



UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y DISEÑO INDUSTRIAL

Grado en Ingeniería Electrónica Industrial y Automática

TRABAJO FIN DE GRADO

Sistema de detección de meteoros basado en Redes Neuronales

Autor: Ibrahim Oulad Amar

Tutora:

Raquel Cedazo León

Departamento de Ingeniería
Eléctrica, Electrónica, Auto-
mática y Física Aplicada.

Madrid, Junio 2021

Abstract

The detection and location of meteors is very important, since through its study it is possible to understand the origin and evolution of the solar system. In addition, thanks to its observation it is possible to deviate its trajectory to avoid impacts with the Earth or other space assets. Currently this task is still performed manually due to the difficulty of implementing inexpensive automatic systems with low processing time and high performance. For all this, it has been decided to investigate the creation of an automatic meteor detection system that does not present these drawbacks.

This research began with the search for data. To do this, I got in touch with researcher Denis Vida from the University of Western Ontario. Who provided me access to the world's largest meteor database (Global Meteor Network) where data is collected from dozens of countries. In this case, a total of 89,263 images from Belgium, Germany, the Netherlands and the United Kingdom have been used. Once the images were obtained, they were organized for further processing.

After the preprocessing, the design of the system began, based on the choice of the architecture and the platform to be used. After a period of analysis it was decided to use a Convolutional Neural Network developed with the Python distribution of TensorFlow. The network training was carried out by varying the number of layers, the regularization, the optimizer and some hyperparameters such as the learning rate, the activation function or the weights initialization.

After training forty models, one was achieved with a 93.1% precision and an F1 score of 0.936. This model was implemented in a Raspberry Pi accomplishing a low processing time. For all this, the developed system outperforms several models of the state of the art, also eliminating the drawbacks of this kind of solutions.

Keywords: meteors, Machine Learning, Convolutional Neural Networks, Python, TensorFlow, Raspberry Pi.

Resumen

La detección y localización de meteoros es de gran importancia, ya que mediante su estudio se puede llegar a comprender el origen y la evolución del sistema solar. Además, gracias a su observación es posible desviar su trayectoria para evitar impactos con la Tierra u otros objetos espaciales. Actualmente, esta tarea se sigue realizando de forma manual, debido a la dificultad de implementar sistemas automáticos de bajo tiempo de procesamiento, alto rendimiento y coste reducido. Por todo esto, se ha decidido investigar sobre la creación de un sistema automático de detección de meteoros que no presente estos inconvenientes.

Esta investigación comenzó con la búsqueda de datos. Para ello, se contactó con el investigador Denis Vida, de la Universidad de Ontario Occidental. Quien proporcionó acceso a la mayor base de datos de meteoros del mundo (*Global Meteor Network*), donde se recogen datos de decenas de países. En este caso, se han utilizado un total de 89.263 imágenes pertenecientes a Bélgica, Alemania, Países Bajos y Reino Unido. Una vez obtenidas las imágenes se organizaron para su posterior procesado.

Después del procesado, comenzó el diseño del sistema, basado en la elección de la arquitectura y la plataforma a utilizar. Tras un periodo de análisis, se decidió utilizar una Red Neuronal Convolutacional desarrollada con la distribución en Python de TensorFlow. El entrenamiento de la red se llevó a cabo variando el número de capas, la regularización, el optimizador y algunos hiperparámetros como el ratio de aprendizaje, la función de activación o la inicialización de pesos.

Tras el entrenamiento de cuarenta modelos, se logró uno con un 93,1% de precisión y un valor F1 de 0,936. Dicho modelo se implementó en una Raspberry Pi logrando un bajo tiempo de procesamiento. Por todo esto, el sistema desarrollado supera el rendimiento de varios modelos del estado del arte, eliminando además los inconvenientes de este tipo de soluciones.

Palabras clave: meteoros, Aprendizaje Automático, Redes Neuronales Convolucionales, Python, TensorFlow, Raspberry Pi.

Índice general

Abstract	1
Resumen	2
Glosario	5
Lista de abreviaturas	6
Índice de tablas	6
Índice de figuras	7
Índice de ecuaciones	8
1 Introducción	11
1.1 Motivación	11
1.2 Objetivos	12
1.3 Estructura del Trabajo Fin de Grado	13
2 Estado del Arte	15
2.1 Transformada de Hough	15
2.2 Método alternativo basado en cuatro etapas	17
2.2.1 Filtrado de imagen	17
2.2.2 Generación de hipótesis	18
2.2.3 Verificación de hipótesis	18
2.2.4 Procesamiento final	18
2.3 Aprendizaje Automático para la detección de meteoros	19
2.3.1 Automatización del procesamiento de CAMS (2017)	19
2.3.2 Aprendizaje Profundo de transferencia (2018)	19
2.3.3 Redes Neuronales Recurrentes sobre vídeos (2019)	20
2.3.4 Redes Neuronales Convolucionales (2020)	21
2.4 Estrategias de computación en Machine Learning	22
2.4.1 Computación en la Nube	22
2.4.2 <i>Fog y Edge Computing</i>	22
2.4.3 Computación local	23
2.4.4 Aprendizaje Automático en el dispositivo final	23
2.5 Machine Learning en sistemas embebidos	23
2.5.1 Memoria	25
2.5.2 Tiempo de ejecución	25
2.5.3 Consumo energético	25
2.5.4 Seguridad y privacidad	25
2.5.5 Escalabilidad y flexibilidad	26

