cambios hechos en la estructura de la red, se mejoró la predicción hasta el 95% de acierto sobre el set de validación y siendo mucho más homogéneos los porcentajes de acierto entre las clases. La clase hacia atrás seguía siendo en la que más se fallaba, pero ya se acertaba en el 88% de los casos, por lo que la mejora era muy significativa y el comportamiento del coche mejoraba en gran medida.

Se hicieron pruebas buscando mejor rendimiento en la red mediante la adición de una nueva capa de convolución. Tras varios intentos, no se conseguía mejorar el rendimiento que ya nos proporcionaba la red con una sola capa de convolución y se descartó la idea.

El paso definitivo se produjo cuando se probó a priorizar el tener una gran cantidad de filtros kernel en la capa de convolución pasando de 100 a 275 y reduciendo la cantidad de neuronas en la capa final de unas 100 a 40. La cantidad de parámetros entrenables de la red era ligeramente superior a la anterior, pasando de alrededor de 23 millones a unos 25. La mejora en el rendimiento de la red fue muy notable ya que se consiguió un porcentaje de acierto sobre el set de validación de un 94,02% y, lo que más nos había costado, conseguir un comportamiento homogéneo entre todas las clases pasando ahora a ser la de "ir a la derecha" la de menor porcentaje de acierto con un 90%. Todo un éxito. Esta es la red definitiva usada en nuestro vehículo y la presentada como arquitectura de nuestra red en apartados anteriores.

4.5. Funcionamiento del coche controlado mediante la red

En este apartado se describe la forma en la que se pone en marcha nuestra red de manera que sea la que rige el comportamiento del vehículo.

4.4.1 Problemas con idea inicial

Inicialmente, la idea con la que nace el proyecto es conseguir un vehículo que tuviera una Raspberry Pi incorporada y ésta se encargara de ejecutar la red neuronal de manera que el ordenador sólo fuese usado para poner en marcha el dispositivo sin tener ningún papel protagonista. La idea es implementable y no parece descabellada, pero por ciertos problemas de software no se ha conseguido hacer funcionar la red neuronal en la RP. Se supone que con la versión TensorFlow Lite no habría problemas para ejecutar la red neuronal en la Rasspberry Pi, pero por algún motivo ajeno a mi entender, ni si quiera consigo instalarla en mi RP por lo que no lo puedo probar.

4.4.2 Solución adoptada

Dada la circunstancia, se ha tenido que buscar una alternativa en la que el PC ejecuta la red neuronal y le manda las órdenes al robot. Se ha implementado un sistema de comunicación extra PC-RP para que la RP le pueda enviar las fotos que toma al PC para que pueda analizarlas y dar órdenes.

Toda esta problemática empeora claramente el comportamiento final del coche, se muestra lento entre movimiento y movimiento debido a lo que se tarda en todo el proceso de manda foto/analiza/manda orden/mueve robot.

4.4.2.1 Programa que manda órdenes desde PC

El código se encuentra en el Anexo 3. El programa tiene la misma filosofía que el presentado en el apartado 3 para conseguir un funcionamiento básico del coche controlándolo desde el PC.

En este caso, el foco de las órdenes pasa a ser la red neuronal creada para ello. Se sigue haciendo uso de la conexión TCP para la comunicación de órdenes PC-RP y se implementa una conexión SFTP para poder manipular archivos remotos desde el PC, en este caso son las imágenes contenidas en la RP que se pasan al PC. Para la implementación de la conexión SFTP se hace uso de la librería Paramiko.

Se considera que el código del programa es muy claro en cada paso que toma dado los comentarios que se han añadido en él.

4.4.2.2 Programa que ejecuta las órdenes en la RP y manda fotos a PC

El código se encuentra en el Anexo 4. Análogamente al anterior, el programa vuelve a ser muy parecido al presentado en el apartado 3 pero implementando ciertas modificaciones en cómo se mueve el vehículo y dándole capacidad para hacer fotos en vez de vídeos.

Si el programa recibe un 0 desde el PC, el robot hace una foto. Si recibe un 5, aborta el programa y la conexión.

