Trabajo Fin de Grado
Ingeniería Electrónica, Robótica y Mecatrónica

Estimación de orientación de un dron mediante método bioinspirado basado en Ocelos

Autor: Fernando Cristo Sanz García-Muñoz

Tutor: Jesús Capitán Fernández

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

Sevilla, 2019

Trabajo Fin de Grado
Ingeniería Electrónica, Robótica y Mecatrónica

Estimación de orientación de un dron mediante método bioinspirado basado en Ocelos

Autor:
Fernando Cristo Sanz García-Muñoz

Tutor:
Jesús Capitán Fernández
Ayudante Doctor

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

Sevilla, 2019

Trabajo Fin de Grado: Estimación de orientación de un dron mediante método bioinspirado basado en Ocelos

Autor: Fernando Cristo Sanz García-Muñoz
Tutor: Jesús Capitán Fernández

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

Para comenzar me gustaría agradecer a los dos tutores con los que traté durante el proyecto. Fernando Caballero el cual me lo asignó y me suministro los datos necesarios para realizarlo, y Jesús Capitán quien amablemente continuó con el puesto de tutor y me ayudo a proseguir con él. Gracias a ambos por las oportunidades y el apoyo que me dieron durante su desarrollo.

Por supuesto a mi familia, por todos estos años a mi lado, que con su apoyo y confianza me instaron a seguir adelante incluso en los malos momentos.

Y por último pero no menos importante a mis amigos, sobretodo a los que me han acompañado durante estos años de carrera. A ellos he de agradecer muchos de los momentos que más recordaré de esta etapa, y la ayuda que siempre me ofrecieron cuando lo necesitaba.

Fernado Cristo Sanz García-Muñoz
Sevilla, 2019

Resumen

Desde siempre, la naturaleza ha sido una de las mayores fuentes de inspiración del ser humano, a lo largo de su historia, e incluso en el mundo actual inundado por la tecnología, todavía esta nos demuestra que nos faltan cosas por aprender y mejorar. Este deseo de aprender de la naturaleza se unió a la ciencia en las llamadas biónica y biomimesis, dando lugar al desarrollo de por ejemplo los sistemas bioinspirados, en los cuales se basará este proyecto.

En concreto esta investigación se nos plantea el desarrollo de un sistema, capaz de replicar la funcionalidad de los órganos visuales llamados ocelos, mediante un sistema basado en las redes neuronales convolucionales. Estos son pequeños ojos encontrados en gran variedad de insectos, y entre otras cualidades se investiga que les ayuden a orientarse durante el vuelo. Al existir esta relación, conseguir una red entrenada permitiría su uso en vehículos y robots aéreos, consiguiendo una alternativa puede que incluso mejor que otros métodos electrónicos. El objetivo por lo tanto, será diseñar una red que con los datos de entrada de los que disponen los ocelos orgánicos, consiga predecir con la mayor fiabilidad posible los datos espaciales del sistema.

Abstract

Historically, nature has been one of the greatest sources of inspiration for the human being, and even in the current world, awash by technology, it still remind us that there are many things left to learn and improve. This desire to learn from nature joined science in the so-called bionics and biomimetics, giving rise to for example the development of bio-inspired systems, on which this project will be based.

Specifically, this research shows the development of a system capable to replicate the functionality of visual organs called ocellis, by using convolutional neural networks. These are small eyes that can be found in a great variety of insects and, among other features, investigations have proved that they are helpful for fly orientation. Therefore, using a trained network would allow us to implement it in aerial vehicles and robots, getting an alternative that might be better than other electronic methods. The objective will be to design a network that, with the input data available for organic ocelli, can predict the spatial data of the system with the highest possible accuracy.

Índice Abreviado

Índice

Notación

$\mathbb{R}$	Cuerpo de los números reales
$\mathbb{C}$	Cuerpo de los números complejos
$\|\mathbf{v}\|$	Norma del vector $\mathbf{v}$
$\langle \mathbf{v}, \mathbf{w} \rangle$	Producto escalar de los vectores $\mathbf{v}$ y $\mathbf{w}$
$\|\mathbf{A}\|$	Determinante de la matriz cuadrada $\mathbf{A}$
$\det(\mathbf{A})$	Determinante de la matriz (cuadrada) $\mathbf{A}$
$\mathbf{A}^\top$	Transpuesto de $\mathbf{A}$
$\mathbf{A}^{-1}$	Inversa de la matriz $\mathbf{A}$
$\mathbf{A}^\dagger$	Matriz pseudoinversa de la matriz $\mathbf{A}$
$\mathbf{A}^{\mathrm{H}}$	Transpuesto y conjugado de $\mathbf{A}$
$\mathbf{A}^*$	Conjugado
c.t.p.	En casi todos los puntos
c.q.d.	Como queríamos demostrar
■	Como queríamos demostrar
□	Fin de la solución
e.o.c.	En cualquier otro caso
e	número e
$\mathrm{e}^{\mathrm{j}x}$	Exponencial compleja
$\mathrm{e}^{\mathrm{j}2\pi x}$	Exponencial compleja con 2π
$\mathrm{e}^{-\mathrm{j}x}$	Exponencial compleja negativa
$\mathrm{e}^{-\mathrm{j}2\pi x}$	Exponencial compleja negativa con 2π
$\mathbb{R}\mathrm{e}$	Parte real
$\mathbb{I}\mathrm{m}$	Parte imaginaria
sen	Función seno
tg	Función tangente
arc tg	Función arco tangente
$\sin^y x$	Función seno de x elevado a y
$\cos^y x$	Función coseno de x elevado a y
Sa	Función sampling
sgn	Función signo
rect	Función rectángulo
Sinc	Función sinc
$\frac{\partial y}{\partial x}$	Derivada parcial de y respecto a x
x°	Notación de grado, x grados.
$\Pr(A)$	Probabilidad del suceso A
$E[X]$	Valor esperado de la variable aleatoria X
σ_X^2	Varianza de la variable aleatoria X
$\sim f_X(x)$	Distribuido siguiendo la función densidad de probabilidad $f_X(x)$
$\mathcal{N}(m_X, \sigma_X^2)$	Distribución gaussiana para la variable aleatoria X, de media m_X y varianza σ_X^2

$\mathbf{I}_n$	Matriz identidad de dimensión n
$\operatorname{diag}(\mathbf{x})$	Matriz diagonal a partir del vector **x**
$\operatorname{diag}(\mathbf{A})$	Vector diagonal de la matriz **A**
SNR	Signal-to-noise ratio
MSE	Minimum square error
:	Tal que
$\stackrel{\text{def}}{=}$	Igual por definición
$\lVert\mathbf{x}\rVert$	Norma-2 del vector **x**
$\lvert\mathbf{A}\rvert$	Cardinal, número de elementos del conjunto **A**
$\mathbf{x}_i, i = 1,2,\ldots,n$	Elementos i, de 1 a n, del vector **x**
$\mathrm{d}x$	Diferencial de x
$\leqslant$	Menor o igual
$\geqslant$	Mayor o igual
$\backslash$	Backslash
$\Leftrightarrow$	Si y sólo si
$x = a+3 \underset{\substack{\uparrow \\ a=1}}{=} 4$	Igual con explicación
$\frac{a}{b}$	Fracción con estilo pequeño, a/b
Δ	Incremento
$b \cdot 10^a$	Formato científico
$\underset{x}{\rightarrow}$	Tiende, con x
O	Orden
™	Trade Mark
$\mathbb{E}[x]$	Esperanza matemática de x
$\mathbf{C}_\mathbf{x}$	Matriz de covarianza de **x**
$\mathbf{R}_\mathbf{x}$	Matriz de correlación de **x**
σ_x^2	Varianza de x

1 Introducción

Inteligencias artificiales, Robots o sistemas automáticos, son solo alguno de los temas más tratados actualmente. Estos se han ido haciendo un hueco en nuestra sociedad, debido a la comodidad con la que nos permiten realizar algunas acciones. Ya sea como medios de estudio, permitiendo al ser humano adentrarse en entornos antes difícilmente accesibles, o evitando daños humanos en procesos con alto riesgo, se puede decir que estas "nueva" tecnologías han ayudado de manera notable al desarrollo de la ciencia.

Pese a que hoy en día se puede decir que es cuando se ha conseguido desarrollar versiones verdaderamente útiles de estos, su estudio más arcaico se remonta a siglos atrás, como podemos observar por ejemplo en los llamados autómatas. Algunos acabaron como simples objetos curiosos, pero aun así, no cabe duda de que estos eran unas pequeñas obras de ingeniería y artesanía, y fueron otra muestra del interés del ser humano por el estudio de la naturaleza, ya que la mayoría eran representaciones de seres vivos los cuales simulaban su conducta, o incluso en los más complejos, imitaban acciones humanas como los creados por Pierre Jaquet Droz en el siglo XVIII.

El estudio e imitación de la naturaleza es algo que ha evolucionado con esta tecnología, y que se sigue desarrollando en la llamada biomimesis y la biónica. Uno de los resultados más destacables fueron las llamadas redes neuronales, las cuales surgieron como estudio del cerebro humano y de sus capacidades, siendo estas la base del que será el proyecto actual.

1.1 Motivación

Este proyecto trata una amplia variedad de temas, como la visión por computador, el desarrollo de redes neuronales, e incluso su posible aplicación a robots aéreos. A todo lo anteriormente comentado, se a de añadir que se trata de un sistema bioinspirado, lo cual añade profundidad a su desarrollo, y a la búsqueda de información de los organismos en los que se basa. El objetivo es poder presentar como se desarrolla al menos parte de este tipo de investigaciones, y mostrar su evolución desde la realización de experimentos, hasta la obtención del sistema final.

Con respecto a la motivación personal, siempre ha existido un gran interés hacia la aplicación de la ingeniería a los organismos vivos, y lo relativo al área de la biónica. Ya sea en un entorno más orientado a la medicina, como la fabricación de maquinaria destinada a intervenciones o el desarrollo de prótesis; como el centrado en la creación de sistemas físicos o virtuales, basándose en alguna cualidad de un ser vivo existente, como es el caso de este proyecto. En cualquier caso siempre me ha fascinado como la naturaleza ha evolucionado hasta lo es hoy en día, y descubrir lo que aun podemos aprender de ella, permitiéndonos avanzar siempre intentado no dañarla en el proceso.

1.2 Objetivos

El objetivo de este proyecto es conseguir replicar la funcionalidad de los ocelos dorsales, presentes en muchos tipos de insectos. En concreto se pretende conseguir un sistema, el cual mediante las lecturas que reciben estos órganos, pueda obtener las velocidades angulares a las que se somete el organismo durante su movimiento.

Para llegar a este punto se estudiaran una serie de experimentos, desarrollados mediante la simulación de los ocelos a través de cámaras con una determinada disposición. Una vez comprobados, se precederá al desarrollo del sistema mediante redes neuronales convolucionales, permitiendo optimizarlo, de cara al uso de imágenes en el desarrollo del mismo.

Previamente al desarrollo se estudiaran en profundidad las redes neuronales y sus tipos, para poder entender completamente su método de aplicación, e intentar conseguir el mejor resultado posible de cara a su precisión en los resultados.

Dado que este es un proyecto, que se encuentra en desarrollo por uno de los equipos de investigación de la universidad de Sevilla, en conjunto con la universidad Pablo Olavide, se parte de una base ya establecida, y por lo tanto en este caso, tras comprobar el estado actual de los sistemas desarrollados, nos centraremos en la mejora de los resultados ya existentes.

1.3 Historia y estado del arte

Como se ha comentado anteriormente, el estudio de la naturaleza y su imitación, es una técnica que siempre a acompañado al ser humano. Esta ha influido en todas sus áreas de desarrollo, ya sea desde las más obvias como pueden ser el arte y la música, hasta la ingeniería, sobre la cual trataremos.
A lo largo de la historia el problema de estas soluciones eran las limitaciones para aplicarlas, es decir, hechos como el vuelo de las aves, la respiración de los peces bajo el agua, o como podían emitir luz las luciérnagas eran sucesos sin una explicación firme, y en muchos casos basada en la imaginación del que la narraba. No fue hasta el desarrollo de los principios físicos, que el ser humanos se decidió por soluciones alternativas, ya que aunque de manera natural no podíamos volar, ni respirar bajo el agua, tal vez si podríamos encontrar soluciones alternativas al respecto. De esta forma la intención de imitar a la naturaleza, cambió a un deseo de entenderla. Todo este movimiento empezó a tomar fuerza en el siglo XVI, donde la época del renacimiento dio cabida a algunos del los científicos e ingenieros más reconocidos históricamente, cuyas descubrimientos fueron los primeros pasos hasta los avances que tenemos hoy en día. Tras esto, a medida que nuestro conocimiento sobre el mundo avanzaba, también aumentaban la cantidad de ideas que surgían al respecto, hasta tal punto que en algunos casos se intentaba desafiar los límites de esta, como fue el intentar devolver la vida a cuerpos inertes, estimulando su sistema nervios con electricidad, hecho que inspiro el libro de Mary Sheller de “El Moderno Prometeo”, o como lo conocemos más actualmente “Frankenstein”. A partir de este punto ya fuera por accidente o tras una ardua investigación, se realizaron descubrimientos que seguimos usando hoy en día, como puede ser el estudio de como el fruto de la bardana o *Arctium lappa*, se adhería al pelaje de los animales y la ropa, lo que dio como resultado el velcro, o de manera más reciente tras el estudio de las neuronas, la lógica umbral, la cual fue la base para el desarrollo de las redes neuronales y la inteligencia artificial.

Ya centrándonos en las redes neuronales, casi paralelamente al desarrollo de la lógica umbral, se crearon varias hipótesis de aprendizaje, donde podemos destacar el aprendizaje de Hebb basado en el mecanismos de plasticidad neuronal. Algunos de estos fueron aplicados a modelos computacionales, esperando conseguir resultados de simulación de procesos, como la llamada maquina de Turing, la cual podía ser adaptada para simular la lógica de cualquier algoritmo de computador. En todo caso la estructura de estas maquinas, en el aquel entonces llamadas calculadoras, distaba de como se desarrollan las redes neuronales hoy en día. En 1958, Frank Rosenblatt creo el modelo de perceptron y la descripción de su circuitería, aunque no fue hasta 1975 con el desarrollo del algoritmo de propagación hacia atrás cuando realmente se pudo procesar por redes neuronales. Los métodos de clasificación que se desarrollaron a partir de esto ganaron popularidad en el aprendizaje automático, y más con el desarrollo del max-pooling, el cual ayudo al reconocimiento de objetos tridimensionales; y la implementación de estas para una unidad de procesamiento gráfico (GPU), lo cual aceleró el proceso de entrenamiento. Estos últimos avances dio lugar a una variación del perceptron multicapa, llama redes neuronales convolucionales, las cuales resultaron ser muy efectivas en tareas de visión artificial como la que nos acontece, debido a que su aplicación se realiza en matrices bidimensionales.