3 Fundamentos y herramientas	27
3.1 Machine Learning	27
3.1.1 Aprendizaje supervisado y no supervisado	27
3.1.2 Aprendizaje por lotes y en línea	29
3.1.3 Aprendizaje Basado en Instancias y Basado en Modelos	29
3.1.4 Principales desafíos del Machine Learning	29
3.1.5 Métricas de rendimiento	31
3.1.6 División de datos	32
3.2 Redes Neuronales Artificiales	33
3.2.1 Perceptrón	33
3.2.2 Perceptrón Multicapa y Retropropagación	35
3.2.3 Redes Neuronales Convolucionales	36
3.3 Tecnología CUDA de NVIDIA	40
3.3.1 Tecnología CUDA	40
3.3.2 CUDA con Redes Neuronales Convolucionales	43
3.4 TensorFlow	46
3.5 Raspberry Pi	48
3.6 Otro software empleado	49
4 Desarrollo	51
4.1 Datos para el entrenamiento	51
4.1.1 Adquisición de datos	51
4.1.2 Filtrado de datos	52
4.2 Entrenamiento	55
4.2.1 Equipo utilizado durante el entrenamiento	55
4.2.2 Datos de entrada	56
4.2.3 Cantidad y división de datos	56
4.2.4 Resolución de las imágenes	57
4.2.5 Optimizador y ratio de aprendizaje	57
4.2.6 Función de activación e inicialización de pesos	58
4.2.7 Regularización	60
4.2.8 Capas del modelo	61
4.3 Implementación en Raspberry Pi	62
5 Resultados	65
5.1 Resultados intermedios	65
5.2 Resultados finales	77
6 Conclusiones	79
6.1 Conclusiones	79
6.2 Desarrollos futuros	80
Bibliografía	81

Glosario

Lista de abreviaturas

TFG Trabajo Fin de Grado.

UPM Universidad Politécnica de Madrid.

CAMS *Cameras for All-Sky Meteor Surveillance.*

CNN Red Neuronal Convolucional (*Convolutional Neural Network*).

GPU Unidad de Procesamiento Gráfico (*Graphics Processing Unit*).

ML Aprendizaje Automático (*Machine Learning*).

IA Inteligencia Artificial.

RNN Red Neuronal Recurrente(*Recurrent Neural Network*).

RTOS Sistema Operativo de Tiempo Real (*Real Time Operating System*).

ANN Red Neuronal Artificial (*Artificial Neural Network*).

LTU Unidad de Umbral Lineal (*Linear Threshold Unit*).

MLP Perceptrón Multicapa (*Multilayer Perceptron*).

DNN Red Neuronal Profunda (*Deep Neural Network*).

CPU Unidad de Procesamiento Central (*Central Processing Unit*).

GPGPU GPU de Propósito General (*General Purpose GPU*).

CUDA *Compute Unified Device Architecture.*

SBC Ordenador de Placa Única (*Single Board Computer*).

IDE Entorno de Desarrollo Integrado (*Integrated development environment*).

FITS *Flexible Image Transport System.*

Índice de tablas

Tabla 2.1	Matriz de confusión del método alternativo de cuatro etapas.	18
Tabla 2.2	Resultados de la CNN desarrollada en [19].	22
Tabla 2.3	Comparación de hardware de microcontroladores y placas de desarrollo.	24
Tabla 3.1	Estructura de la matriz de confusión en un clasificador binario.	31
Tabla 5.1	Matriz de confusión del modelo final en el conjunto de validación.	77
Tabla 5.2	Matriz de confusión del modelo final en el conjunto de entrenamiento.	77
Tabla 5.3	Métricas de rendimiento del modelo final.	78

Índice de figuras

Figura 1.1	Publicaciones en revistas científicas relacionadas con IA [3].	12
Figura 1.2	Inversión total en IA [3].	12
Figura 2.1	Ejemplo de la Transformada de Hough.	17
Figura 2.2	Arquitectura de la CNN con mejor rendimiento en [19].	21
Figura 3.1	Comparación entre la programación tradicional y el Aprendizaje Automático.	28
Figura 3.2	Subajuste, ajuste óptimo y sobreajuste. Fuente: https://www.educative.io/	31
Figura 3.3	Múltiples capas en una Red Neuronal biológica. Fuente: https://en.wikipedia.org/wiki/Cerebral_cortex	33
Figura 3.4	Arquitectura LTU. Fuente: https://www.javatpoint.com/	34
Figura 3.5	Esquema de un Perceptrón [52].	35
Figura 3.6	Perceptrón Multicapa. Fuente: https://www.tutorialspoint.com/	36
Figura 3.7	Campo receptivo tipo filtro de Gabor de una célula simple. Fuente: https://towardsdatascience.com/	37
Figura 3.8	Modelos de células simples y complejas propuestos en [55]	37
Figura 3.9	Arquitectura de la CNN LeNet-5 [57]	38