Se obliga a que el robot siempre comience cada ciclo de movimiento (cada vez que se inicie el programa) yendo hacia delante, suele ser la situación ideal comenzar así. Se implementa que el movimiento del robot será el captado por una orden anterior, de esta manera se consigue paliar el efecto de tener la cámara tan sumamente adelanta con respecto a la mayoría del cuerpo del coche ya que si no, hay ocasiones en las que se sube por la cuerda. En otras palabras, si el robot hace una foto y llega una orden de que se tiene que mover hacia la derecha, el robot guardará esa orden para el siguiente movimiento ya que en esta ocasión ejecutará la recibida para la foto anterior. Se implementa un retraso de una secuencia para paliar el efecto de tener el sensor adelantado.

Otra mejora que se impone en el movimiento del coche es que, si se gira dos veces consecutivas hacia un lado u otro, se obliga a que vaya hacia delante tras ello ya que el robot cuando se encuentra en una zona curvada de cuerda tiende a sobre girar hacia ella y con esta implementación se palia este problema.

Se implementa que, tras realizar cada orden, la RP avise con una señal al PC de ello para permitir la sincronización de ambos programas y evitar fallos por este motivo.

5 CONCLUSIONES

5.1. Análisis de resultados obtenidos

La parte más importante del proyecto era la implementación de la red neuronal haciéndola funcionar y conseguir buenos resultados, se puede decir que es un objetivo cumplido tal y como se ha ido explicando en el apartado de la puesta en marcha de la red.

La construcción del coche en sí ha sido de cierta dificultad, pero creo que la idea inicial se ha cumplido en este aspecto, habiendo un margen de mejora importante tal y como se explica a continuación.

El único objetivo que no se ha logrado alcanzar ha sido la implementación de la red para ser ejecutada en la Raspberry Pi a bordo del vehículo. La alternativa adoptada ha dado unos resultados decentes, pero en cierto modo se considera decepcionante ya que el hardware utilizado tiene potencial suficiente para llevar a cabo la ejecución a bordo.

En los vídeos siguientes se puede observar cómo nuestro robot dependiendo de las condiciones de la cuerda actúa de una manera u otra, aunque el recorrido sea prácticamente el mismo.

Esta es la que se puede considerar prueba 1: https://youtu.be/UJn2xAb3sAQ . En ella se intenta que el vehículo siga una cuerda para la que no ha sido entrenado y en un suelo que sí que aparece en nuestro dataset, pero en relativamente pocas ocasiones. Se comprueba como el robot en las zonas donde la curva está poco pronunciada consigue seguir la cuerda desconocida para él, pero llega un momento en el que ya no sabe a donde ir entrando en bucle de direcciones erróneas del que no consigue salir. Se intentó en varias ocasiones este recorrido y el resultado siempre fue el mismo.

En la prueba 2 (https://youtu.be/4lyyV9RY124) el robot sigue una cuerda que sí que aparece en el dataset, pero relativamente poco, y en el mismo suelo que la anterior, suelo poco común también. Se comprueba como el coche en este caso sí que es capaz de seguir perfectamente la cuerda de principio a fin dado que sí que parece tenerla identificada gracias a su aparición en el dataset.

La prueba 3 (https://youtu.be/BeE0klPEaTI) se hace en el mismo suelo que las dos anteriores, pero con la cuerda más predominante en el dataset. El resultado es que consigue seguirla perfectamente y sin aparente esfuerzo.

Analizando las tres pruebas, se puede comprobar lo importante que es la representatividad del dataset ya que es curioso lo que le cuesta al robot seguir la cuerda en la primera prueba, esta cuerda no aparece en su set de entrenamiento, pero sí que es muy parecida a la común. En cambio, en la prueba 2 sí que sigue perfectamente la cuerda, aunque sea muy diferente a la común, pero al aparecer en el dataset la tiene identificada. La prueba 3 es el ejemplo de que nuestro robot generaliza bien para diferentes suelos ya que sigue la cuerda perfectamente en un suelo no tan común.

Por último, se hace una prueba de mayor dificultad y duración en el suelo común para la red y con la cuerda que más aparece en el dataset. La sigue muy bien a pesar de cambios de luz entre habitaciones y aquí se puede ver el vídeo demostrativo: https://youtu.be/gFwn5DzcxuY

El análisis de las pruebas anteriores nos dice que, si se ampliara la diversidad de nuestro dataset, el robot sería capaz de seguir cuerdas en un número casi infinito de situaciones.