Con respecto al estado actual de las redes neuronales, se puede decir que estas se han abierto un hueco en muchas de las áreas de la tecnología y la innovación. A continuación se expondrán algunas de los proyectos

más actuales junto con su área de estudio:

- Medicina: La detección temprana de una enfermedad puede ser un factor clave en el tratamiento, por lo que en pruebas en las que puede resultar clave el reconocimiento de patrones específicos, la redes neuronales han resultado ser de gran utilidad, mostrando posibles problemas en muestras de tejidos o biopsias, e incluso en radiografías, otorgando diagnósticos que no se podrían obtener a simple vista.
- Militar: Dentro de este ámbito, cobran especial importancia las aplicaciones que puedan influir o analizar situaciones en el campo de batalla de manera inteligente, es decir, los vehículos no tripulados de ataque, ya sea aéreos o terrestres, pueden ser diseñados para portar cámaras de visión artificial con la finalidad de que actúen de forma más eficiente para contrarrestar las fuerzas enemigas. Esta tecnología que puede ser aplicada a otro tipo de armas dando como resultado las llamadas armas inteligentes.
- Producción y revisión de maquinaría: La flexibilidad de las redes neuronales con respecto a diferentes propósitos, hacen que se suelan poder aplicar al mundo de la robótica, y aun más en la industria, donde abundan los procesos repetitivos, los cuales pueden ser fácilmente sustituidos por maquinaría. Debido a esto, se pude ver porque muchas fabricas poseen robots equipados con diferentes sensores y/o visión artificial, permitiendoles realizar el ensamblaje o la operación de manera más eficiente y productiva, y no solo esto, ya que muchos sistemas implementados además pueden detectar fallos internos, deteniendo su producción y realizando el aviso correspondiente.
- Mercado financiero: las aplicaciones en esta área son de las más interesantes, ya que en un mundo tan cambiante con es el mercado de valores, el poder predecir el comportamiento de las acciones mediante diferentes análisis las hacen una herramienta de gran valor para las empresas en su ámbito financiero, pudiendo generar además modelos de riesgo para inversiones o incluso predicción de tipos de interés.

Como queda patente, la versatilidad de las redes neuronales es uno de los grandes factores que las vuelven de importancia en el desarrollo tecnológico actual, donde el margen de evolución que todavía queda abierto puede dar lugar aplicaciones que aun desconocemos, y que pueden suponer un gran avance para la tecnología y la ciencia en general.

2 Estudio y Simulación de los Ocelos

Con el fin de comprender mejor la naturaleza de estos órganos, a continuación se procederá a realizar una explicación de su funcionamiento y estructura, indicando su diferencia con otros órganos de visión, y su funcionalidad basada en varias investigaciones realizadas al respecto. Tras esto, se mostrará la solución diseñada por el equipo de investigación, para la simulación de los ocelos, indicando superficialmente, como se realizaron los experimentos que se tomaran como entrenamiento para la red.

2.1 Visión de los insectos

En el conocimiento del publico general, se suele saber que la visión los insectos y de otros animales es claramente diferente a la nuestra, y por lo tanto también la percepción del entorno. En el caso de los insectos presentan dos tipos de ojos, los llamados ojos compuestos y los ojos simples u ocelos. Cada uno recibe la información visual de distinta manera, y el uso en conjunto de estos le permite al insecto reaccionar a diferentes estímulos, que pueden ser cruciales par su supervivencia, como localizar alimentos o presas, o detectar peligros. A continuación se expondrán las cualidades de cada uno, y se definirá su proceso de visión, intentando aclarar en esta, todos los términos que puedan resultar ajenos a los lectores no relacionados con esta area.

2.1.1 Ojo compuesto

De forma general los insectos tienen dos ojos de este tipo, siendo órganos sensoriales mucho más complejos que los ocelos. Estos están formados por la agrupación de omatidios, los cuales se tratan de unidades fotorreceptoras capaces de distinguir entre la presencia y ausencia de luz, y en algunos casos distinguir entre colores. Estos están formados por celulas alargadas, de las cuales la capa externa contiene una lente corneal, bajo la que se posiciona un cono cristalino. Los omatidios a su vez se encuentran separados por una capa de pigmento que le permite comportarse como un ojo separado, variando mucho su número según la especie (las mosca común cuenta con unos 4000, las mariposas entre 10000 y 30000, y las libélulas más de 40000). Podemos distinguir dos tipos básicos de ojos compuestos según como registren las imágenes:

- Ojos de aposición: Este tipo adquiere distintos fragmentos de una misma imagen en cada omatidio, fusionándose posteriormente en el cerebro.
- Ojos de superposición: Estos abarcan dos formas de captura de imágenes, pudiendo diferenciar entre los refractantes, en los cuales la retina toma una imagen completa de la luz enfocada por cada omatidio, y los reflectantes donde cada omatidio toma una imagen completa de la luz enfocada.

Cada omatidio apunta en una dirección ligeramente diferente, proporcionando un campo visual muy amplio, a lo cual se suma que estos se activan al captar movimiento, emitiendo una señal intermitente y poniendo al insecto en alerta, lo cual nos muestra porque resulta tan difícil atraparlos.
Como curiosidad final, cabe indicar que los ojos compuestos no cuentan con parpados que los protejan, hecho que obliga a muchos insectos a limpiarlos continuamente. Esta es una acción fácilmente observable, ya que por ejemplo casi todo el mundo, a observado alguna vez a una mosca realizando esta acción, pasando sus patas delanteras por su cabeza.

Figura 2.1 Ojo compuesto.

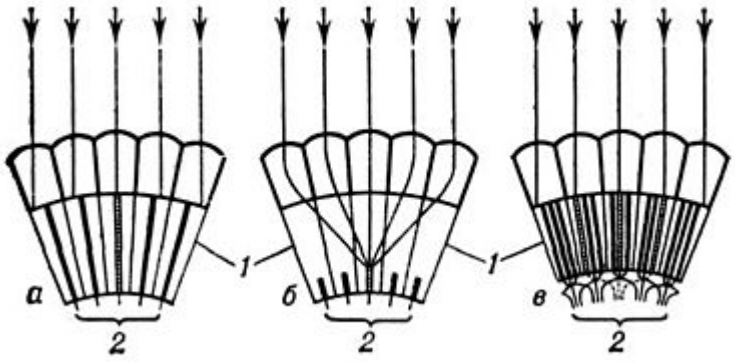

Figura 2.2 Sección de ojo compuesto con omatidios.

2.1.2 Ojo simple u ocelo

Son estructuras muy rudimentarias, careciendo incluso de mecanismo de enfoque. Podemos distinguir entre los ocelos dorsales, que son los más comunes encontrados en las formas adultas de los insectos, y sobre los que se realizará el estudio; y los ocelos laterales que existen solo en algunas especies de larvas. Normalmente estos se presentan en una agrupación de tres, dos laterales y uno central, aunque algunos insectos terrestres como las cucarachas solo presentan dos. Independientemente de la especie los ocelos tienen dos caracterias que suelen estar presentes:

- La refracción de la lente no es suficiente para que se frome una imagen en las células fotorreceptoras.
- La mayoría de las células fotorreceptoras transmiten la información a una sola neurona de segundo nivel en la cual convergen.

Estas dos características concluyen que los ocelos no pueden distinguir las formas, ocupándose unicamente de detectar la presencia de luz y su intensidad. Además debido a su baja refracción, gran apertura de lente y estructura nerviosa, se conoce que son más sensibles a la luz y proporcionan una respuesta mas rápida que los ojos compuestos. Por último y como base en la cual esta desarrollado el proyecto, una de las teorías existentes supone que una de sus funcionalidades es mantener la estabilidad en el vuelo, sobre la cual se ha demostrado que las libélulas y langostas realizan correcciones como reacción a la luz.

Figura 2.3 Ocelos dorsales.

2.2 Simulación de los ocelos

El diseño del equipo de investigación se basó en la estructura de las Drosophilas, un genero de pequeñas moscas más conocidas como "moscas de la fruta", en las que cada ocelo contiene 40 fotorreceptores.
La estructura preparada consistió en tres pequeñas cámaras, dispuestas según la geometría de los ocelos de las Drosophilas. Estas fueron adaptadas para asemejarse al órgano biológico, de forma que se redujo la resolución para obtener imágenes de únicamente 80 píxeles, y se modificó la óptica para conseguir un amplio campo de visión en cada cámara.
A cada cámara se le agrego una unidad de medida inercial(UMI), con el fin de conseguir un punto de tierra verdadero en las medidas de la velocidad angular. Esta integraba un giroscopio, un acelerómetro y un magnetómetro, todos de tres ejes, para poder conseguir una velocidad angular suave y sin bias, datos que se usaran para entrenar la red.
A la configuración anterior se le añadió un conjunto de marcadores infrarrojos pasivos, los cuales son usados por un sistema de seguimiento, para estimar la posición en los 6 grados de libertad del sistema, comprobando doblemente la orientación de la UMI.
Durante el proceso de experimentación todos esto datos eran capturados y grabados usando ROS(Robot Operating System), de forma que a cada conjunto de seis imágenes (dos por cada cámara, ya que registran dos puntos temporales, el presente y la muestra pasada), se le asociaba la posición, orientación y velocidad angular correspondiente.
Por último cabe puntualizar, que los fotogramas por segundo están fijados a 30Hz, lo cual aunque no es un problema en la suavidad de rotación, limita la velocidad angular máxima que puede registrar, debido a que el sistema esta basado en el análisis temporal de las imágenes. Esto sin embargo no invalida la aproximaciones realizadas, ya que en el mercado se pueden encontrar cámaras más rápidas de baja resolución.

Figura 2.4 *Drosophila Melanogaster*.

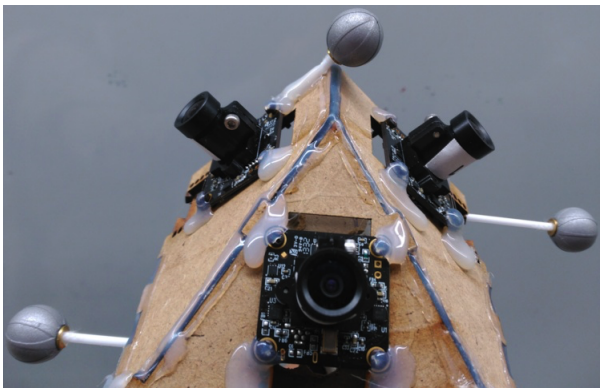

Figura 2.5 Hardware de la simulación.

3 Hardware y Software

En este apartado se especificarán los dispositivos y el software utilizado para el desarrollo del proyecto, incluyendo especificaciones técnicas del equipo e indicando el sistema operativo del entorno de desarrollo.

3.1 Hardware

El hardware utilizado en el desarrollo del proyecto se trata de un portátil Asus, en concreto el modelo GL553VD-DM468T. Dentro de las especificaciones cuenta con procesador intel i7-7700HQ de 2.8 GHz hasta 3,8 GHz, 12 GB de memoria RAM, y un controlador gráfico NVIDIA GeForce GTX1050ti.

EL controlador gráfico es de especial interés dentro del desarrollo del proyecto, ya que en conjunto al software de desarrollo utilizado, este permite optimizar el proceso de aprendizaje, ahorrando un tiempo considerable a la hora de realizar pruebas en la estructura de la red.

3.2 Software

Con respecto al software y más concretamente el sistema operativo, este proyecto se comenzó a desarrollar en una maquina virtual con Ubuntu 14.04. Se optó por este sistema con el objetivo de aprovechar las ventajas que aportaba el entorno, pero al tratarse de una máquina virtual, el procesamiento de la cantidad de datos necesaria para el entrenamiento, lo volvía demasiado lento haciéndolo tedioso en algunos momentos. Por esta razón, se decidió trasladar los programas al sistema operativo principal del equipo, Windows 10, y ya de paso investigar sobre las posibilidades existentes para acelerar los entrenamientos.

El lenguaje de programación elegido para trabajar fue Python 3.7, haciendo uso de la distribución Anaconda. Esta se eligió debido a que incluía casi todas las librerías que serían necesarias para el proyecto, y ademas facilitaba la manipulación de los entornos de trabajo, los cuales iban a ser necesarios.
Una vez decidido el lenguaje, se decidió usar Keras como biblioteca para el desarrollo de las redes neuronales. Como cualidades destacan que es capaz de ejecutarse sobre TensorFlow, Microsoft Cognitive Toolkit, o Theano, además de ser modular, extensible, y amigable para el usuario.
De todos las bibliotecas sobre las que podía ejecutarse Keras se eligío TensorFlow. Esta no se encuentra incluida entre las disponibles en Anaconda inicialmente, pero eso finalmente resultó ser una ventaja. Todo debido a que de esta forma, era más sencillo hacer uso de la oportunidad que da esta biblioteca, la cual permite poder acelerar sus procesos mediante una instalación alternativa, utilizando la GPU en lugar de la CPU para realizar los cálculos.
Es importante destacar que elegir esta opción no siempre es posible, ya que al menos para la fecha en la que se realiza esta memoría, solo se permite si se tiene instalada una tarjeta gráfica de NVIDIA, y dentro de ellas, unicamente se puede realizar con los modelos indicados en la web de las mismas.
Dicho esto, si se posee la una de las tarjetas listadas, se puede proceder a la instalación siguiendo los pasos indicados, y tras esto configurar un entorno de trabajo como el que se realizó para el sistema. Cabe destacar que una vez instalado, no presenta ninguna diferencia con respecto al trabajo con el código, más alla de la

velocidad a la que realiza los entrenamientos.

Para finalizar, otras librerías que fueron de vital importancia dentro de python, fueron por ejemplo NumPy, la cual aporta un mayor soporte para las matrices y vectores, facilitado su manipulación; y Scipy la cual agrega herramientas y algoritmos matemáticos que serán necesarios para trabajar.