Figura 3.10 Conexiones entre dos capas consecutivas en una CNN [51].	38
Figura 3.11 Conexiones entre dos capas consecutivas en una CNN con zan- da [51].	39
Figura 3.12 Ejemplo de un <i>kernel</i> de CUDA [61].	41
Figura 3.13 Suma de dos matrices con CUDA mediante manejo de hilos [61]. . .	41
Figura 3.14 Red de Bloques de Hilos en una GPU de NVIDIA [61].	42
Figura 3.15 Jerarquía de Memoria en CUDA [64].	43
Figura 3.16 Tiempo de ejecución (a) y comparación (b) de las implementacio- nes en [65].	44
Figura 3.17 Comparación del tiempo de ejecución de las implementaciones de CNN estudiadas en [66].	45
Figura 3.18 Memoria utilizada en las implementaciones de CNN estudiadas en [66].	46
Figura 3.19 Arquitectura de TensorFlow 2.0. Fuente: https://www.simplilearn.com	47
Figura 3.20 Raspberry Pi 4B. Fuente: https://www.raspberrypi.org	48
Figura 4.1 Acceso al servidor de datos de la Universidad de Ontario Occiden- tal.	52
Figura 4.2 Ejemplo de archivos presentes en un archivo FITS siguiendo el formato FTP.	53
Figura 4.3 Ejemplos de los datos utilizados.	54
Figura 4.4 Estructura y nomenclatura de los datos filtrados.	55
Figura 4.5 Aumento de datos con desplazamiento lateral excesivo.	57
Figura 4.6 Error en conjunto de entrenamiento frente a ratio de aprendizaje con distintos optimizadores.	58
Figura 4.7 Representación gráfica de la función sigmoide.	59
Figura 4.8 Representación gráfica de la función ReLu.	60
Figura 4.9 Representación gráfica de la función ELU.	60
Figura 4.10 Esquema del modelo final.	62
Figura 5.1 Resultados del modelo 1.	66
Figura 5.2 Resultados del modelo 2.	66
Figura 5.3 Resultados del modelo 3	67
Figura 5.4 Resultados del modelo 4.	68
Figura 5.5 Resultados del modelo 5.	68
Figura 5.6 Resultados del modelo 6.	69
Figura 5.7 Resultados del modelo 7.	70
Figura 5.8 Resultados del modelo 8.	70
Figura 5.9 Resultados del modelo 9.	71
Figura 5.10 Resultados del modelo 10.	72
Figura 5.11 Resultados del modelo 11.	72
Figura 5.12 Resultados del modelo 10 con distintas tasas de abandono.	73
Figura 5.13 Resultados del modelo 12.	74
Figura 5.14 Resultados del modelo 12 con distintas técnicas de regularización. .	75
Figura 5.15 Resultados del modelo 13.	76
Figura 5.16 Resultados del modelo 14.	76
Figura 5.17 Resultados del modelo final.	77

Índice de ecuaciones

Ecuación 2.1	Ecuación de la recta en forma polar.	15
Ecuación 3.1	Ecuación para el cálculo de la precisión.	32
Ecuación 3.2	Ecuación para el cálculo de la exhaustividad.	32
Ecuación 3.3	Ecuación para el cálculo del valor F1.	32
Ecuación 3.4	Ecuación de aprendizaje del Perceptrón.	35
Ecuación 3.5	Ecuación de la salida de una neurona en una capa de convolución.	39
Ecuación 4.1	Ecuación utilizada para cambiar el ratio de aprendizaje.	58
Ecuación 4.2	Ecuación de salida de la función de activación sigmoide.	59
Ecuación 4.3	Ecuación de la salida de la función de activación ReLu.	59
Ecuación 4.4	Ecuación de la salida de la función de activación ELU.	59

Capítulo 1

Introducción

1.1 Motivación

Los meteoroides son objetos espaciales de tamaño variable, desde granos de polvo hasta pequeños asteroides. Podemos pensar en ellos como rocas espaciales. Cuando estos meteoroides entran en la atmósfera de un planeta a gran velocidad, se queman, convirtiéndose en bolas de fuego o estrellas fugaces, denominadas meteoros. Los meteoros se forman en altitudes de entre 75 y 100 km, en la Mesosfera. Si un meteoro sobrevive a su paso por la Atmósfera, cae a la tierra convirtiéndose en un meteorito [1].

La detección de meteoros, así como la localización de meteoritos, es importante debido a dos razones principales. La primera es que los meteoros son restos del comienzo del sistema solar, de modo que mediante su examinación se puede estudiar el origen y la evolución del mismo. Por otro lado, los meteoritos de otros planetas ayudan a comprender los procesos que tienen lugar en su interior. Por ejemplo, se sabe que el centro de la tierra está compuesto por níquel y hierro gracias a los meteoritos.