Como conclusión se podría decir, dado que se partía desde completamente cero, que el resultado del trabajo en términos generales se considera un éxito.

5.2. Mejoras y campos de trabajo futuros

El campo de mejoras para investigación en el futuro que ofrece este proyecto es muy amplio.

En primer lugar, centrándonos en el software cabría destacar la mejora que supondría un salto radical en el comportamiento de este vehículo: implementar la ejecución de la red neuronal en la misma Raspberry Pi. Esta mejora dejaría atrás la ineficiencia de tener que estar constante enviando fotos al PC y proporcionaría un movimiento mucho más continuo de nuestro robot.

En términos de lo que se puede considerar la red neuronal y aspectos que marcan su comportamiento, se podría estudiar una nueva forma de clasificación por clases de manera que fuera aún más exacto el seguimiento de las líneas, quizás ampliando la gama de clases entre las que se clasifica, esto supondría una gran dificultad tal y como se ha comentado. Adicionalmente, se podrían probar nuevas arquitecturas de CNNs para comprobar si hay margen de mejora en la exactitud que nos proporciona nuestra red actualmente. Cabe destacar que se considera bastante satisfactorio el resultado obtenido tras el entrenamiento de la red y serían ya mejoras para obtener una red prácticamente perfecta.

Por otra parte, la estructura del coche en sí se considera que tiene un amplio margen para mejorar el comportamiento que nos proporciona. Como mejora principal se podría decir que el coche tiene problemas de tracción debido a un fallo en la concepción del vehículo dado que los elementos que más pesan del hardware (batería y Raspberry Pi) se sitúan en la parte delantera dejando casi sin peso a las ruedas que dan tracción al vehículo. Otra deficiencia detectada en el vehículo es que el voltaje proporcionado por la batería a los motores(5V) en ciertos momentos es insuficiente para mover el vehículo, por lo que se podría usar otra batería de mayor voltaje para mejorar este aspecto y, quizás, cambiar los motores por unos de mayor rango ya que los actuales solo llegan hasta 6V. En última instancia, sería una mejora difícil de implementar, se podrían poner ruedas delanteras orientadas mediante servomotores de manera que giren hacia un lado u otro según la dirección que tenga que tomar el vehículo, podría mejorar el comportamiento de éste en gran medida.

En último lugar, se podrían estudiar nuevas formas de señalización del camino a tomar e incluso implementar nuevas cámaras o sensores que se complementaran con la cámara ya existente.

REFERENCIAS

[1] F. S. Caparrini. [En línea]. Available: http://www.cs.us.es/~fsancho/?e=221.

[2] «La Vanguardia,» [En línea]. Available: https://www.lavanguardia.com/motor/20210324/6604505/futuro-automovil-esta-inteligencia-artificial-conduccion-autonoma-brl.html.

[3] «Hackercar.com,» [En línea]. Available: https://hackercar.com/los-retos-que-les-quedan-por-superar-a-los-coches-autonomos/.

[4] «BBVA Open Mind,» [En línea]. Available: https://www.bbvaopenmind.com/tecnologia/inteligencia-artificial/el-comienzo-de-la-era-de-la-inteligencia-artificial/.

[5] Á. R. Pinilla. [En línea]. Available: http://oa.upm.es/56835/1/TFG_ANGEL_RECAS_PINILLA.pdf.

[6] F. S. Caparrini. [En línea]. Available: http://www.cs.us.es/~fsancho/?e=72.

[7] «Joel-RedesNeuronalesArtificiales.blogspot.com,» [En línea]. Available: http://joel-redesneuronalesartificiales.blogspot.com/2008/07/redes-neuronales-artificiales.html.

[8] «Wikipedia.org,» [En línea]. Available: https://es.wikipedia.org/wiki/Perceptr%C3%B3n_multicapa.

[9] «Wikipedia.org,» [En línea]. Available: https://es.wikipedia.org/wiki/Red_neuronal_artificial.

[10] «DiegoCalvo.es,» [En línea]. Available: https://www.diegocalvo.es/clasificacion-de-redes-neuronales-artificiales/.

[11] F. S. Caparrini. [En línea]. Available: http://www.cs.us.es/~fsancho/?e=77.

[12] «Wikipedia.org,» [En línea]. Available: https://en.wikipedia.org/wiki/Activation_function.