4 Estudio de las Redes neuronales

Como ya ha quedado claro en los planteamientos anteriores, las redes neuronales serán el medio para el desarrollo del sistema propuesto, pero para la correcta comprensión de su uso será necesaria más información. El problema es que hasta este momento, solo se ha realizado un breve resumen de su historia sin entrar en mas detalles teóricos, y comenzar a realizar el programa sin este conocimiento, dejaría la capacidad de desarrollo reducida a los ejemplos que se puedan encontrar al respecto. Por este motivo, la razón principal de este capitulo, es entender la lógica tras funciones de las se hará uso en la implementación del modelo, y además poder ver sus diferentes tipos y elementos.
Se comenzará con la presentación del perceptrón y sus características, para luego proseguir con sus evoluciones y mejoras realizadas, hasta el modelo de las redes neuronales convolucionales que se usarán en este proyecto. En este desarrollo también se incluirán los algoritmos de aprendizaje utilizados para el trabajo con las redes.

4.1 La neurona artificial

El perceptrón o neurona artificial, es la unidad básica a partir de la cual se pueden crear redes neuronales artificiales más complejas. Fueron creadas por Frank Rosenblatt, el cual se inspiró en el trabajo de Warren McCulloch y Walter Pitts.
El perceptrón toma como entrada un vector binario x con elementos $x_1, x_2, ...x_n$, y produce una sola salida binaria. Para calcularla Rosenblatt introduce el concepto de pesos ($w1, w2, ...wn$), los cuales representan a un número real que expresa la relevancia de su entrada respectiva, con la salida. El resultado de esta, dependerá de si la suma de los productos escalares de las entradas por sus pesos, es mayor o menor de un determinado umbral u. Una vez averiguado el valor, el primer modelo desarrollado de esta utilizaba una función de activación de tipo escalón, la cual entrega el valor binario correspondiente. La fórmula a la que responde es la siguiente:

$$f(x) = \begin{cases} 1 & \text{si } \sum_{i=1}^{n} w_i x_i - u > 0 \\ 0 & \text{en otro caso} \end{cases} \tag{4.1}$$

La función de activación puede variar, existiendo varios tipos que pueden aportar diferentes resultados, según la red que se quiere construir. En una sección más adelante se mostrarán otros tipos disponibles, y sus diferentes aplicaciones.
Una vez puntualizado el tema de la función de activación, se muestra el diagrama general del perceptrón o neurona artificial:

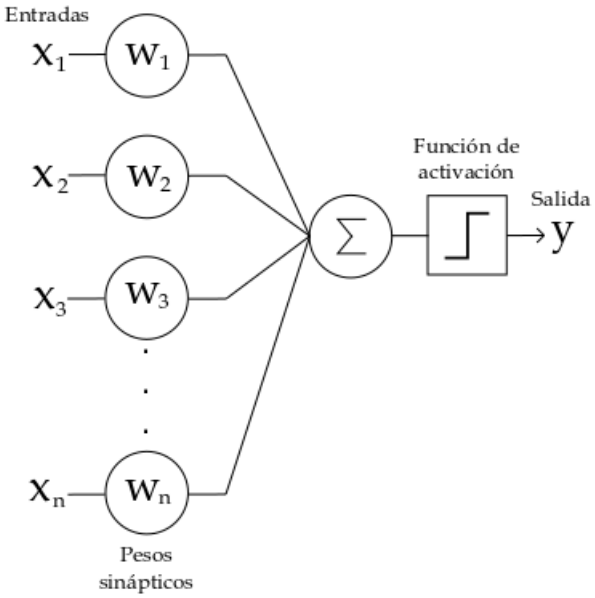

Figura 4.1 Diagrama de una neurona artificial.

Por último, hay que indicar que al igual que en el caso de las neuronas biológicas en las que se inspiran, las neuronas artificiales solo muestran su verdadera utilidad, cuando se asocian con otras formando un red, extendiendo así la señal que se capta como entrada a través de ellas. Para ver realmente la similitud de entre las dos, se expondrá brevemente la estructura de la neurona biológica.
Los canales de entrada y salida de estas, son llamados dendritas y axones respectivamente. Las dendritas recogen información de la zona en la que se encuentran, y la envían al cuerpo de la neurona o soma. Este procesa la información y envía la respuesta por medio del axón, que la transmite al resto de neuronas conectadas mediante la llamada sinapsis. A continuación se muestra una imagen de su estructura para facilitar su comprensión:

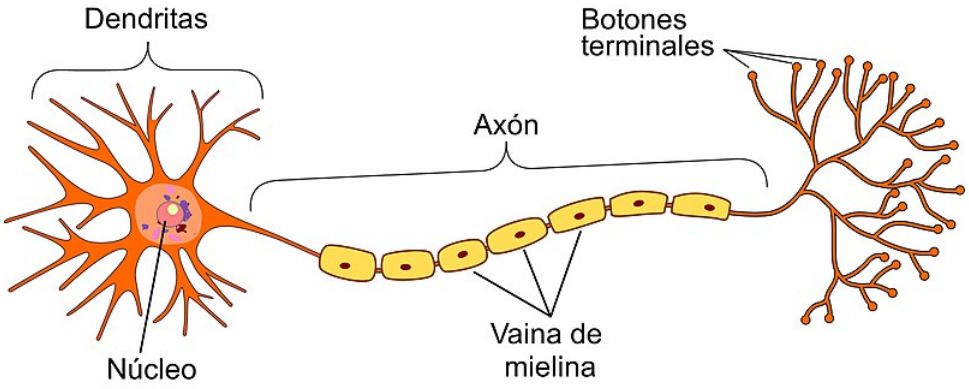

Figura 4.2 Estructura de una neurona.

4.2 El perceptrón multicapa

El perceptrón multicapa es una evolución del concepto de perceptrón, donde se unen varias neuronas en lugar de una sola. Esto permite resolver problemas no lineales, lo cual no era posible con el perceptrón simple. También se introduce el concepto de capas, diferenciando entre tres tipos diferentes:

- Capa de entrada: Formada por las neuronas que introducen los patrones de entrada en la red, sin embargo no se produce procesamiento en ellas.
- Capas ocultas: Son las neuronas que recogen los datos de capas anteriores como entrada, y sus salidas pasan a capas posteriores.
- Capa de salida: Los valores de salida de estas neuronas son las salidas de toda la red.

Hay que destacar que en el momento de creación de este tipo de red, los valores de entrada y salida de la red aun eran binarios, y los valores de los pesos eran asignados manualmente. Una vez que se incluyó el algoritmo de propagación hacia atrás o Backpropagation, fue posible entrenar las de manera supervisada, haciendo que estas redes también fuesen conocidas como red de retropropagación.

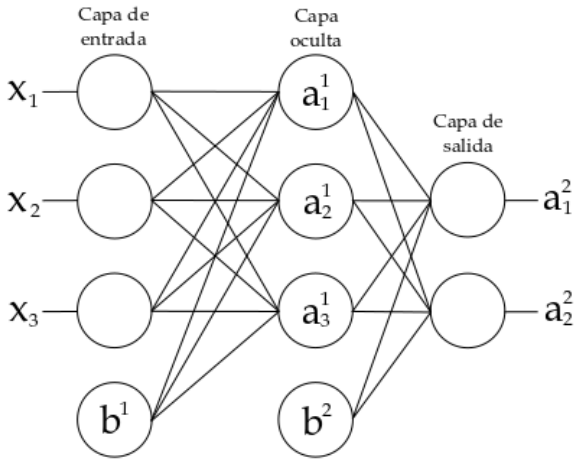

Figura 4.3 Diagrama de un ejemplo de perceptrón multicapa.

4.3 La neuronas sigmoideas

Hasta este punto las neuronas desarrolladas solo permitían el trabajo con valores binarios, pero la llegada de las neuronas sigmoideas cambia este hecho. Estas son parecidas al perceptrón, pero permiten como entrada valores de números reales.
En este momento también se incluye la neurona "bias", la cual permanece siempre activa (a 1). Esta puede ser representada como x_0 y permite implementar el umbral utilizado en el perceptrón original como un peso más ($-w_0$), simplificando los sistemas de entrenamiento que la acompañarían.
Con la inclusión de estos dos elementos, las salidas de las redes no estarían limitadas a una respuesta binaria, sino que vendrían definidas por la expresión $f(\sum_{i=1}^{n} w_i x_i + b)$ donde b equivaldría a $x_0(-w_0)$, y f equivaldría a la función sigmoide, es decir:

$$f(z) = \left(\frac{1}{1+e^{-z}}\right) \tag{4.2}$$

Seguidamente se mostrará la respuesta de la función:

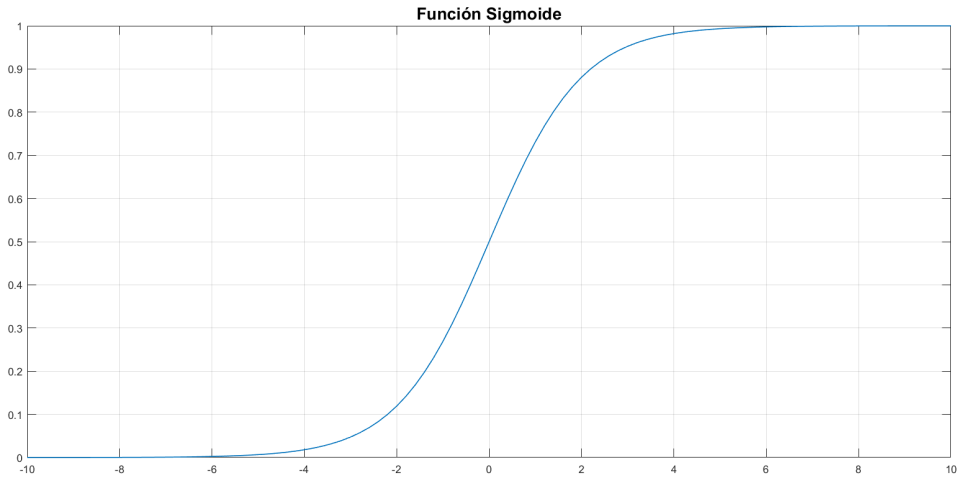

Figura 4.4 Función sigmoide.

Dado que en el perceptrón clásico se hacia uso de una función escalón para la salida debido a las limitaciones, esta podría clasificarse como la primera función de activación como tal desarrollada. A esta también le siguieron otras de funciones de activación también muy utilizadas, como la ReLu o la función tangente hiperbólica cuyos algoritmos y respuesta se pueden ver a continuación.

- Función ReLu:

$$f(z) = max(0, z) \tag{4.3}$$

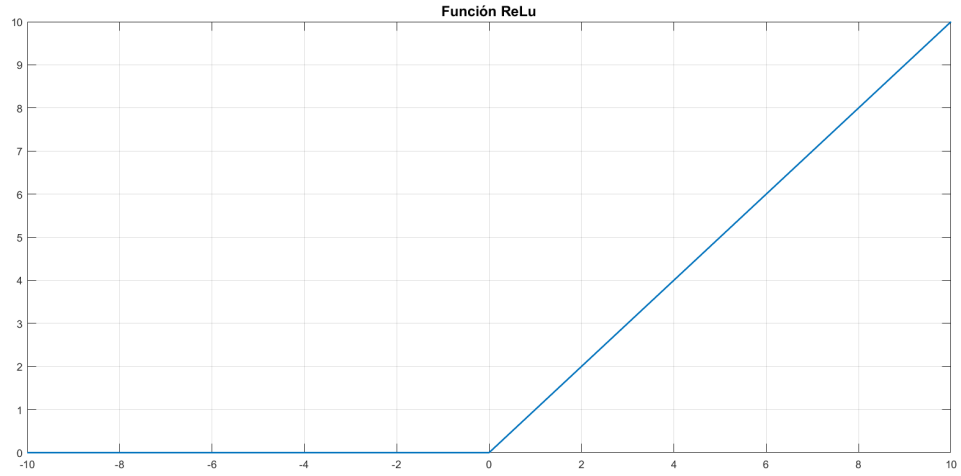

Figura 4.5 Función ReLu.

- Función tangente hiperbólica:

$$f(z) = tanh(z) = \left(\frac{e^z - e^{-z}}{e^z + e^{-z}}\right) \tag{4.4}$$

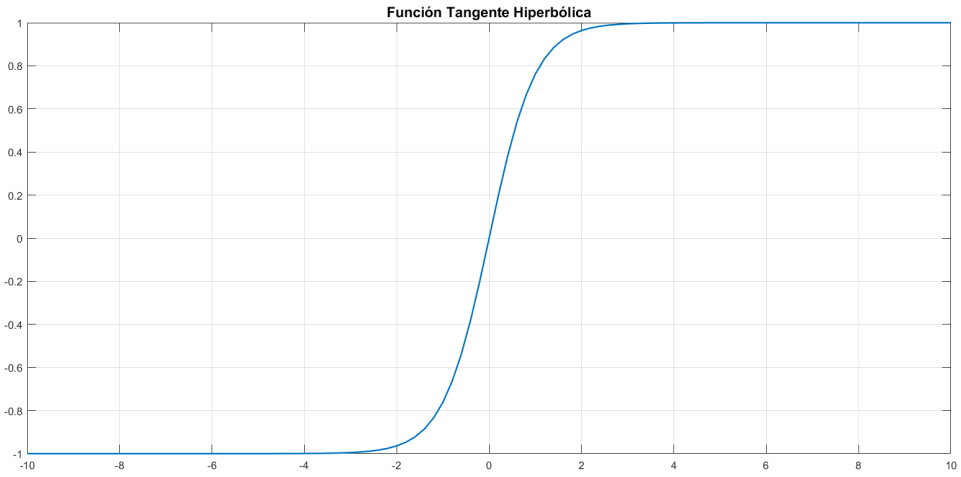

Figura 4.6 Función tangente hiperbólica.

4.4 Redes Feedforward

Una red feeforward, es una red neuronal artificial donde el flujo de datos solo avanza en una única dirección, desde la capa de entrada atravesando las capas ocultas, hacia las de salida. Esto quiere decir que no hay ningún tipo de bucle o ciclo, sino que se siempre se "alimenta" con valores a las capas hacia adelante, aunque si existe este tipo de comportamiento en las llamadas redes neuronales recurrentes.
La red neuronal prealimentada fue la primera y más simple red neuronal, donde también describió el concepto de "fully connected layer", donde todas las neuronas de una capa, estan conectadas a todas las neuronas de las siguiente.