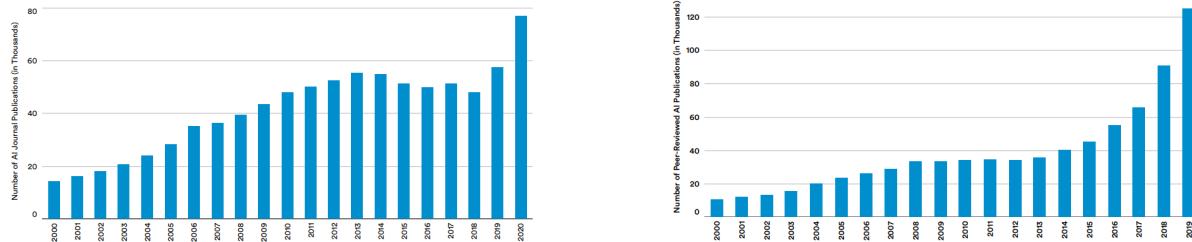
La segunda razón es la seguridad de la Tierra y los objetos espaciales, satélites y naves principalmente. Dado el peligro de impacto con la superficie terrestre o con objetos espaciales, los cuales al orbitar alrededor de la Tierra no están protegidos por la Atmósfera, es importante monitorizar las lluvias de meteoros para reducir esta amenaza [2].

Por otro lado, la Inteligencia Artificial (IA) es un campo que en los últimos años ha visto incrementada su relevancia, tanto en el apartado de investigación como en el económico como se puede observar en las Figuras 1.1 y 1.2 respectivamente. Esto se debe al gran potencial de la IA en tareas como la clasificación de imágenes, reconocimiento facial y visión artificial.

El gran auge de la IA mencionado anteriormente ha despertado mi interés y me ha llevado a investigar y aprender acerca de este campo. Tras realizar algunos cursos en línea y leer bibliografía, he decidido utilizar estas técnicas como base en mi Trabajo Fin de Grado (TFG) con el objetivo de adquirir más conocimientos, así como obtener experiencia en un proyecto real.

Finalmente, a lo anterior se suma un gran interés por la programación, así como por los sistemas embebidos y la electrónica surgido en el Grado. Por todo ello, un proyecto combinando estos tres pilares me resulta muy atractivo, pues es una oportunidad

de adquirir y aplicar conocimientos vanguardistas muy presentes en la industria a un problema real de cierta complejidad.



(a) Publicaciones totales en revistas científicas relacionadas con IA.

(b) Publicaciones revisadas por pares relacionadas con IA.

Figura 1.1: Publicaciones en revistas científicas relacionadas con IA [3].

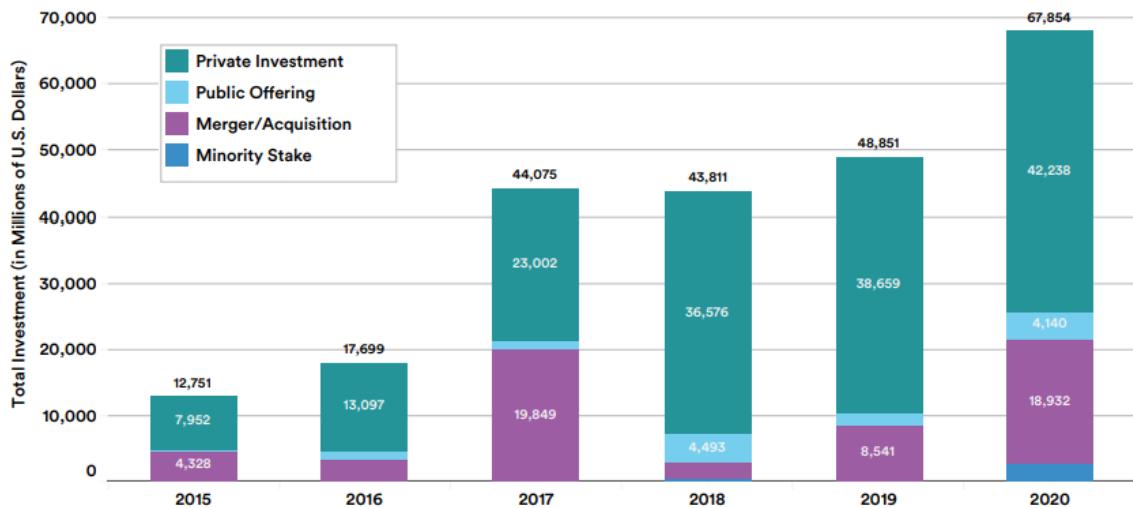


Figura 1.2: Inversión total en IA [3].

1.2 Objetivos

Los objetivos principales de este trabajo son, en concordancia con lo expuesto anteriormente, los siguientes:

1. Recolectar, organizar y filtrar los datos para el entrenamiento del algoritmo.
2. Diseñar y entrenar un algoritmo basado en Redes Neuronales para la detección de meteoros a partir de los datos adquiridos en el punto anterior.
3. Implementar el algoritmo desarrollado en un sistema embebido y realizar ensayos para comprobar la eficacia y el rendimiento de dicho algoritmo en un entorno real.