[13] «BootCampAI.Medium.com,» [En línea]. Available: https://bootcampai.medium.com/redes-neuronales-13349dd1a5bb.

[14] «Wikipedia.org,» [En línea]. Available: https://en.wikipedia.org/wiki/Softmax_function.

[15] «ProgrammerClick.com,» [En línea]. Available: https://programmerclick.com/article/79781455632/#_3.

[16] «Wikipedia.org,» [En línea]. Available: https://es.wikipedia.org/wiki/Propagaci%C3%B3n_hacia_atr%C3%A1s.

[17] «InteractiveChaos.com,» [En línea]. Available: https://interactivechaos.com/es/manual/tutorial-de-machine-learning/backpropagation.

[18] «Wikipedia.org,» [En línea]. Available: https://en.wikipedia.org/wiki/Gradient_descent.

[19] «VicentBlog.xyz,» [En línea]. Available: https://vincentblog.xyz/posts/conceptos-basicos-sobre-redes-neuronales.

[20] «IgnacioGavilan.com,» [En línea]. Available: https://ignaciogavilan.com/catalogo-de-componentes-de-redes-neuronales-iii-funciones-de-perdida/.

[21] «Wikipedia.org,» [En línea]. Available: https://en.wikipedia.org/wiki/Convolutional_neural_network#Dropout.

[22] «Medium.com,» [En línea]. Available: https://medium.com/metadatos/t%C3%A9cnicas-de-regularizaci%C3%B3n-b%C3%A1sicas-para-redes-neuronales-b48f396924d4.

[23] «AprendeMachineLearning.com,» [En línea]. Available: https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/.

[24] «AprendeMachineLearning.com,» [En línea]. Available: https://www.aprendemachinelearning.com/clasificacion-de-imagenes-en-python/.

[25] «ComputerScienceWiki.org,» [En línea]. Available: https://computersciencewiki.org/index.php/Max-pooling_/_Pooling.

[26] «BootCampAI.Medium.com,» [En línea]. Available: https://bootcampai.medium.com/redes-neuronales-convolucionales-5e0ce960caf8.

[27] «SuperDataScience.com,» [En línea]. Available: https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-3-flattening.

[28] «Blog.JoseMarianoAlvarez.com,» [En línea]. Available: http://blog.josemarianoalvarez.com/2018/03/15/categorias-y-la-codificacion-one-hot/.

[29] «VicentBlog.xyz,» [En línea]. Available: https://vincentblog.xyz/posts/redes-neuronales-convolucionales.

[30] «Xataka.com,» [En línea]. Available: https://www.xataka.com/basics/arduino-raspberry-pi-que-cuales-sus-diferencias.

[31] «Datasheets.Raspberrypi.org,» [En línea]. Available: https://datasheets.raspberrypi.org/rpi4/raspberry-pi-4-datasheet.pdf.

[32] «Sensoricx.com,» [En línea]. Available: https://sensoricx.com/electronica-de-control/circuito-integrado-l293d-que-es-para-que-sirve-y-como-funciona/.

[33] «TalosElectronics.com,» [En línea]. Available: https://www.taloselectronics.com/blogs/tutoriales/puente-h.

[34] «AprendeAI.com,» [En línea]. Available: https://aprendeia.com/que-es-tensorflow-como-funciona/.

[35] «Wikipedia.org,» [En línea]. Available: https://es.wikipedia.org/wiki/TensorFlow.

[36] «LuisLlamas.es,» [En línea]. Available: https://www.luisllamas.es/machine-learning-con-tensorflow-y-keras-en-python/.

[37] «Wikipedia.org,» [En línea]. Available: https://es.wikipedia.org/wiki/Keras.

[38] «SitioBigData.com,» [En línea]. Available: https://sitiobigdata.com/2018/09/19/red-neuronal-en-keras/#.

[39] «AprendeConAlf.es,» [En línea]. Available: https://aprendeconalf.es/docencia/python/manual/numpy/.

[40] «Docs.Python.org,» [En línea]. Available: https://docs.python.org/es/3.10/library/os.html.

[41] «Docs.Python.org,» [En línea]. Available: https://docs.python.org/es/3/library/re.html.

[42] «Programacion.net,» [En línea]. Available: https://programacion.net/articulo/introduccion_a_la_libreria_matplotlib_de_python_1599.