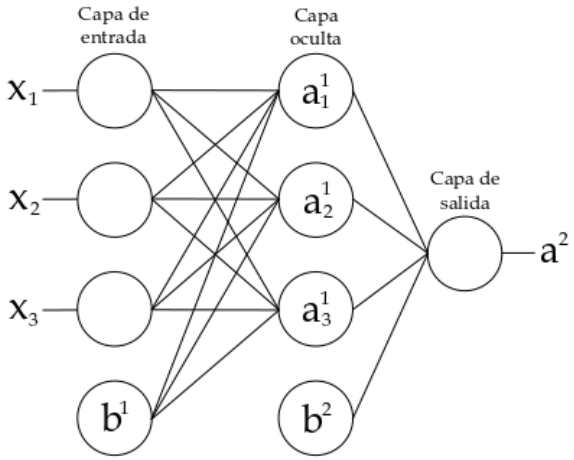

Figura 4.7 Ejemplo de una red Feedforward.

El cálculo de la salida de cada capa sigue haciendo uso de la ecuación vista en la sección de las neuronas sigmoideas, pero en este caso será necesario añadir varios indices para poder hacer referencia a la capa y neuronas con respecto a las que se calcula. La formula entonces será representada como:

$$z_i^{(l)} = \sum_{j=1}^{n} w_{ij}^{(l)} a_j^{(l-1)} + b^{(l)} \tag{4.5}$$

El indice l representa la capa con la que se esta trabajando, y los indices i y j indican respectivamente el elemento de salida para el que se calcula la suma ponderada, y los elementos del conjunto de parametro w y entrada a que se usará en cada paso del sumatorio para su calculo. En concreto para w estos indices se refieren a que el elemento j de los pesos, es aplicado al elemento i de la salida a, asemejando así la relación de "conexión" entre ellos.
La variable a como ya se ha indicado, es la entrada recibida por la capa, la cual se identifica como la salida o también llamada activación de la capa anterior. En el caso de que la capa anterior sea la de entrada, esta se corresponde con los datos de entrada de la red.
Por último b es el parámetro bias perteneciente a la capa para la cual se calcula la salida.

La ecuación anterior sin embargo no es la salida como tal de la capa, ya que aun es necesario aplicar la función de activación. Basándonos en la ecuación anterior (4.5), esta se calcula como:

$$a_i^{(l)} = f(z_i^{(l)}) \tag{4.6}$$

Como se puede ver, f representa a la función de activación, que es aplicada a la suma ponderada de las entradas de la neurona más el bias. Y a por su parte representa a la salida real o activación de esta, la cual como ya hemos visto se podrá aplicar como entrada de la capa siguiente. A continuación se presenta el cálculo de la salida de la red de la imagen 4.7 como ejemplo.

$$a_1^{(1)} = f(w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + w_{13}^{(1)} x_3 + b^{(1)}) \tag{4.7}$$
$$a_2^{(1)} = f(w_{21}^{(1)} x_1 + w_{22}^{(1)} x_2 + w_{23}^{(1)} x_3 + b^{(1)}) \tag{4.8}$$
$$a_3^{(1)} = f(w_{31}^{(1)} x_1 + w_{32}^{(1)} x_2 + w_{33}^{(1)} x_3 + b^{(1)}) \tag{4.9}$$
$$a_1^{(2)} = f(w_{11}^{(2)} a_1^{(1)} + w_{12}^{(2)} a_2^{(1)} + w_{13}^{(2)} a_3^{(1)} + b_1^{(2)}) \tag{4.10}$$

Cabe destacar que en el calculo de las salidas en las ecuaciones 4.7, 4.8 y 4.9, al tratarse de la primera capa los datos recibidos serán los de entrada de la red, de ahí que se hiciera uso de x para representarlos en lugar de a como se ha visto en la ecuación general.

Por último para esta sección, hay que indicar que las ecuaciones anteriores se pueden reescribir de forma compacta si se representan de forma vectorial. Así las ecuaciones 4.5 y 4.6 quedarían de la siguiente manera:

$$z^{(l)} = w^{(l)} a^{(l-1)} + b^{(l)} \tag{4.11}$$
$$a^{(l)} = f(z^{(l)}) \tag{4.12}$$

4.5 Algoritmo de Backpropagation

El algoritmo de propagación hacia atrás o backpropagation, permitió el entrenamiento de las redes neuronales de múltiples capas de manera supervisada. Este se realizaba en dos fases, primero se aplicaba un patrón como estimulo a la entrada de red, atravesando este todas las capas hasta llegar a las salidas generando una respuesta; en segundo lugar la respuesta obtenida se comparaba con la deseada, y se calcula la señal de error para cada una de las salidas.
Las señales de error se propagan hacia atrás, hacia las neuronas de la capa oculta directamente anterior a la

salida, a la que contribuyen. El error que reciben sin embargo no es la señal total, sino una proporción basada en la contribución relativa de cada una de ellas con respecto a la salida original. Todo este procedimiento se repite capa por capa, hasta llegar a la entrada, donde todas la neuronas deben haber recibido su señal de error en función de su porcentaje de participación en el error total.
De esta manera se van realizando pequeños ajustes en cada iteración, organizándose las neuronas a si mismas para reconocer distintas características de las entradas. Tras este entrenamiento la red podrá clasificar los datos que reciba, aunque estos contengan algún tipo de ruido o se encuentren incompletos, reconociendo los patrones que puedan contener que se asemejen a las características que hubiesen sido identificadas durante el proceso.

Para conseguir entrenar a la red, será necesario ajustar los parámetros de cada capa, es decir los pesos (w) y el bias (b). Estos se conseguirá en la última capa través de las siguientes fórmulas:

$$w_i^{(L)} = w_i^{(L)} + \alpha \frac{\partial C}{\partial w_i^{(L)}} \tag{4.13}$$

$$b^{(L)} = b^{(l)} + \alpha \frac{\partial C}{\partial b^{(L)}} \tag{4.14}$$

la variable α corresponderá al ratio de entrenamiento, y C será la función de coste, con la que se calcula el error entre la salida obtenida por parte de la red y la deseada. Para C normalmente se hace uso de la función del error cuadrático medio, representándose entonces de la siguiente forma:

$$C(a_i^{(L)}) = \frac{1}{2}\sum_i (y_i - a_i^{(L)})^2 \tag{4.15}$$

La función de coste calcula el error entre las activaciones de la ultima capa ($a^{(L)}$), y las salidas (y) registrada en el set para entrenamiento comparándolas. Por lo tanto esta será la señal de error total que se propagará por la red.

Una vez conocidos los elementos de las ecuaciones 4.13 y 4.14, se procederá a explicar como calcular la deriva parcial presente en ambas, aplicando la lógica del algoritmo de propagación hacia atrás.
Los parámetros de la capa no son utilizados directamente en el calculo del error, sino que en primer lugar son aplicados para hallar la suma ponderada (z^L), y tras esto obtener la activación de la capa. De esta manera los parámetros se relacionan con el coste a través de la composición de funciones $C(a(z^{(L)}))$, siendo necesario aplicar la regla de la cadena para el obtener la derivada parcial propuesta.
Teniendo en cuenta la composición anterior la derivada se puede descomponer como:

$$\frac{\partial C}{\partial w^{(L)}} = \frac{\partial C}{\partial a^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial z^{(L)}}{\partial w^{(L)}} \tag{4.16}$$

$$\frac{\partial C}{\partial b^{(L)}} = \frac{\partial C}{\partial a^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial z^{(L)}}{\partial b^{(L)}} \tag{4.17}$$

El primer componente ($\frac{\partial C}{\partial a^{(L)}}$) representa como varía el error de la red cuando variamos la salida en la última capa. Si la función de coste como hemos visto antes es la del error cuadrático medio, esta corresponderá con su derivada como se ve a continuación.

$$\frac{\partial C}{\partial a_i^{(L)}} = (a_i^{(L)} - y_i) \tag{4.18}$$

El segundo componente ($\frac{\partial a^{(L)}}{\partial z^{(L)}}$) indica como varía la salida de la neurona cuando variamos la suma ponderada de esta. Corresponderá con la deriva de la función de activación, que si es la función sigmoide se obtendrá:

$$a^{(L)}(z^{(L)}) = \frac{1}{1+e^{-z^L}} \tag{4.19}$$

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = a^{(L)}(z^{(L)})(1-a^{(L)}(z^L)) \tag{4.20}$$

Los últimos componentes ($\frac{\partial z^{(L)}}{\partial w^{(L)}}$ y $\frac{\partial z^{(L)}}{\partial b^{(L)}}$) por su parte nos muestran como varía la suma ponderada respecto a la variación de los parámetros. Según la fórmula de la suma ponderada, su derivada con respecto a ambos parámetros resultará como:

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)} \tag{4.21}$$

$$\frac{\partial z^{(L)}}{\partial b^{(L)}} = 1 \tag{4.22}$$

Cabe destacar que como se ve en 4.21, la deriva de la suma ponderada con respecto al parámetro del peso, corresponde con entrada de la neurona, es decir, la activación de la capa anterior.

Al calcular las tres derivadas parciales, podemos observar que el conjunto del primer y segundo componente corresponde a como varía el error en función de la suma ponderada. Esto se referirá a en que grado se modifica el coste (error), cuando se produce un cambio en la suma de la neurona. Si el resultado del conjunto de ambas derivadas es grande, ante un pequeño cambio en el valor de la neurona este se verá muy reflejado en el resultado final. Sin embargo si es pequeño, aunque variásemos el resultado de la suma este no afectaría al valor de la red.
Debido a lo que implica este conjunto de las derivadas parciales, se identificará con el error imputado a la neurona y se representará con $\delta^{(L)}$.

Con todos los datos anteriores ya podemos resolver las derivadas parciales del coste con respecto a los parámetros, obteniendo los siguientes resultados:

$$\frac{\partial C}{\partial w^{(L)}} = \delta^{(L)}\frac{\partial z^{(L)}}{\partial w^{(L)}} = \delta^{(L)}a^{(L-1)} \tag{4.23}$$

$$\frac{\partial C}{\partial b^{(L)}} = \delta^{(L)}\frac{\partial z^{(L)}}{\partial b^{(L)}} = \delta^{(L)} \tag{4.24}$$

Una vez resultas las derivas anteriores, ya podríamos calcular los nuevos parámetros con respecto a la capa L (la de salida), pero aun sería necesario hacer los cálculos para las $L-1$ capas restantes. Aplicando la lógica de los casos anteriores a los cálculos para la capa $L-1$ serían:

$$\frac{\partial C}{\partial w^{(L-1)}} = \frac{\partial C}{\partial a^{(L)}}\cdot\frac{\partial a^{(L)}}{\partial z^{(L)}}\cdot\frac{\partial z^{(L)}}{\partial a^{(L-1)}}\cdot\frac{\partial a^{(L-1)}}{\partial z^{(L-1)}}\cdot\frac{\partial z^{(L-1)}}{\partial w^{(L-1)}} \tag{4.25}$$

$$\frac{\partial C}{\partial b^{(L-1)}} = \frac{\partial C}{\partial a^{(L)}}\cdot\frac{\partial a^{(L)}}{\partial z^{(L)}}\cdot\frac{\partial z^{(L)}}{\partial a^{(L-1)}}\cdot\frac{\partial a^{(L-1)}}{\partial z^{(L-1)}}\cdot\frac{\partial z^{(L-1)}}{\partial b^{(L-1)}} \tag{4.26}$$

Identificando los componentes para este caso en 4.24 y 4.25, los dos primeros son iguales que el caso anterior y por lo tanto los podremos sustituir por la variable $\delta^{(L)}$. El tercero por su parte, representa como varía la suma ponderada de la capa cuando se modifica la salida de la neurona de la capa previa, que resolviéndolo corresponde al vector de pesos que conecta ambas capas.

$$\frac{\partial z^{(L)}}{\partial a^{(L-1)}} = w^{(L)} \tag{4.27}$$

Este es el encargo de mover el error de una capa a la anterior, distribuyendo el error en función de cuales son las ponderaciones de las conexiones.

Con los componentes restantes se opera de la misma forma que en los cálculos de la capa L. El cuarto se representa de nuevo con la derivada de la función de activación, en este caso de la capa $L-1$; y el quinto y último es la derivada de la suma ponderada al igual que antes, por lo que se resuelve como 1 con respecto al bias, y a^{L-2} para los pesos ya recibe las salidas de la capa $L-2$.

De esta forma nuevamente tendríamos una expresión con la que obtener los nuevos parámetros, donde podemos identificar los cuatro primeros componentes como el valor de δ para la capa $L-1$, ya que este bloque será el que represente en este caso el error de las neuronas para la capa.

$$\frac{\partial C}{\partial w^{(L-1)}} = \delta^{(L-1)} \frac{\partial z^{(L-1)}}{\partial w^{(L-1)}} = \delta^{(L-1)} a^{(L-2)} \tag{4.28}$$

$$\frac{\partial C}{\partial b^{(L-1)}} = \delta^{(L-1)} \frac{\partial z^{(L-1)}}{\partial b^{(L-1)}} = \delta^{(L-1)} \tag{4.29}$$

Los cálculos realizados en esta capa son extensibles a todas las demás aplicando la misma lógica, lo cual demuestra la eficacia del algoritmo de propagación hacia atrás. A continuación se muestra un resumen de la rutina a seguir para el cálculo de todos los parámetros.

1. Computo del error de la última capa

$$\delta^{(L)} = \frac{\partial C}{\partial a^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \tag{4.30}$$

2. Retropropagación del error a la capa anterior

$$\delta^{(l-1)} = w^{(l)} \delta^{(l)} \cdot \frac{\partial a^{(l-1)}}{\partial z^{(l-1)}} \tag{4.31}$$

3. Calculo de las derivadas de la capa usando el error

$$\frac{\partial C}{\partial w^{(l-1)}} = \delta^{(l-1)} a^{(l-2)} \tag{4.32}$$

$$\frac{\partial C}{\partial b^{(l-1)}} = \delta^{(l-1)} \tag{4.33}$$

4.6 Redes neuronales convolucionales

Son un tipo de red neuronal multicapa que se inspira en el método de visión de los animales. Las neuronas que las forman responden de manera muy similar a las neuronas de la corteza visual primaria de un cerebro biológico. Los fundamentos de estas se basan en el Neocognitrón desarrollado por Kunihiko Fukushima, que posteriormente fue mejorado con la instroducción del algoritmo de propagación hacia atrás para poder entrenar el sistema, y finalmente fueron refinadas al implementarlas para una unidad de procesamiento gráfico.