1.3 Estructura del Trabajo Fin de Grado

La estructura de este TFG queda definida de la siguiente forma:

- En este primer capítulo se describe la motivación, objetivos y estructura del trabajo.
- El segundo capítulo expone un estudio de la evolución de los métodos empleados para la detección automática de meteoros.
- El tercer capítulo recoge los fundamentos teóricos de este trabajo, así como las herramientas empleadas para el desarrollo del mismo.
- En el cuarto capítulo se detalla el desarrollo del sistema. Este capítulo se divide en tres secciones, centrándose cada una en uno de los objetivos de este TFG.
- En el quinto capítulo se presentan los resultados obtenidos a lo largo del desarrollo de este trabajo. También, se discuten los resultados finales obtenidos.
- En el sexto capítulo se exponen las conclusiones extraídas de este trabajo, así como posibles desarrollos futuros.
- Por último, se incluye la bibliografía empleada.

Capítulo 2

Estado del Arte

El método visual es el más sencillo para observar y detectar meteoros. En este caso, un observador estima si hay un meteoro, su magnitud y decide si pertenece o no a una lluvia de meteoros. Sin embargo, este método presenta algunas desventajas. La primera es la necesidad de tener observadores que revisen todas las imágenes obtenidas. La segunda es la dependencia de las condiciones meteorológicas, así como de la luminosidad en el momento en el que los meteoros atraviesan la Atmósfera.

Las comunicaciones de ráfaga de meteoros, en inglés *radio meteor scatter*, son otro método muy extendido para detectar meteoros. En este caso, es posible observar meteoros continuamente, sin verse afectado por las condiciones meteorológicas ni la luminosidad como en el método anterior. Está basado en el reflejo de ondas de radio desde la cola del meteoro. Cuando un meteoro aparece entre dos estaciones de radio, con una separación máxima de 2000 km, una puede recibir una pequeña porción de la emisión de la otra estación.

La observación mediante vídeo es el método más reciente y, a su vez, el más avanzado. Proporciona una precisión muy alta en todos los parámetros de un meteoro, tales como la posición, brillo y velocidad. *The Croatian Meteor Network* (CMN) [4] es una red de videovigilancia de bajo coste que es utilizada para monitorizar el cielo en Croacia y los países vecinos. La detección precisa de los recorridos de meteoros en imágenes es de gran importancia dado que afecta a la estimación de órbitas.

2.1 Transformada de Hough

La Transformada de Hough fue introducida en el año 1962 por Paul V.C. Hough en el artículo titulado “*A method and Means for Recognizing Complex Patterns*”. Esta herramienta es utilizada para detectar figuras que pueden ser expresadas matemáticamente, tales como rectas, círculos o elipses, en una imagen. En el caso de los meteoros, la Transformada de Hough se utiliza para detectar rectas. Por lo tanto, el objetivo es encontrar puntos alineados presentes en la imagen. Es decir, puntos que satisfagan la Ecuación 2.1.

$$\rho = x \cdot \cos \theta + y \cdot \sin \theta, \text{ con } \theta \in [0, \pi] \text{ y } \rho > 0 \quad (2.1)$$

De esta forma, el primer paso es realizar una transformación desde el plano imagen (x, y) al plano o espacio de parámetros (ρ, θ) , denominado espacio de Hough, para el conjunto de rectas de dos dimensiones. Para un punto cualquiera de la imagen (x_i, y_i) , las rectas que pasan por dicho punto son los pares (ρ, θ) que cumplen la Ecuación 2.1 donde ρ , la distancia entre la línea y el origen, está determinada por θ . Esto corresponde a una curva sinusoidal en el espacio (ρ, θ) que es única para ese punto.

Si las curvas correspondientes a dos puntos se interceptan, el punto de intersección en el espacio de Hough se corresponde a una recta en el espacio de la imagen que pasa por estos dos puntos. En consecuencia, un conjunto de puntos que forman una recta en la imagen se cortarán en los parámetros de dicha recta.

La Transformada de Hough hace uso de la matriz acumulador, cuya dimensión es igual al número de parámetros desconocidos del problema. En el caso de una recta será dos. Para formar esta matriz el primer paso es discretizar dichos parámetros. Esta discretización se realiza sobre los intervalos (ρ_{min}, ρ_{max}) y $(\theta_{min}, \theta_{max})$.

Cada celda del acumulador representa una figura cuyos parámetros se pueden obtener a partir de la posición de cada celda. Cada punto (x_i, y_i) en la imagen vota por las posibles rectas a las que puede pertenecer. Esto se realiza buscando todas las combinaciones posibles de (ρ, θ) . Si un punto cumple la ecuación de una recta, se busca la posición en el acumulador de la recta con dichos parámetros y se incrementa el valor de dicha posición [8, 9].