[43] «InteractiveChaos.com,» [En línea]. Available: https://interactivechaos.com/es/python/function/sklearnmodelselectiontraintestsplit.

[44] «JoserZapata.github.io,» [En línea]. Available: https://joserzapata.github.io/courses/python-ciencia-datos/ml/.

[45] «Keras.io,» [En línea]. Available: https://keras.io/api/models/model_training_apis/.

[46] «UniPython.com,» [En línea]. Available: https://unipython.com/desarrolla-primera-red-neural-python-keras-paso-paso/.

[47] «HDMagazine.org,» [En línea]. Available: http://46.101.4.154/Art%C3%ADculos%20t%C3%A9cnicos/Python/Paramiko%20-%20Conexiones%20SSH%20y%20SFTP.pdf.

ANEXOS

Anexo 1

```
import socket
from pynput import keyboard
import sys

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('192.168.43.73', 1234))#ip del ordenador
s.listen(1)


while True:
    clientSocket, address = s.accept()
    print(f'Connection with {address} established')

    def on_press(key):
        if key == keyboard.KeyCode.from_char('r'):
            print('Speed up')
            clientSocket.send(bytes("5", 'utf-8'))

        if key == keyboard.KeyCode.from_char('f'):
            print('Speed down')
            clientSocket.send(bytes("6", 'utf-8'))

        if key == keyboard.KeyCode.from_char('p'):
            print('Taking photos')
            clientSocket.send(bytes("7", 'utf-8'))

        if key == keyboard.Key.up:
            print('Forward')
            clientSocket.send(bytes("1", 'utf-8'))

        elif key == keyboard.Key.down:
            print('Down')
            clientSocket.send(bytes("4", 'utf-8'))

        elif key == keyboard.Key.left:
            print('Left')
            clientSocket.send(bytes("2", 'utf-8'))

        elif key == keyboard.Key.right:
            print('Right')
            clientSocket.send(bytes("3", 'utf-8'))
```

```
def on_release(key):
    print('Stop!')
    clientSocket.send(bytes("0", 'utf-8'))

    if key == keyboard.Key.esc:
        print('Exiting Program')
        clientSocket.send(bytearray("404", 'utf-8'))
        return False

with keyboard.Listener(
        on_press=on_press,
        on_release=on_release) as listener:
    listener.join()
sys.exit()
```

Anexo 2

```
import picamera
import datetime
import socket
import sys
import RPi.GPIO as GPIO
#robot = Robot(left=(17,27), right=(23,24))#GPIOs de robot

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('192.168.43.73', 1234))#ip del ordenador

#enablers
e1=33
e2=32
#outputs
i1=11
i2=13
i3=18
i4=16

#set GPIO numbering mode and define output pins
GPIO.setmode(GPIO.BOARD)
GPIO.setup(i1,GPIO.OUT)
GPIO.setup(i2,GPIO.OUT)
GPIO.setup(i3,GPIO.OUT)
GPIO.setup(i4,GPIO.OUT)

#setup PWM control
v=48
GPIO.setup(e1,GPIO.OUT)
GPIO.setup(e2,GPIO.OUT)
speedleft = GPIO.PWM(e1,v)
speedright = GPIO.PWM(e2,v)
speedleft.start(v)
speedright.start(v)

record = 0#to stop or start recording

#camera
camera = picamera.PiCamera()
camera.vflip = True
camera.hflip = True
camera.resolution = (1280,720)
camera.framerate = (25)


while True:
    msg = s.recv(1024)
```

```
msg = msg.decode('utf-8')
speedleft.ChangeDutyCycle(v)
speedright.ChangeDutyCycle(v)

if msg == '1':
    #robot.forward
    GPIO.output(i1,False)
    GPIO.output(i2,True)
    GPIO.output(i3,True)
    GPIO.output(i4,False)

elif msg == '2':
    #robot.left
    speedleft.ChangeDutyCycle(0.1*v)
    GPIO.output(i1,False)
    GPIO.output(i2,True)
    GPIO.output(i3,True)
    GPIO.output(i4,False)

elif msg == '3':
    #robot.right
    speedright.ChangeDutyCycle(0.1*v)
    GPIO.output(i1,False)
    GPIO.output(i2,True)
    GPIO.output(i3,True)
    GPIO.output(i4,False)

elif msg == '4':
    #robot.backward
    GPIO.output(i1,True)
    GPIO.output(i2,False)
    GPIO.output(i3,False)
    GPIO.output(i4,True)

elif msg == '0':
    #robot.stop
    GPIO.output(i1,False)
    GPIO.output(i2,False)
    GPIO.output(i3,False)
    GPIO.output(i4,False)

elif msg == '5':

    if v == 100:
        print("Maximum speed, not possible to speed up")

    else:
        v=v+13
        speedleft.ChangeDutyCycle(v)
```

```
            speedright.ChangeDutyCycle(v)
            print("Speed:",v)

    elif msg == '6':

        if v == 22:
            print("Minimum speed, not possible to speed down")

        else:
            v=v-13
            speedleft.ChangeDutyCycle(v)
            speedright.ChangeDutyCycle(v)
            print("Speed:",v)

    elif msg == '7':#to start or stop recording

        if record == 1:
            record = 0
            camera.stop_recording()
            print('Stop recording')

        else:
            record = 1
            moment = datetime.datetime.now()
            camera.start_recording('/home/pi/Desktop/tanque/images/vid_%02d_%02d_%02d.
h264' % (moment.hour, moment.minute, moment.second))
            print('Start recording')

    elif msg == '404':
        print("Quiting")
        GPIO.cleanup()
        sys.exit()

    else:
        print("Connected to Server!")
        print("Speed",v)
```