Las estructuras de estas redes consisten en múltiples capas de filtros convolucionales de una o más dimensiones, aplicando generalmente tras cada capa una función para realizar un mapeo causal no-lineal. La arquitectura de las capas implementan dos procesos, uno de extracción de características y posteriormente uno de clasificación.
La primera fase se asemeja al proceso de estimulación de las células de la corteza visual, estando compuesta por capas alternas convolucionales y de reducción de muestreo. A medida que se procesan los datos en esta fase disminuyen sus dimensiones, de forma las neuronas de las capas más profundas son menos sensibles a perturbaciones, pero a la vez pueden reconocer características más complejas.

Por último, una vez finalizado el proceso de extracción de caracteristicas, al final de la red se disponen un conjunto de neuronas sencillas, que realizan la clasificación final sobre estas.

4.6.1 Capa convolucional

En este tipo de capas dado que están planteadas para el trabajo con matrices, no realizarán el calculo de la salida mediante las neuronas artificiales como tal, sino que pasan a ser una serie de operaciones matriciales. El proceso de convolución usado en este tipo de red dista ligeramente de la operación matemática usual. En primer lugar esta se realiza entre dos tensores, la imagen de dos dimensiones que la capa recibe como entrada, y el kernel, el cual se trata de una matriz de pequeñas dimensiones encargada de detectar diferentes caracteristicas. El proceso en sí consiste en tomar grupos de píxeles cercanos en la imagen, e ir operando matemáticamente con un producto escalar contra el kernel. La formula matemática a la que responde para una imagen de un solo canal es la siguiente:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n) \tag{4.34}$$

En la ecuación I será la matriz de la imagen que se recibe, y K será el kernel usado para la operación, con m y n como sus dimensiones. S por su parte será la nueva matriz que generará el proceso de convolución, y que será la salida de la capa para la que se calcula. Por último i y j indicarán el componente de S para el que se realiza el cálculo, cuyas dimensiones si no se aplica otra técnica corresponderán a:

$$m_S = m_I - m_K + 1 \tag{4.35}$$

$$n_S = n_I - n_K + 1 \tag{4.36}$$

Los subíndices S, I y K indican a que conjunto pertenecen las dimensiones, es decir, la matriz de salida, la imagen de entrada y el kernel respectivamente.
Suponiendo que el kernel es de 2x2, recorrerá la imagen de izquierda a derecha y de arriba a abajo operando sobre secciones de esta del mismo tamaño, generando así una nueva matriz. A continuación se muestra un diagrama de este proceso para su comprensión.

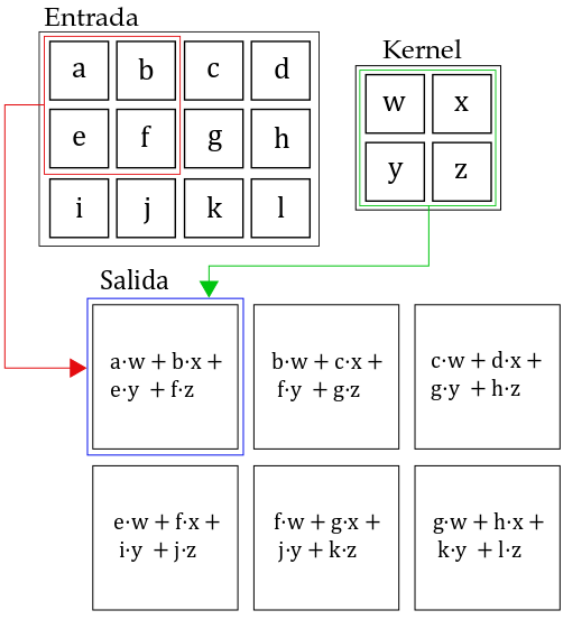

Figura 4.8 Ejemplo de convolución 2D.

Uno de los problemas de este proceso, es que cada vez que se aplica disminuyen las dimensiones de la matriz que se obtiene con respecto a la original, lo cual puede ser muy relevante si las dimensiones del kernel utilizado son grandes. Para solucionarlo se hace uso del "zero-padding", el cual puede controlar las dimensiones de la matriz de salida, añadiendo ceros de forma simétrica en el contorno a la matriz de entrada.

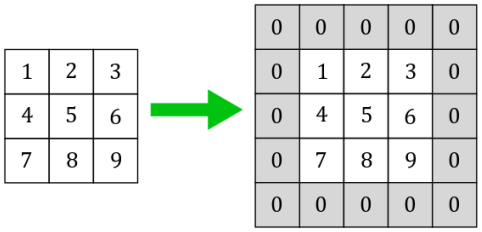

Figura 4.9 Ejemplo de zero-padding.

En la práctica no se aplicará un solo kernel, sino un conjunto de ellos que normalmente reciben el nombre de filtros. El número de ellos que se apliquen definirán la cantidad de matrices que se obtengan como salida, así por ejemplo con 50 filtros y una matriz de salida de 10x10 por cada kernel, se conseguirá un grupo de matrices de 10x10x50 que reciben el nombre de "feature mapping".

Por último una vez conseguidas el conjunto de matrices de convolución, se les aplicará la función de activación seleccionada para la capa. Normalmente para este tipo de capas se acostumbra usar la función ReLu, la cual suele aportar buenos resultados, aunque se puede hacer uso de cualquier otra.

Cabe destacar que los procesos anteriores estaban referidos a tratar imágenes de un solo canal, pero más allá de la puntualización que se hará a continuación del proceso de convolución, los demás conceptos son fácilmente aplicables a esta situación.
En la convolución para una imagen de más de un canal, el kernel utilizado debe tener la misma profundidad que esta, es decir, para una imagen a color con 3 canales (RGB), el kernel tendrá un tamaño de MxNx3. A cada una de los canales de la imagen se le aplicará su kernel correspondiente, y una vez calculadas las tres matrices resultantes, estas se sumarán para formar una única de salida con un solo canal. Esto se repetirá para tantos filtros como se indiquen obteniendo de nuevo el "feature mapping" correspondiente.

4.6.2 Capa de subsampling

Otro de los problemas que conlleva la aplicación de la capa de convolución, es que pese a que reduce las dimensiones de la matriz de salida, la aplicación de los filtros hace que incremente el número de neuronas con respecto a la capa anterior. Si tenemos un imagen de entrada de 32x32 píxeles para la primera capa, serán necesarias 1024 neuronas para llevar a cabo la operaciones pertinentes, sin embargo tras la primera convolución con un kernel de 3x3 y 20 filtros, las dimensiones de los datos de entrada para la siguiente capa serán de 30x30x20, y por lo tanto serán necesarias 18000 neuronas. Si se realizará otra capa de convolución tras esta, el número de neuronas aumentaría de forma desproporcionada, lo cual podría implicar un coste de procesamiento demasiado alto.
Para reducir este tamaño se aplicará un proceso llamado subsampling, donde a parte de disminuir las dimensiones de las imágenes filtradas, se deberán conservar las caracteristicas más importantes detectadas por cada filtro. Hay varios tipos de operaciones de subsampling pero el más utilizado es el max-pooling.

La aplicación del max-pooling es parecida a la opercación con el kernel, es decir, para ambos casos se recorrerá la matriz de izquierda a derecha y de arriba a abajo, operando con diferentes secciones de la matriz de entrada. En este caso sin embargo lo que se selecciona es el tamaño del marco que se aplicará, dentro del cual se seleccionará el elemento con el valor más alto. El avance de este sobre la entrada no se realizará de un píxel como en el caso de la convolución, sino que para cada elemento de la nueva matriz, se tomará una sección donde todos sus elementos no hayan sido evaluados en un caso anterior.
Para el caso de tener como entrada el conjunto calculado al inicio de la sección (30x30x20), al aplicar un max-pooling de 2x2 a cada una de las matrices de los 20 filtros, el resultado sería de 15x15x20. Como se ve

las dimensiones de la matriz se han reducido a la mitad, y las neuronas necesarias para la siguiente capa ahora serían 4500, que son muchas menos que en el caso anterior y en teoría siguen conservando la información necesaria para detección de caracteristicas. A continuación se muestra un ejemplo gráfico de la aplicación del max-pooling.

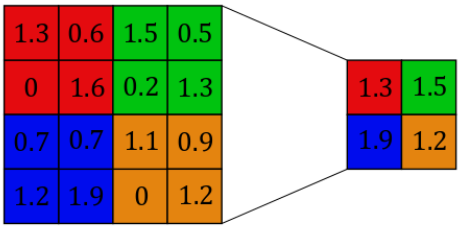

Figura 4.10 Ejemplo de max-pooling con un marco de 2x2.

4.6.3 Capa de clasificación

Una vez aplicadas todas las capas de convolución y subsampling deseadas, donde se han extraido las caracteristas de la imagen de entrada, lo único restante es el proceso de clasificación. En este punto los resultandos de la última capa hasta ahora tridimensionales (ancho, alto y número de mapas) se aplanan pasado a un vector, el cual será conectado una capa de neuronas tradicionales. Se irán aplicando capas de este tipo ajustando la cantidad de neuronas según sea necesario, hasta conseguir las dimensiones de salida requeridas. Durante este proceso según el tipo de resultado que se quiera obtener a la salida, será conveniente usar un tipo de función de activación u otra. El caso de querer obtener la clase a la que pertenece la imagen dentro un grupo concreto de ellas, estamos hablando de un tipo de red de clasificación. Este tipo es el más usual dentro de las redes neuronales convolucionales, debido a la labor de reconocimiento de objetos y patrones que desempeña. Para este caso la función de activación utilizada es la de Softmax, la cual modifica el vector de valores reales que recibe, a uno de iguales dimensiones pero en un rango de 0 a 1. Suponiendo que haya tres posibles clases, este recibirá una vector de tres componentes de números reales de la red, y el resultado que aportará el algoritmo será un vector igualmente de tres dimensiones pero con un valor entre 1 y 0. Este se puede interpretar como la probabilidad de que la imagen de entrada pertenezca a cada un de las clases, es decir, si se recibe [0.2, 0.1, 0.7] habrá un 20 %, un 10 % y un 70 % de probabilidades de que la imagen pertenezca a la primera, segunda o tercera clase respectivamente. El algoritmo responde a la siguiente formula:

$$\sigma(z(i)) = \left(\frac{e^{z(i)}}{\sum_{k=1}^{N} e^{z(k)}} \right) \tag{4.37}$$

En la ecuación, z corresponderá al vector al que se quiere aplicar la función de activación, calculándose de manera independiente para cada elemento i del mismo. N por otro lado será el tamaño del vector, y σ corresponderá a la función del algoritmo Softmax.
Como un ejemplo práctico si se deseara aplicar el algoritmo al vector [1, 2, 3, 1, 5] el resultado obtenido sería [0.015, 0.041, 0.111, 0.015, 0.818], donde se puede confirmar que la suma de todas las probabilidades del vector da 1.

Normalmente la gente asocia directamente las redes convolucionales a este tipo de red de clasificación, pero no es el único existente. Si lo que se desea obtener como salida es un conjunto de números reales, como la interpretación de la imagen de entrada, entonces no se trata de un problemas de clasificación sino uno de regresión. Este el caso por ejemplo de sistema que tenemos entre manos, el cual debe averiguar las velocidades angulares a las que se somete el sistema que recoge las imágenes analizándolas.
Para esta finalidad las funciones de activación que son utilizadas, son las que se usarían con un red neuronal clásica, por ejemplo la función sigmoide o la ReLu, debiendo estudiar la utilidad de cada una mediante

pruebas de entrenamiento.

5 Implementación del código

En este capitulo se desarrollará la explicación del código creado para el entrenamiento, dejando las estructuras de red utilizadas y sus resultados para más adelante.

5.1 Librerías

Las librerías necesarias para el correcto funcionamiento del código, además de las ya citas para el tratamiento de los datos y su representación, son sobre todo las relacionadas con Keras, y el tratamiento de las redes neuronales. A continuación se mostrará el conjunto completo de ellas, y posteriormente se realizará una breve descripción de su utilidad.

Código 5.1 Librerías.

```
import sys
import scipy.io
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Dropout, Flatten, Dense, Input, merge
from keras.models import Sequential, Model
```

- sys: Provee variables y funcionalidades directamente relacionadas con el interprete. Esta fue añadida en el caso de querer implementar la selección de la fuente de los experimentos mediante el interprete de comandos.
- scipy.io: Este modulo de SciPy permite el uso de funcionalidades para la lectura y escritura de datos del sistema. Esta es de especial importancia, ya que hace posible la correcta lectura de los datos de los experimentos para su posterior uso.
- numpy: Como ya se comento en un capitulo anterior, numpy añade herramientas para el uso y la manipulación de matrices y vectores. Su inclusión nos permite trabajar con las matrices de las imágenes, realizando los ajustes necesarios en cada momento.
- train_test_split: Este módulo de Sklearn permite separar un conjunto de datos en sets para entrenamiento y test. La separación en los dos grupos se hace según la proporción que se indique, y previamente a esta todos los datos son barajados de forma aleatoria, habiéndose podido demostrar durante las pruebas que esto último afectaba positivamente en el entrenamiento de la red.
- mean_squared_error: Permite realizar de forma sencilla el error cuadrático medio entre dos conjuntos de datos, lo cual será imprescindible para comprobar la fiabilidad de nuestra red una vez sea entrenada.

- Módulos de Keras: Cada uno de los módulos añadidos, permiten incluir los diferentes tipos de capas a los representan. Son usados para el diseño de la estructura, la compilación y el posterior entrenamiento del modelo.

5.2 Extracción y tratamiento de los experimentos

Los experimentos fueron entregados para su estudio en varios archivos *.mat*, por lo que era necesario extraer cada uno y posteriormente unirlos en un solo conjunto. Seguidamente se mostrará el código de este primer proceso, sobre el cual se puntualizarán algunos factores a tener en cuenta.