Finalmente, las rectas se detectan buscando las posiciones del acumulador con mayor valor, es decir, los máximos locales en el espacio del acumulador. El algoritmo se puede describir como sigue:

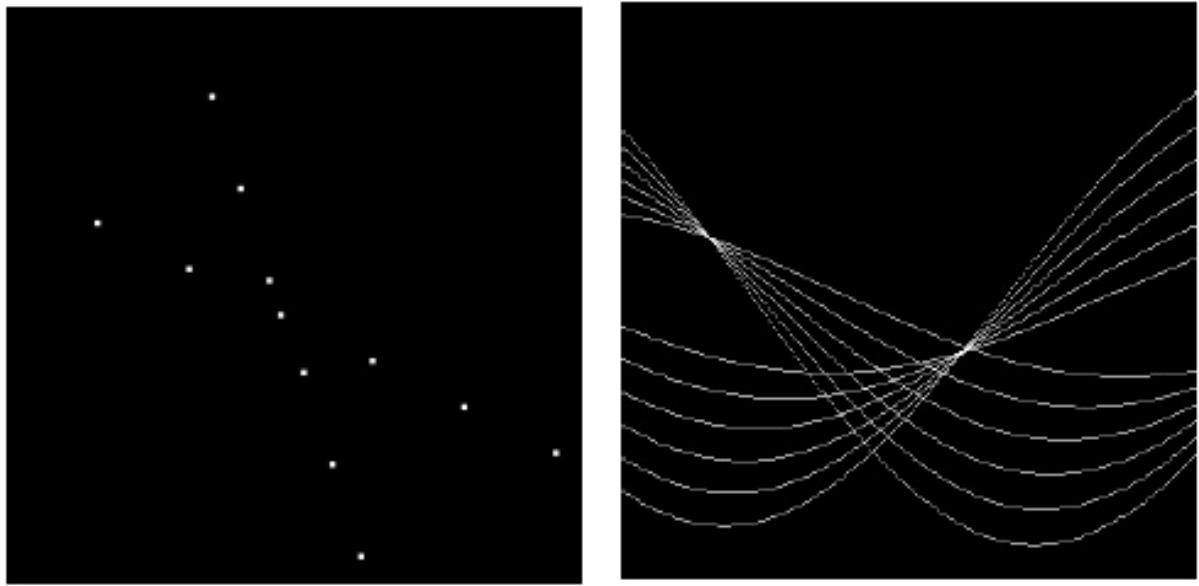
```

for all  $(\rho, \theta)$  do
     $votos_{(\rho, \theta)} \leftarrow 0$ 
for all  $(x, y)$  do
    for all  $\theta$  do
         $\rho \leftarrow x \cdot \cos \theta + y \cdot \sin \theta$ 
         $votos_{(\rho, \theta)} \leftarrow votos_{(\rho, \theta)} + 1$ 
return  $votos_{(\rho, \theta)}$ 
```

En la Figura 2.1 se puede observar un ejemplo de esta transformada. Por un lado, en la Figura 2.1(a) se representa la imagen original, que está formada por diversos puntos blancos sobre un fondo negro, formando dos líneas que se cortan. Por otro lado, en la Figura 2.1(b) tenemos la Transformada de Hough de la imagen anterior. En este caso se puede observar que hay dos puntos con un nivel de brillo muy superior al resto, lo cual indica que en la imagen existen dos rectas. A partir de las coordenadas de estos dos puntos se podría calcular la posición y orientación de las rectas en la imagen original.

Las principales ventajas de la Transformada de Hough son las siguientes:

- Robustez frente al ruido.
- Robustez ante la presencia de otras formas geométricas.
- Detección de múltiples líneas a la vez.



(a) Imagen original.

(b) Transformada de Hough de la imagen.

Figura 2.1: Ejemplo de la Transformada de Hough.

Sin embargo, esta técnica presenta algunas desventajas:

- Detección de falsos positivos.
- Coste computacional.
- Precisión de los parámetros.

Existen diversos algoritmos basados en imágenes para la detección de meteoros cuya idea básica es aplicar la transformada de Hough y localizar picos en el espacio de Hough, los cuales se corresponden con líneas en la imagen original [5, 6, 7]. Sin embargo, estos métodos también detectan otros objetos como aviones, pájaros y nubes, obteniendo falsos positivos. Debido a esto, es habitual la necesidad de un procesamiento posterior o la intervención de un humano para descartar estos falsos positivos.

2.2 Método alternativo basado en cuatro etapas

Existen numerosos algoritmos alternativos al uso de la Transformada de Hough para la detección de meteoros. Sin embargo, el más destacado es el propuesto en [2]. Donde se propone un novedoso método basado en cuatro etapas conectadas. Este método detecta si un meteorito está presente en la imagen y, en caso de ser así, proporciona la localización exacta, así como la dirección en la que viaja. Además, es capaz de detectar más de un meteorito en la misma imagen. Las cuatro etapas del método se explican a continuación.

2.2.1 Filtrado de imagen

La entrada a este primer módulo es una imagen en escala de grises. Esta imagen se procesa mediante un filtro espacial en la dirección horizontal para cada fila con el obje-

tivo de eliminar el fondo de la imagen, quedando así solamente los objetos brillantes. La imagen obtenida tras esta etapa, de la misma dimensión que la imagen de entrada, será la entrada del siguiente módulo.

2.2.2 Generación de hipótesis

En este módulo, se realiza un agrupamiento teniendo en cuenta solamente los elementos de la matriz imagen cuyo valor es igual a uno. Cada grupo o localización está definido por un centro y un radio. Estas localizaciones se denominan hipótesis, ya que son posibles meteoros. Este agrupamiento se realiza calculando la distancia euclídea entre dos puntos de la imagen, siendo del mismo grupo si esta es inferior a un umbral denominado *ddist*. Finalmente, los grupos con más de un cierto umbral, denominado *nclust*, de elementos, son la entrada del siguiente módulo.