Anexo 3

```
import tensorflow as tf
from skimage.transform import resize
import matplotlib.pyplot as plt
import time
import numpy as np
import paramiko
import socket

#cargamos modelo de red neuronal
model = tf.keras.models.load_model('red_entrenada_final_2.h5')
print('Red neuronal cargada')

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('192.168.43.73', 1234))#ip del ordenador
s.listen(1)

clientSocket, address = s.accept()
print(f'Conexión establecida con {address}')

#creamos una conexión SFTP para manipular archivos remotos gracias
#a la libreria Paramiko
ssh_client=paramiko.SSHClient()
ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
#datos de la Raspberry Pi
ssh_client.connect('192.168.43.127',username='pi',password='luis')
ftp_client=ssh_client.open_sftp()



time.sleep(1)


print('entro while')
try:
    while True:
        #Recibir nueva imagen de la rpi:
        #se manda orden de hacer foto nueva
        clientSocket.send(bytes("0", 'utf-8'))

        #se espera a recibir confirmación de que acaba orden
        clientSocket.recv(1024)

        images=[]
        filename='img.jpg'

        #cogemos el archivo de la RP y lo guardamos en la carpeta en la que
        #estamos ejecuntando este programa con el mismo nombre
```

```
        ftp_client.get('/home/pi/Desktop/tanque/images/img.jpg','img.jpg')
        #aquí se coge la foto nueva

        print('nueva foto')

        #preprocesamos imagen
        image = plt.imread(filename,0)
        #por si hiciera falta redimensionarla y se le hace aplica
        #anti aliasing que suaviza la foto y es recomendable en estos casos
        image_resized = resize(image, (72, 128),anti_aliasing=True,clip=False,preserve
_range=True)
        images.append(image_resized)

        X = np.array(images, dtype=np.uint8) #convierto de lista a numpy
        test_X = X.astype('float32')
        test_X = test_X / 255.

        predicted_classes = model.predict(test_X)
        #nos da un array con las probabilidades

        for i, img_tagged in enumerate(predicted_classes):
            pred=img_tagged.tolist().index(max(img_tagged))
            #la predicción será la clase más probable
        print('nueva prediccion')

        #según la clase predicha se manda una orden u otra
        if pred == 0:
            clientSocket.send(bytes("1", 'utf-8'))
            print('Frente')
        elif pred == 3:
            clientSocket.send(bytes("2", 'utf-8'))
            print('Izquierda')
        elif pred == 2:
            clientSocket.send(bytes("3", 'utf-8'))
            print('Derecha')
        else:
            clientSocket.send(bytes("4", 'utf-8'))
            print('Atrás')
        #se espera confirmación de que acaba orden
        clientSocket.recv(1024)

# si se pulsa ctrl+C el programa se acaba mandando
#señal de finalización también a la RP
except KeyboardInterrupt:
    clientSocket.send(bytes("5", 'utf-8'))
    clientSocket.close()
    s.close()
    ftp_client.close()
    print('Cierre de conexión')
```