Código 5.2 Extracción de los experimentos.

```
#Número de experimentos
N_EXP = 14
#Ubicación en el equipo de los experiementos y coger el primero
D_EXP = 'C:/TFG/EXPERIMENTOS/cnn_ocelli_1.mat'

#Extraer el primer experimento
data_test = scipy.io.loadmat(D_EXP)
XL = data_test["imgLL"]
XR = data_test["imgRL"]
XF = data_test["imgFL"]
Y = data_test["target"]

#Extraer todos los experimentos restantes
for i in range(1, N_EXP):
  #Cambiar a siguiente experimento
  D_EXP = D_EXP.replace(str(i),str(i+1))

  #Extraer el siguiente experimento
  data_test= scipy.io.loadmat(D_EXP)

  XL_aux = data_test["imgLL"]
  XR_aux = data_test["imgRL"]
  XF_aux = data_test["imgFL"]
  Y_aux = data_test["target"]

  #Unir todos los experimentos en un solo conjunto
  XL = np.concatenate((XL, XL_aux))
  XR = np.concatenate((XR, XR_aux))
  XF = np.concatenate((XF, XF_aux))
  Y = np.concatenate((Y, Y_aux))
```

Una de las cosas que se tuvo que tener en cuenta, es que al estar los experimentos separados en diferentes archivos no era posible extraerlos de una vez. Esto provocó la necesidad de extraerlos de uno en uno, y añadir toda información a una misma variable.
En concreto, como se puede observar en el código, se toma el primer conjunto y dirección del mismo archivo como base, y posteriormente en un bucle *for* se van consiguiendo el resto. Aquí se va modificando el nombre del archivo a extraer, y una vez guardados los datos en la variable auxiliar, se añaden tras los ya extraídos.

Ya teniendo todos los experimentos disponibles, es necesario separarlos en los set que serán destinados para el entrenamiento y el test. Antes de esto, para que la mezcla de los datos sea uniforme, y ya pensando en la estructura necesaria para ser usados como datos de entrada de la red, se redimensionarán las matrices datos.

Código 5.3 Obtenición de los sets de entrenamiento y test.

```
#Redimensionar las muestras
(T1, ver1, M1, N1) = XL.shape

XL2 = XL.reshape([T1, 10, 8, 2])
XR2 = XR.reshape([T1, 10, 8, 2])
XF2 = XF.reshape([T1, 10, 8, 2])

XL2_aux = XL2.reshape([T1, 1, 10, 8, 2])
XR2_aux = XR2.reshape([T1, 1, 10, 8, 2])
XF2_aux = XF2.reshape([T1, 1, 10, 8, 2])

#Unir las tres entradas en un conjunto para realizar la mezcla
X = np.concatenate((XL2_aux, XF2_aux, XR2_aux), axis=1)

print('Dimensiones de la unión:')
print(X.shape)

#Mezclar y dividir los datos segun la proporcion train-test
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2)

print('Dimensiones de las entradas para entrenamiento:')
print(X_train.shape)
print('Dimensiones de las entradas para test:')
print(X_test.shape)
print('Dimensiones de las salidas para entrenamiento:')
print(Y_train.shape)
print('Dimensiones de las salidas para test:')
print(Y_test.shape)

#Recuperar las tres entradas
(T2, L2, M1, N1, CH2) = X_train.shape
(T3, L3, M3, N3, CH3) = X_test.shape

XL_train = X_train[0:T2, 0]
XF_train = X_train[0:T2, 1]
XR_train = X_train[0:T2, 2]

XL_test = X_test[0:T3, 0]
XF_test = X_test[0:T3, 1]
XR_test = X_test[0:T3, 2]

print('Dimensiones de los tres grupos para entrenamiento:')
print(XL_train.shape)
print(XF_train.shape)
print(XR_train.shape)

print('Dimensiones de los tres grupos para test:')
print(XL_test.shape)
print(XF_test.shape)
print(XR_test.shape)
```

Como se puede observar en el código, los tres grupos de imagenes se redimensionan dos veces, antes de la posterior unión de las mismas. En la primera se modifica la posición del indice, que diferencia entre los dos canales del grupo de imagenes de cada muestra, uno para cada instante temporal. Posteriormente se añade una dimensión, para asegurar un indice correcto sobre el que realizar la concatenación de datos en el

siguiente paso.
Una vez realizadas la modificaciones necesarias, se unen los tres grupos sobre el eje ya especificado anteriormente. Aquí hay que tener un cuenta el orden en el que se han añadido los grupos de imagenes, ya que identificar cada grupo es importante para el entrenamiento.
Con el grupo de datos listo, ya solo queda hacer uso de la función extraída en las librerías, introduciendo los conjuntos en el orden indicado por la documentación correspondiente. En primer lugar los datos que serán las entradas, en segundo las salidas, y como tercer argumento la proporción que se desea entre los dos sets.
Como salida del proceso se obtienen cuatro grupos de datos, los grupos de datos de entrada y salida para el entrenamiento, y sus homónimos para el test.
Ya habiéndose realizado este proceso con éxito, solo queda volver a separar los tres grupos de imágenes, prestando especial atención al orden como se ha indicado anteriormente, para así evitar clasificarlos de manera errónea.
Como ayuda visual, se mostrarán las respuestas del programa a los diferentes *print* distribuidos por el código, para así poder conseguir un mejor entendimiento de la evolución del proceso.

Código 5.4 Dimensiones de los conjuntos en el proceso.

```
Dimensiones de la unión:
(35356, 3, 10, 8, 2)
Dimensiones de las entradas para entrenamiento:
(28284, 3, 10, 8, 2)
Dimensiones de las entradas para test:
(7072, 3, 10, 8, 2)
Dimensiones de las salidas para entrenamiento:
(28284, 3)
Dimensiones de las salidas para test:
(7072, 3)
Dimensiones de los tres grupos para entrenamiento:
(28284, 10, 8, 2)
(28284, 10, 8, 2)
(28284, 10, 8, 2)
Dimensiones de los tres grupos para test:
(7072, 10, 8, 2)
(7072, 10, 8, 2)
(7072, 10, 8, 2)
```

5.3 Estructura de la red neuronal

Ya habiendo obtenido cada uno de los sets, los datos estas listos para el entrenamiento, pero antes de ese punto se ha de definir la estructura de la red, tras lo cual el modelo debe ser compilado.
La estructura definida a continuación con el código, se trata de uno de los últimos experimentos realizados para el proyecto. Sobre este se explicará el proceso de desarrollo de la estructura de un modelo, pudiéndose aplicar la lógica utilizada para otras configuraciones, eso sí, prestando atención a que este caso está destinado a un tipo de red de regresión, no de clasificación.

Código 5.5 Codificación de la estructura de un red.

```
Imagen_L = Input(shape=(10, 8, 2))
Imagen_F = Input(shape=(10, 8, 2))
Imagen_R = Input(shape=(10, 8, 2))

L = Conv2D(filters=50, kernel_size=(3, 3), padding='valid', activation="relu")(
    Imagen_L)
L = Dropout(0.4)(L)
```

```
L = Conv2D(filters=50, kernel_size=(3, 3), padding='valid', activation="relu")(
    L)

F = Conv2D(filters=50, kernel_size=(3, 3), padding='valid', activation="relu")(
    Imagen_F)
F = Dropout(0.4)(F)
F = Conv2D(filters=50, kernel_size=(3, 3), padding='valid', activation="relu")(
    F)

R = Conv2D(filters=50, kernel_size=(3, 3), padding='valid', activation="relu")(
    Imagen_R)
R = Dropout(0.4)(R)
R = Conv2D(filters=50, kernel_size=(3, 3), padding='valid', activation="relu")(
    R)

merged = merge([L, F, R], mode='concat', concat_axis=2)

Red = Conv2D(filters=20, kernel_size=(1, 1), padding='valid', activation="relu"
    )(merged)

Red = Flatten()(Red)
Red = Dense(units = 50, activation = 'relu')(Red)
Red = Dense(units = 50, activation = 'relu')(Red)
Red = Dense(units = 20, activation = 'relu')(Red)

Vel = Dense(units = 3, activation = 'linear')(Red)


Modelo = Model([Imagen_L, Imagen_F, Imagen_R], Vel)
Modelo.summary()
```

EL código de desarrollado utilizada la API proporcionada por Keras, la cual define las estructuras de forma diferente, al procedimiento de crear un modelo *sequential*.
Cada una de las ordenes, salvo las que definen las entradas que solo lo devuelven, reciben un tensor, el cual hace referencia a la capa anterior configurada de la de la red. Estos a su vez generan otro tensor con la capa añadida en la sentencia, de forma que se pueda aplicar a la siguiente capa que le siga.
Este modo de creación de capas, es realmente útil en el caso de que se quiera tratar, con redes cuya estructura tenga varias entradas y/o salidas, e incluso aplicando que todas las ramas generadas tengan diferentes tratamientos.
En el caso del proyecto actual, la estructura siempre cuenta con tres entradas, una por cada posición de las imágenes de los experimentos, y un vector de salida con tres datos, uno por cada velocidad angular en cada eje. Esta diferencia de dimensiones hace necesario aplicar una capa de unión (merge), en la que todas las ramas de las imágenes se unirán en una sola. Esta permite diferentes tipos de procedimientos a la hora de unir las capas, como la suma de los elementos a tratar, o la concatenación de los datos, pero de cualquier forma la diferencia entre cada uno, se verá mejor en el capitulo de los resultados obtenidos por cada estructura.

Comentado el uso de las funciones utilizadas, en primer lugar encontramos *Input*, *Conv2D* y *Dropout*, las cuales configuran una entrada de las red, aplican una capa de convolución, y aplican una capa de dropout, como se intuye.
La función *Input*, generará una entrada de las dimensiones indicadas en *shape* para la red, devolviendo el primer tensor que se utilizará.
Para *Conv2D* sus argumentos son más extensos, permitiendo confifurar el número de filtros que aplicará en la capa (*filters*), el tamaño del marco sobre el que se realizará la convolución (*kernel_size*), las opciones sobre la aplicación de *padding*, y la función de activación que se usará (*activation*).
La tercera función indicada (*Dropout*) aplicará una capa, que ignora un porcentaje de las neuronas de la capa anterior con respecto a la siguiente, seleccionándolas de forma aleatoria. Esta técnica permite que la red generalice mejor, y que sea menos probable que se adapte a los datos de entrenamiento. Para el uso de esta

función, unicamente hay que indicar el porcentaje de neuronas que se desean ignorar.
Con respecto a las ramas de cada imagen, cabe destacar que las tres tienen la misma estructura, tratándose todas por igual los datos. De ahí que las tres secciones donde reciben los tensores en las variables L, F y R, sean copias unas de las otras.

Para el último tramo de la estructura, tras aplicarse la capa final de convolución, es necesario aplicar la función *Flatten* como se observa en el código. Esta aplana la matriz de datos que la atraviesa, permitido el estudio de los datos resultantes por capas de conexión total. Estas se aplican mediante la orden *Dense*, indicando el número de neuronas y la función de activación cada caso. Para la última capa el número de neuronas a utilizar, será igual al tamaño de la salida que deseemos, en este caso tres datos.

Una vez configurada la arquitectura de la red, solo queda crear el modelo mediante *Model*, indicando los tensores de las entradas y las salidas.

Para finalizar este apartado, la última orden que aparece nos muestra un resumen de la red que acabamos de crear, indicado la evolución de las dimensiones de los datos. Seguidamente podremos observar la respuesta que ofrece esta función para la red indicada.

Código 5.6 Resumen ofrecido por la función *summary*.

```
Layer (type)                 Output Shape       Param #    Connected to
====================================================================================
input_1 (InputLayer)         (None, 10, 8, 2)   0
------------------------------------------------------------------------------------
input_2 (InputLayer)         (None, 10, 8, 2)   0
------------------------------------------------------------------------------------
input_3 (InputLayer)         (None, 10, 8, 2)   0
------------------------------------------------------------------------------------
conv2d_1 (Conv2D)            (None, 8, 6, 50)   950        input_1[0][0]
------------------------------------------------------------------------------------
conv2d_3 (Conv2D)            (None, 8, 6, 50)   950        input_2[0][0]
------------------------------------------------------------------------------------
conv2d_5 (Conv2D)            (None, 8, 6, 50)   950        input_3[0][0]
------------------------------------------------------------------------------------
dropout_1 (Dropout)          (None, 8, 6, 50)   0          conv2d_1[0][0]
------------------------------------------------------------------------------------
dropout_2 (Dropout)          (None, 8, 6, 50)   0          conv2d_3[0][0]
------------------------------------------------------------------------------------
dropout_3 (Dropout)          (None, 8, 6, 50)   0          conv2d_5[0][0]
------------------------------------------------------------------------------------
conv2d_2 (Conv2D)            (None, 6, 4, 50)   22550      dropout_1[0][0]
------------------------------------------------------------------------------------
conv2d_4 (Conv2D)            (None, 6, 4, 50)   22550      dropout_2[0][0]
------------------------------------------------------------------------------------
conv2d_6 (Conv2D)            (None, 6, 4, 50)   22550      dropout_3[0][0]
------------------------------------------------------------------------------------
merge_1 (Merge)              (None, 6, 12, 50)  0          conv2d_2[0][0]
conv2d_4[0][0]
conv2d_6[0][0]
------------------------------------------------------------------------------------
conv2d_7 (Conv2D)            (None, 6, 12, 20)  1020       merge_1[0][0]
------------------------------------------------------------------------------------
flatten_1 (Flatten)          (None, 1440)       0          conv2d_7[0][0]
------------------------------------------------------------------------------------
dense_1 (Dense)              (None, 50)         72050      flatten_1[0][0]
------------------------------------------------------------------------------------
```

```
dense_2 (Dense)              (None, 50)         2550       dense_1[0][0]
__________________________________________________________________________________
dense_3 (Dense)              (None, 20)         1020       dense_2[0][0]
__________________________________________________________________________________
dense_4 (Dense)              (None, 3)          63         dense_3[0][0]
==================================================================================
Total params: 147,203
Trainable params: 147,203
Non-trainable params: 0
```

5.4 Compilación y entrenamiento de la red

Tras la configuración de la estructura, el siguiente paso es la compilación del modelo. En este proceso se definirá el método de entrenamiento, y las medidas que se realizaran durante este. A continuación se expondrá la sentencia utilizada para este fin.

Código 5.7 Compilación del modelo.

```
Modelo.compile(optimizer='adam', loss='mean_squared_error', metrics=['mae'])
```

Los argumentos de la función implican en este caso, que se utilizará un optimizador de tipo *adam*, que la diferencia entre los valores predichos por la red y los reales se medirá en error cuadrático medio (*loss*), y además se aportará el error medio absoluto (*metrics*).
Si la compilación resulta correcta ya solo faltaría realizar el entrenamiento del modelo.