2.2.3 Verificación de hipótesis

Cada hipótesis generada se verifica de la siguiente manera:

- Se aplica un filtro en la imagen original en la dirección vertical, y solamente en los elementos de las localizaciones calculadas anteriormente. Este filtro es análogo al utilizado en el primer módulo.
- Comparando los autovalores de los grupos [10].

La hipótesis es considerada un meteoro si el filtro produce más de nV unos y el autovalor más grande es, al menos, el doble que el más pequeño. La salida de este módulo son los grupos clasificados como meteoros.

2.2.4 Procesamiento final

Finalmente, los meteoros son procesados para estimar su dirección. Esto se realiza asignando una línea a cada grupo o meteoro. Si se detectan varios meteoros, se comparan las direcciones para descartar detecciones dobles, puesto que un meteoro puede ser parcialmente cubierto por nubes u otros cuerpos.

Los resultados de este algoritmo se muestran mediante la siguiente matriz de confusión:

	Etiquetados como meteoros	Etiquetados como no meteoros
Imágenes que contienen, al menos, un meteoro	95	5
Imágenes que no contienen ningún meteoro	15	85

Tabla 2.1: Matriz de confusión del método alternativo de cuatro etapas.

Como se puede observar en la Tabla 2.1, la exhaustividad es de un 95% (verdaderos positivos) y la precisión es de un 90%.

2.3 Aprendizaje Automático para la detección de meteoros

Durante los últimos años, el aprendizaje automático ha ganado una gran importancia en diversos campos, entre ellos, la detección de objetos en imágenes o vídeos. Esto ha sido así también en el campo de la detección de meteoros, donde ha habido grandes progresos que han permitido una automatización plena en muchos casos. A continuación, se realiza un repaso de las soluciones más importantes de los últimos años que emplean estas técnicas.

2.3.1 Automatización del procesamiento de CAMS (2017)

Las *Cameras for All-Sky Meteor Surveillance* (CAMS) son el método más utilizado para monitorizar el cielo en busca de posibles meteoros. Un enfoque interesante para automatizar el procesamiento de las imágenes obtenidas de las CAMS es el desarrollado en [11]. En este estudio se entrena un clasificador *Random Forest* (“bosque aleatorio”) en un conjunto de datos de unas 200 000 detecciones, obteniendo una precisión del 90% y una exhaustividad de un 81%.

Tras esto, se ha entrenado un clasificador basado en una Red Neuronal Convolutacional (CNN) en el mismo conjunto de datos. Según los investigadores, esta decisión fue motivada dado que las CNN son las que estaban consiguiendo el mayor rendimiento en tareas de visión artificial. La red utilizada en este estudio contiene cinco capas de convolución, seguidas de dos capas completamente conectadas y una última neurona binaria que actúa como salida. Además, para aumentar los datos de entrenamiento, se han utilizado técnicas de aumento de datos tales como la rotación y giro de imágenes. Con esta arquitectura, se ha logrado una precisión del 88.3% y una exhaustividad de un 90.3%.

Finalmente, se ha entrenado una red LSTM (*Long-Short Term Memory*) que utiliza datos secuenciales de un vídeo. Con esta arquitectura se han obtenido una precisión y exhaustividad del 90.0% y el 89.1%, respectivamente.

2.3.2 Aprendizaje Profundo de transferencia (2018)

En este trabajo [12] se utiliza una CNN pre-entrenada, que se ajusta para el problema de la detección de meteoros. Este tipo de prácticas se denominan Aprendizaje de Transferencia, y se utilizan para aprovechar redes complejas entrenadas en una gran cantidad y variedad de datos, ajustando las últimas capas y/o añadiendo capas nuevas para orientarlas a un problema concreto.

Los datos utilizados en este trabajo fueron proporcionados por *EXOSS Citizen Science* y la UNIVAP (*Universidade do Vale do Paraíba*). *EXOSS Citizen Science* es una organización sin ánimo de lucro que estudia y monitoriza meteoros en Brasil, disponiendo de más 50 estaciones activas [13]. En total, se han utilizado 1000 imágenes de meteoros y 660 de no meteoros para entrenar la red. Al entrenar una red pre-entrenada, la escasez de datos se mitiga.

El ratio de aprendizaje se ha ido modificando a lo largo del entrenamiento utilizando la técnica denominada *Stochastic Gradient Descent* con reinicios calientes desarrollada en [14]. Por otro lado, se ha utilizado ADAM [15] como algoritmo de optimización de

pesos en la red. Dado que los pesos de las capas de convolución en la red pre-entrenada son capaces de extraer características significativas, se han reutilizado para el caso de los meteoros. Mientras que los pesos de las últimas capas, las completamente conectadas, se han reemplazado con valores aleatorios para ser entrenadas en el conjunto de datos de meteoros. En este caso, hay dos capas completamente conectadas, con 1024 y 512 neuronas, respectivamente. La última capa es una neurona *softmax*. Tras entrenar las últimas capas y encontrar los pesos óptimos, se ha procedido a entrenar algunas capas convolucionales. Concretamente, se han mantenido las seis primeras capas dado que las primeras capas tienden a encontrar características generales.