Anexo 4

```
import picamera
import time
import socket
import sys
import RPi.GPIO as GPIO

GPIO.cleanup()

e1=33
e2=32
i1=11
i2=13
i3=18
i4=16

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('192.168.43.73', 1234))#ip del ordenador

#set GPIO numbering mode and define output pins
GPIO.setmode(GPIO.BOARD)
GPIO.setup(i1,GPIO.OUT)
GPIO.setup(i2,GPIO.OUT)
GPIO.setup(i3,GPIO.OUT)
GPIO.setup(i4,GPIO.OUT)

#setup PWM control
v=100
GPIO.setup(e1,GPIO.OUT)
GPIO.setup(e2,GPIO.OUT)
speedleft = GPIO.PWM(e1,v)
speedright = GPIO.PWM(e2,v)
speedleft.start(v)
speedright.start(v)

#se da esta orden de parar por si en el anterior
#intento hubo algun error y se quedó en movto
GPIO.output(i1,False)
GPIO.output(i2,False)
GPIO.output(i3,False)
GPIO.output(i4,False)

#camera
camera = picamera.PiCamera()
camera.vflip = True
camera.hflip = True
camera.resolution = (128,72)
#resolucion que necesita Red Neuronal
```

```
#se preinicializa la cámara durante 1 segundo
camera.start_preview()
time.sleep(1)
camera.capture('images/img.jpg')

#imponemos que se empieza yendo hacia delante
msgant='1'
msgpre='12'
r1=0

print('entro while')
while True:
    r=0
    speedleft.ChangeDutyCycle(v)
    speedright.ChangeDutyCycle(v)
    #se espera una nueva orden
    msg = s.recv(1024)
    msg = msg.decode('utf-8')

    if msg == '0':
        #Capturamos imagen cuando el PC lo indique
        camera.capture('images/img.jpg')
        print('Capturo foto')


    else:
        #para señalizar si la orden de movto anterior ha sido
        #el mismo que se realiza ahora
        #solo se hace en caso de giros
        if msgant==msgpre and msgant!='1' and msgant!='4':
            r=1

        #la orden anterior es la que se procesa en cada iteración

        if msgant == '1':
            #robot.forward()
            GPIO.output(i1,False)
            GPIO.output(i2,True)
            GPIO.output(i3,True)
            GPIO.output(i4,False)
            print('Voy frente')
            time.sleep(0.12)

        elif msgant == '2':
            #robot.left()
            speedleft.ChangeDutyCycle(0.1*v)
            GPIO.output(i1,False)
            GPIO.output(i2,True)
```

```
            GPIO.output(i3,True)
            GPIO.output(i4,False)
            print('Voy izq')
            time.sleep(0.12)
            #se mueve más tiempo hacia la izquierda porque se detecta
            #que el motor izquierdo proporciona menos potencia que
            #el derecho

        elif msgant == '3':
            #robot.right()
            speedright.ChangeDutyCycle(0.1*v)
            GPIO.output(i1,False)
            GPIO.output(i2,True)
            GPIO.output(i3,True)
            GPIO.output(i4,False)
            print('Voy dcha')
            time.sleep(0.12)

        elif msgant == '4':
            #robot.backward()
            GPIO.output(i1,True)
            GPIO.output(i2,False)
            GPIO.output(i3,False)
            GPIO.output(i4,True)
            print('Voy atras')
            time.sleep(0.12)

        #si se han recibido dos señales de movto iguales seguidas que no sean
        #ir hacia delante o atrás, se va hacia delante
        if r==1 and r1==0:
            #robot.forward()
            GPIO.output(i1,False)
            GPIO.output(i2,True)
            GPIO.output(i3,True)
            GPIO.output(i4,False)
            print('Voy frente')
            time.sleep(0.12)
            r1=1
        elif r1==1:
            r1=0
        msgant=msg
        msgpre=msgant

#se para el vehiculo hasta nueva orden
GPIO.output(i1,False)
GPIO.output(i2,False)
GPIO.output(i3,False)
GPIO.output(i4,False)
```

```
    if msg == '5':
        break

    #se señaliza al PC que ha acabado una nueva orden
    print('acaba orden')
    s.send('9')

camera.stop_preview()
s.close()
GPIO.cleanup()
sys.exit()
```