Para el entrenamiento serán necesarios, tanto el modelo ya compilado, como los set de datos que ya preparamos para el mismo fin. Seguidamente se puede observar la función utilizada.

Código 5.8 Entranamiento del modelo.

```
history = Modelo.fit([XL_train, XF_train, XR_train], Y_train, epochs=200,
    batch_size=128, verbose=1, validation_split=0.2)
```

Esta función recibe las entradas y salidas preparadas para el entrenamiento, el número de pasadas que realizará el algoritmo de entrenamiento sobre el set de datos (*epochs*), el número de segmentos en los se dividirá el set de datos para el proceso (*batch_size*), la opción de *verbose* activada para poder visualizar la barra de evolucion, y por último utilizar un porcentaje del set de entrenamiento como validación.
Cabe destacar un par de datos sobre los argumentos, siendo el primero que para el *batch_size* se recomienda utilizar un número de segmentos múltiplo de dos. El segundo por otra parte, es que la validación de datos se usa durante el entrenamiento, para comprobar la respuesta de la red ante un set diferente al que esta usando para entrenar, pudiendo ver así su respuesta, y dejando el verdadero set de test para la red completamente entrenada.

Para mostrar un ejemplo de los datos recibidos durante el proceso de entrenamiento, se podrán los datos del último paso.

Código 5.9 Muestra del entrenamiento.

```
Epoch 200/200
22627/22627 [==============================] - 2s 108us/step - loss: 0.0638 -
    mean_absolute_error: 0.1705 - val_loss: 0.0936 - val_mean_absolute_error:
    0.1903
```

Comenzando a analizar al información que contiene el segmento, se puede comprobar que efectivamente se trata del último paso de la de los 200 configurados. De las 28284 muestra que contenía el set de entrenamiento, solo se han usado 22627, coincidiendo la proporción restante con introducida para los datos de validación. Por

último, con respecto a las medidas, se puede ver que estas coinciden con las configuradas en la compilación, es decir, la perdida en error cuadrático medio, y una medida adicional del error medio absoluto. Estas son aplicadas tanto al set de entrenamiento como tal, como al de validación.
En los resultados obtenidos claramente se observa, que la red aplicada al set sobre el que entrena, obtiene un mejor resultado que para el de validación, lo cual es lo esperable.

5.5 Fiabilidad de la red

Llegados a este punto, la red esta lista para realizar la últimas pruebas para comprobar su efectividad al predecir los datos. Para ello, se usará al función correspondiente, con el fin de evaluar las entradas del set de test, y los resultados obtenidos se analizarán con los registrados para las salidas.

Código 5.10 Predicción y contraste de datos.

```
res = Modelo.predict([XL_test, XF_test, XR_test])
mse0 = mean_squared_error(Y_test[:, 0], res[:, 0])
mse1 = mean_squared_error(Y_test[:, 1], res[:, 1])
mse2 = mean_squared_error(Y_test[:, 2], res[:, 2])
```

La función *predict* unicamente toma el conjunto de entradas aptas (según lo configurado en la arquitectura) para la red, que se le pasen como argumento, y devuelve la predicción de las salidas.
Habiendo conseguido una predicción, solo resta comprobar cuan fiable es la red. Para ello se usará la función *mean_squared_error*, la cual en este caso calcula el error cuadrático medio entre los dos conjuntos, el de test y las predicciones, diferenciando entre los tres ejes para realizar los cálculos. Finalizando el seguimiento de la red que hemos comentado durante todo el capitulo se mostrarán los resultados a continuación.

Código 5.11 MSE para los datos de cada eje.

```
0.08331408660973724
0.08023791863328693
0.11105114056100494
```

Por último concluyendo es capítulo, para la evaluación de los resultados hay que tener en cuenta que siempre existirá cierta proporción de error, por lo que depende de la aplicación que le vayamos a dar a la red, podremos aceptar o no los resultados.

6 Resultados de las arquitecturas utilizadas

La finalidad de este capítulo, es mostrar los resultados de la fiabilidad de la red a través de las diferentes estructuras probadas. Se mostrará la configuración tomada en cada caso, y tras ello se analizarán los resultados incluyendo las gráficas pertinentes, mostrando en los casos más prometedores los errores cuadráticos medios.

6.1 Arquitectura del experimento 1

Tabla 6.1 Arquitectura del experimento 1.

Entradas	Imagen Izquierda	Imagen Frontal	Imagen derecha
1º Capa	Convolucional(3x3) 50 Filtros ReLu	Convolucional(3x3) 50 Filtros ReLu	Convolucional(3x3) 50 Filtros ReLu
2º Capa	Dropout(0.4)	Dropout(0.4)	Dropout(0.4)
3º Capa	Convolucional(3x3) 50 Filtros ReLu	Convolucional(3x3) 50 Filtros ReLu	Convolucional(3x3) 50 Filtros ReLu
4º Capa	Unión(Suma)		
5º Capa	Convolucional(1x1) 20 Filtros ReLu		
6º Capa	Flatten		
7º Capa	Dense(50) ReLu		
8º Capa	Dense(50) ReLu		
9º Capa	Dense(20) ReLu		
10º Capa/ Salida	Dense(3) SoftMax		

En esta arquitectura y varias de las que le siguen, hay que tener en cuenta un factor que influyó en los resultados. Durante los primeros experimentos, el set de datos de entrenamiento no se barajó, lo cual provocaba un problema de overfitting sobre la distribución de los datos. En esta y en las redes en las que se de, la medida de la perdida generada para el set de entrenamiento, variará bastante con respecto al de validación, obteniendo un error mayor para el segundo grupo.
En las primeras pruebas que se realizaron, más que conseguir el menor error posible, se quería comprobar los conceptos mostrados en el capitulo de teoría de la redes neuronales convolucionales. En este caso se prueba la función de activación “Softmax”, que como ya se vio aplicado a un problema de regresión tiene que dar un

error alto, ya que nunca otorgará una salida más alla del rango entre 0 y 1. A continuación podemos ver la evolución de ambas perdidas medidas en error cuadrático medio.

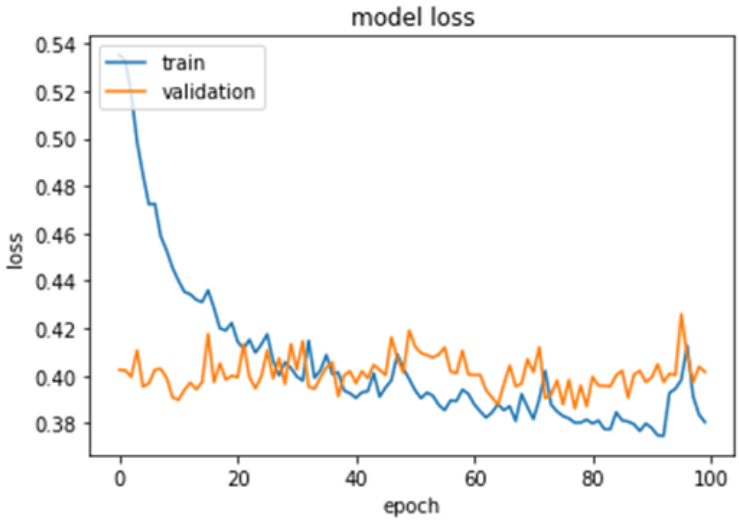

Figura 6.1 Evolución del MSE durante el entrenamiento 1.

Como era de esperar los errores recibidos son muy altos y poco estables, haciendo patente lo que ya se conocía, que es inútil usar la función "SoftMax" para el tipo de problema que tenemos entre manos.

6.2 Arquitectura del experimento 2

Tabla 6.2 Arquitectura del experimento 2.

Entradas	Imagen Izquierda	Imagen Frontal	Imagen derecha
1º Capa	Convolucional(3x3) 50 Filtros ReLu	Convolucional(3x3) 50 Filtros ReLu	Convolucional(3x3) 50 Filtros ReLu
2º Capa	Dropout(0.4)	Dropout(0.4)	Dropout(0.4)
3º Capa	Convolucional(3x3) 50 Filtros ReLu	Convolucional(3x3) 50 Filtros ReLu	Convolucional(3x3) 50 Filtros ReLu
4º Capa	Unión(Concatenación)		
5º Capa	Convolucional(1x1) 20 Filtros ReLu		
6º Capa	Flatten		
7º Capa	Dense(50) ReLu		
8º Capa	Dense(50) ReLu		
9º Capa	Dense(20) ReLu		
10º Capa/ Salida	Dense(3) Linear		

En este caso se quería ver, si realizar la unión por medio de la concatenación en vez de la suma aportaba una diferencia en su aplicación, de cara a comprobar cual de ella otorgaba mejores resultados. Como ya se había aplicado la suma al experimento anterior, se decidió utilizar la misma arquitectura, teniendo en mente que

además de esta forma como usamos “SoftMax”, pese a saber que no vamos a obtener un buen resultado, una mejora con respecto al caso anterior se haría evidente en las gráficas. Cabe destacar que la concatenación se realizó sobre el segundo eje de las matrices.

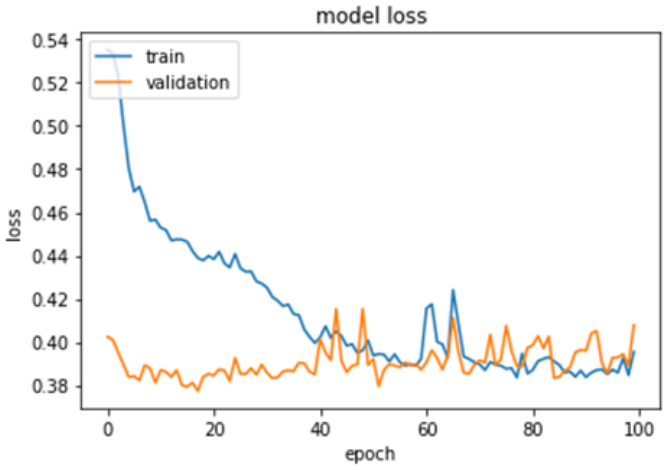

Figura 6.2 Evolución del MSE durante el entrenamiento 2.

A pesar de seguir siendo alto el error obtenido, se puede apreciar una menor diferencia entre ambos, además de haberse estabilizado hasta cierto punto. Estas pequeñas mejoras con respecto al experimento anterior, implicaban que tal vez la concatenación fuera una mejor opción que la suma, por lo que se decidió seguir usando en los siguientes casos, planeando volver a comprobar su diferencia con una red de mejores resultados.

6.3 Arquitectura del experimento 3

Tabla 6.3 Arquitectura del experimento 3.

Entradas	Imagen Izquierda	Imagen Frontal	Imagen derecha
1º Capa	Convolucional(3x3) 50 Filtros ReLu	Convolucional(3x3) 50 Filtros ReLu	Convolucional(3x3) 50 Filtros ReLu
2º Capa	Dropout(0.4)	Dropout(0.4)	Dropout(0.4)
3º Capa	Convolucional(3x3) 50 Filtros ReLu	Convolucional(3x3) 50 Filtros ReLu	Convolucional(3x3) 50 Filtros ReLu
4º Capa	Unión(Concatenación)		
5º Capa	Convolucional(1x1) 20 Filtros ReLu		
6º Capa	Flatten		
7º Capa	Dense(50) ReLu		
8º Capa	Dense(50) ReLu		
9º Capa	Dense(20) ReLu		
10º Capa/ Salida	Dense(3) Linear		

Para esta arquitectura se decido seguir los pasos realizados por el equipo de investigación, y se aplicó la función “Linear” a la salida de la red. También se mantuvo el método de unión en el modo de concatenación, y se modificó el dropout con respecto al que se describía en su articulo.

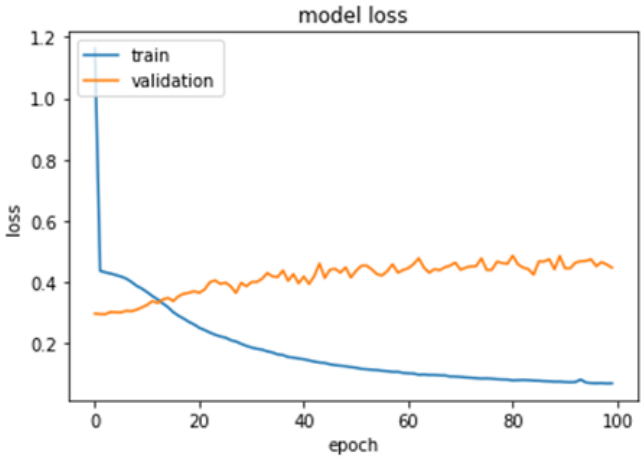

Figura 6.3 Evolución del MSE durante el entrenamiento 3.

Aunque aquí la red había a conseguido unos resultados muchos mejores que antes, la diferencia con el error en la validación era demasiado acusado. Esto claramente era debido al problema del overfitting antes citado, por lo que tras percatarse de ello investigando sobre este tipo de resultados, se halló la solución de barajar los datos antes de realizar la separación en sets.

6.4 Arquitectura del experimento 4

Tabla 6.4 Arquitectura del experimento 4.

Entradas	Imagen Izquierda	Imagen Frontal	Imagen derecha
1º Capa	Convolucional(3x3) 50 Filtros ReLu	Convolucional(3x3) 50 Filtros ReLu	Convolucional(3x3) 50 Filtros ReLu
2º Capa	Dropout(0.4)	Dropout(0.4)	Dropout(0.4)
3º Capa	Convolucional(3x3) 50 Filtros ReLu	Convolucional(3x3) 50 Filtros ReLu	Convolucional(3x3) 50 Filtros ReLu
4º Capa	Unión(Concatenación)		
5º Capa	Convolucional(1x1) 20 Filtros ReLu		
6º Capa	Flatten		
7º Capa	Dense(50) ReLu		
8º Capa	Dense(50) ReLu		
9º Capa	Dense(20) ReLu		
10º Capa/ Salida	Dense(3) Linear		

Para este experimento la arquitectura utilizada era igual a la anterior. Todo debido a que lo importante en este caso, era averiguar si el overfitting desaparecía, al haber mezclado los datos de los experimentos de forma aleatoria.

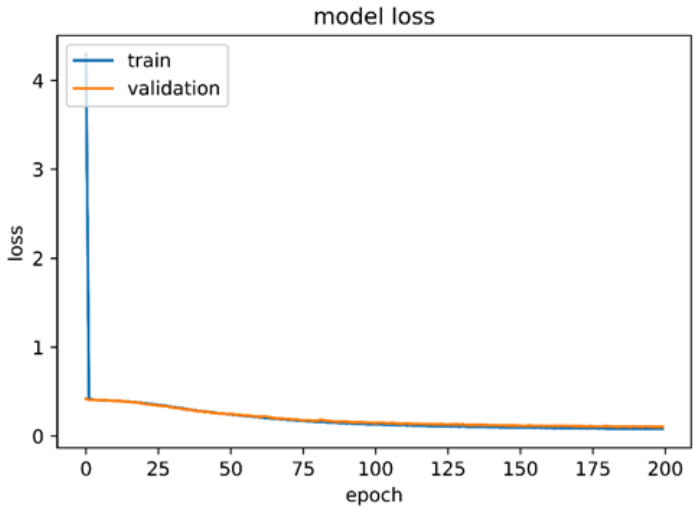

Figura 6.4 Evolución del MSE durante el entrenamiento 4.

El resultado obtenido deja probado que el overfitting ha desaparecido, comprobando que la integración de la mezcla de los datos era necesaria.
Por otra parte el error entra dentro del objetivo buscado, por lo que se calculó el error cuadrático medio de las predicciones sobre el set de test.

Código 6.1 MSE de las prediccines del experimento 4.

```
0.08331408660973724
0.08023791863328693
0.11105114056100494
```

También se mostrará a continuación una comparación de 60 muestras aleatorias, de la salida predicha por la red y la registrada para el set de test.

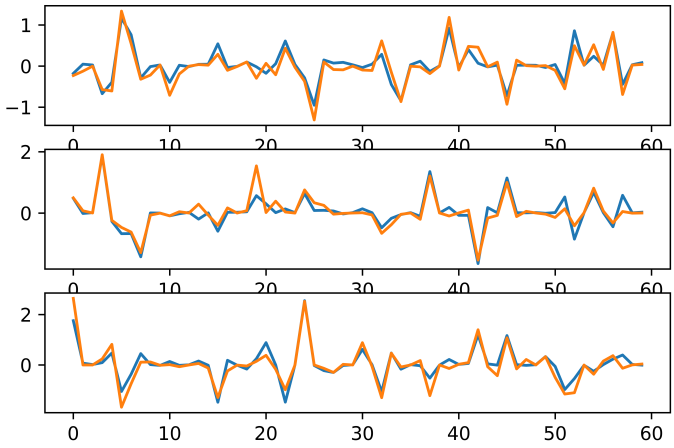

Figura 6.5 Salida predicha por la red vs. Salida del set de test.

Estos errores conseguidos se acercan bastante a los buscados, pero tal vez resulten ligeramente mayores a los que se desean conseguir, por lo que se intentará probar otra estructura para superarlo.

6.5 Arquitectura del experimento 5

Tabla 6.5 Arquitectura del experimento 5.

Entradas	Imagen Izquierda	Imagen Frontal	Imagen derecha
1º Capa	Convolucional(3x3) 50 Filtros ReLu	Convolucional(3x3) 50 Filtros ReLu	Convolucional(3x3) 50 Filtros ReLu
2º Capa	Dropout(0.4)	Dropout(0.4)	Dropout(0.4)
3º Capa	Convolucional(3x3) 50 Filtros ReLu	Convolucional(3x3) 50 Filtros ReLu	Convolucional(3x3) 50 Filtros ReLu
4º Capa	Unión(Suma)		
5º Capa	Convolucional(1x1) 20 Filtros ReLu		
6º Capa	Flatten		
7º Capa	Dense(50) ReLu		
8º Capa	Dense(50) ReLu		
9º Capa	Dense(20) ReLu		
10º Capa/ Salida	Dense(3) Linear		

Aprovechando que ya se disponía de una red que aportaba un error aceptable, se decidió volver a enfrentar el método de unión de las capa de suma y concatenación. Por lo tanto la única diferencia con la arquitectura anterior, será el uso de la suma.

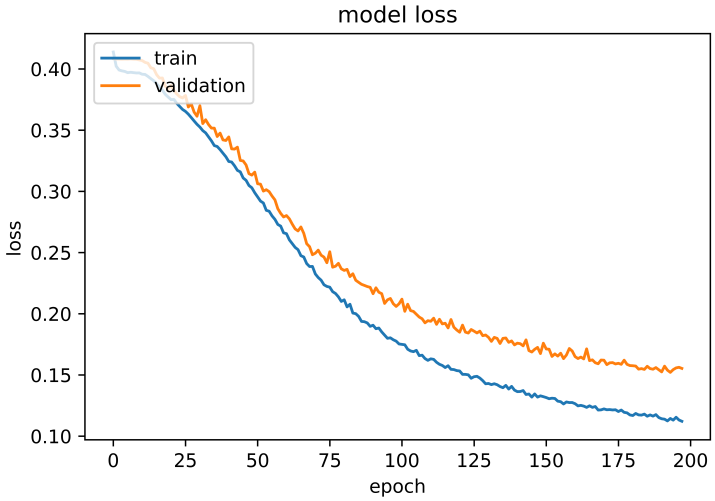

Figura 6.6 Evolución del MSE durante el entrenamiento 5.

Como se puede ver se consigue un buen resultado, pero peor que en el caso anterior. Esto nos confirma la deducción realizada anteriormente, siendo la concatenación la que aporta mejores resultados para la

distribución de tres entradas planteada.

6.6 Arquitectura del experimento 6

Tabla 6.6 Arquitectura del experimento 6.

Entradas	Imagen Izquierda	Imagen Frontal	Imagen derecha
1º Capa	Convolucional(2x2) 40 Filtros ReLu	Convolucional(2x2) 40 Filtros ReLu	Convolucional(2x2) 40 Filtros ReLu
2º Capa	Dropout(0.4)	Dropout(0.4)	Dropout(0.4)
3º Capa	MaxPooling2D(2x2)	MaxPooling2D(2x2)	MaxPooling2D(2x2)
4º Capa	Unión(Concatenación)		
5º Capa	Convolucional(2x2) 20 Filtros ReLu		
6º Capa	Flatten		
7º Capa	Dense(100) ReLu		
8º Capa	Dense(50) ReLu		
9º Capa	Dense(20) ReLu		
10º Capa/ Salida	Dense(3) Linear		

Para ver como reaccionaba la red a otro tipo de arquitectura, se hizo uso de otra de las comprobadas por el equipo de investigación, pero aplicándole la variaciones referentes a los avances conseguidos con las anteriores. Los resultados obtenidos fueron los siguientes:

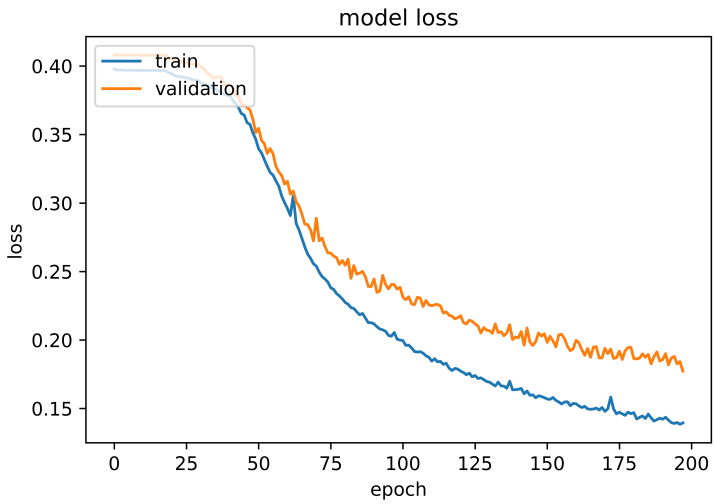

Figura 6.7 Evolución del MSE durante el entrenamiento 6.

Según los resultados la nueva arquitectura no consigue mejorar el mínimo de error conseguido hasta el momento, pero antes de sacar cualquier conclusión se comprobará la última distribución del equipo de investigación en el siguiente experimento.

6.7 Arquitectura del experimento 7

Tabla 6.7 Arquitectura del experimento 7.

Entradas	Imagen Izquierda	Imagen Frontal	Imagen derecha
1º Capa	Convolucional(2x2) 40 Filtros ReLu	Convolucional(2x2) 40 Filtros ReLu	Convolucional(2x2) 40 Filtros ReLu
2º Capa	Dropout(0.4)	Dropout(0.4)	Dropout(0.4)
3º Capa	MaxPooling2D(2x2)	MaxPooling2D(2x2)	MaxPooling2D(2x2)
4º Capa	Unión(Concatenación)		
5º Capa	Convolucional(2x2) 20 Filtros ReLu		
6º Capa	Flatten		
7º Capa	Dense(50) ReLu		
8º Capa	Dense(30) ReLu		
9º Capa	Dense(15) ReLu		
10º Capa/ Salida	Dense(3) Linear		

Como ya se ha comentado, este experimento toma como base la última de las tres arquitectura desarrolladas por el equipo de investigación.
La diferencia con la anterior radica en la sección de clasificación, es decir, la red de neuronas clásicas unida tras la última capa de convolución. En estas solo se modifican la cantidad de neuronas por capa, lo que ya de paso nos permitirá ver la reacción de la red ante este hecho.

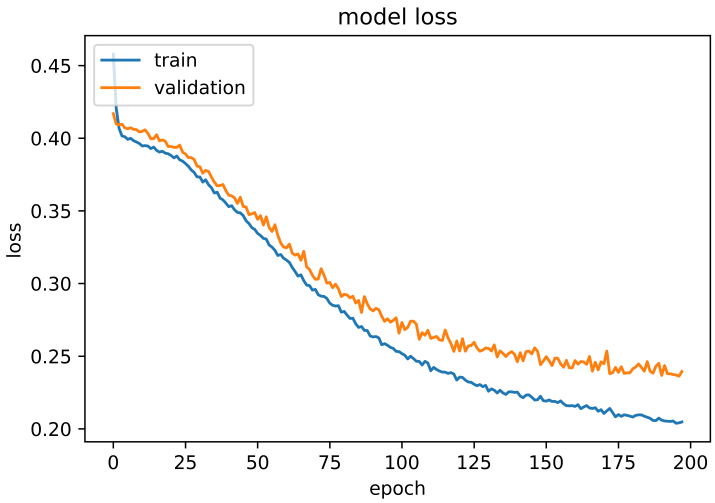

Figura 6.8 Evolución del MSE durante el entrenamiento 7.

El resultado es aun peor que los anteriores, dejando a la estructura del experimento 4 como la mejor. Aun que no se hayan conseguido resultados satisfactorios para este caso, al menos se ha demostrado la importancia de conseguir la cantidad adecuada de neuronas en las últimas capas, por lo que sería interesante para investigaciones posteriores, hallar como evoluciona el error de la estructura según modificamos su número, y así intentar identificar algún patrón.

7 Conclusiones

Tal como se ha mostrado en el desarrollo del proyecto, podemos confirmar que es posible obtener una red neuronal que responda de acuerdo a la funcionalidad de los ocelos. La cantidad de error que aportan los resultados puede no ser óptimo para una aplicación inmediata al control de un robot aéreo, pero deja patente que una arquitectura de no demasiadas capas es capaz de obtener una estimación correcta de las salidas. Por lo tanto es cuestión de realizar un desarrollo más complejo del trabajo realizado hasta ahora, el conseguir un sistema para ser aplicado.

El desarrollo en sí de este proyecto, basado en un sistema bioinspirado, fue pensado para poder conocer más de cerca las áreas del desarrollo científico basado en la naturaleza, y aprender de ellas.
Tanto la biónica como la biomimesis son ciencias que trabajan para conseguir una sociedad sostenible, aplicando en sus descubrimientos métodos que la naturaleza perfeccionó durante cientos de siglos de evolución, para la solución de diferentes problemas o necesidades existentes. El poder compartir con el lector el conocimiento recolectado sobre estas materias, espera conseguir un mayor interés acerca de estos por parte del público y así puedan comprobar por si mismos la variedad de proyectos prometedores con los que trabajan.

Por último para este proyecto, con respecto a las redes neuronales en si, se ha podido demostrar lo versátiles que son, y el poder trabajar en la resolución de un tipo de problema menos usual para las redes neuronales convolucionales como es el de regresión, ha llevado a comprender mejor como funcionan, y así poder proceder de manera más eficaz con ellas.

Índice de Figuras

Índice de Tablas

Índice de Códigos

Bibliografía

[1] Juan Ignacio Bagnato, *Aprende machine learning*, *https://www.aprendemachinelearning.com/*, 2019.

[2] Janine M. Benyus, *Biomímesis: Innovaciones inspiradas por la naturaleza*, 2002 ed., William Morrow, 2002.

[3] Stanford CS, *Convolutional neural networks for visual recognition.*, *http://cs231n.github.io/*, 2019.

[4] Deeplearning.vn, *Deep learning*, *https://deeplearning.vn/*, 2019.

[5] Github.com, *Keras*, *https://github.com/keras-team/keras/*, 2019.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep learning*, 1 ed., The MIT Press, 2017.

[7] Antonio Gulli, *Deep learning with keras*, 2017 ed., Pack>, 2017.

[8] G. Adrian Horridge, *El ojo compuesto de los insectos*, Revista Investigación y Ciencia (1977).

[9] Keras.io, *Keras*, *https://keras.io/*, 2019.

[10] M. Mérida-Floriano, F. Caballero, D. García-Morales, and L. Merino, *Bioinspired vision-only uav attitude rate estimation using machine learning*, Universidad Pablo Olavide (2017).

[11] Fernando Jordán Montés, *La visión de los insectos*, Revista Investigación y Ciencia (2013).

[12] Kevin M. Passino, *Biomimicry for optimization, control, and automation*, 2005 ed., Springer, 2005.

[13] Alejandro Trujillo Quevedo, *Sistema de reconocimiento de caracteres manuscritos usando redes neuronales convolucionales implementado en python*, Universidad de Sevilla (2017).

[14] StackExchange, *Artificial intelligence*, *https://ai.stackexchange.com/*, 2019.

[15] R. Volkel, M. Eisner, and K.J. Weible, *Miniaturized imaging systems*, Microelectronic Engineering (2003).

[16] Wikipedia.org, *Wikipedia*, *https://es.wikipedia.org/*, 2019.

[17] Martin Wilson, *The functional organisation of locust ocelli*, Journal of comparative physiology (1978).