Por último, el marco de trabajo utilizado (*framework*) es PyTorch [16], desarrollado por Facebook. Todos los entrenamientos se han realizado en un ordenador con una Unidad de Procesamiento Gráfico (GPU) NVIDIA Quadro P4000 de 8GB de memoria y 1792 núcleos. A continuación se discuten los resultados obtenidos.

En las capas completamente conectadas se ha utilizado un *Dropout* o capa de abandono de un 50% en la primera y un 25% en la segunda. Respecto al ratio de aprendizaje, se ha utilizado un valor de $2 \cdot 10^{-4}$. En cuanto a las técnicas de aumento de datos, cuando se han utilizado los resultados han empeorado. Esto indica que las transformaciones que se llevan a cabo en estas técnicas pueden ser incompatibles con este tipo de datos. Cabe destacar que una de las hipótesis mencionadas en este artículo es que las rotaciones y giros de las imágenes empeoran los resultados, esto se debe a que modifican la trayectoria del meteoro. En relación con la profundidad de la red, se ha observado que a partir de 50 capas la precisión no mejora.

Finalmente, se ha optado la red entrenada en los datos de Fashion-MNIST frente a la red entrenada en los datos ImageNet dado el mejor rendimiento obtenido para este problema.. Con todo esto, se ha logrado una precisión del 96,0% y un valor F1 de 0.94.

2.3.3 Redes Neuronales Recurrentes sobre vídeos (2019)

En el año 2019, Peter. S. Gural, un investigador muy destacado en el área de detección y monitorización de meteoros, propone en [17] un algoritmo basado en Redes Neuronales Recurrentes (RNN) para la detección de meteoros en vídeos.

En este campo normalmente se utilizan imágenes como entrada a los Algoritmos de Aprendizaje. Principalmente por el menor coste computacional y memoria que requiere procesar una imagen frente a un vídeo, en el entrenamiento y una vez desplegada la red. Sin embargo, en este estudio se propone añadir información temporal a los conjuntos de datos, esto es, utilizar vídeos en lugar de imágenes con el propósito de mejorar el rendimiento.

Los datos son vídeos con un máximo de 256 fotogramas. Se ha entrenado con aproximadamente 100 000 meteoros y 100 000 falsos meteoros. Por lo que se tiene un conjunto de datos balanceado y numeroso. En consecuencia, no es necesario utilizar técnicas de aumento de datos, que además podrían disminuir el rendimiento. Por último, estos datos se han dividido en tres conjuntos: un 85% para entrenamiento, 5% validación y el 10% restante para la evaluación final. Estos datos se han obtenido a través de CAMS desplegadas en California y Virginia.

En este estudio se entrena dos tipos de redes, una LSTM (*Long Short-Term Memory*)

y la otra biLSTM (*bidirectional LSTM*). Estas arquitecturas son un tipo de RNN utilizadas para procesar secuencias enteras de datos, con memoria a corto y largo plazo. Las redes propuestas se basan en el trabajo de Zoghbi et al [18] del equipo FDL (*Frontier Development Lab*) de la NASA. Para el desarrollo y entrenamiento se ha utilizado MATLAB, concretamente la *Deep Learning Toolbox*. Como máquina de entrenamiento se ha utilizado un ordenador portátil con un procesador Ryzen 5.

Finalmente, los resultados obtenidos han sido de un 98.1% de exhaustividad y un 2.1% de fugas (meteoro que han sido clasificados como falsas alarmas) para la red LSTM y de un 98.0% de exhaustividad y un 2.3% de fugas en el caso de la red biLSTM.

2.3.4 Redes Neuronales Convolucionales (2020)

El trabajo más avanzado en este campo es el desarrollado por D. Cecil y M. Campbell-Brown [19] del *Canadian Automated Meteor Observatory*. Esta investigación desarrolla un método para aplicar CNNs a la detección de meteoro formado por tres principales bloques:

- Filtrado inicial: en este bloque se realiza un primer filtrado para encontrar regiones de interés en una imagen. Concretamente, se trata de un algoritmo desarrollado en Python que detecta regiones del fotograma donde el nivel de brillo ha aumentado respecto al fotograma anterior. A estas regiones de interés se les aplica la Transformada de Hough para detectar líneas. Si se detectan líneas, entonces estas regiones de interés se escalan a una resolución de 128x128 y se convierten en la entrada del siguiente bloque.
- Filtrado mediante CNN: en este caso, se aplica una CNN a la imagen generada en el bloque anterior de tamaño 128x128. La red con los mejores resultados se muestra en la Figura 2.2 y como se puede observar está formada por cuatro etapas de convolución y agrupación seguidas de dos capas completamente conectadas, finalizando en la neurona final en la cual se obtiene el resultado de la clasificación.
- Extracción de eventos: en este último bloque se utilizan las detecciones calculadas en los bloques anteriores para separar los meteoro espacial y temporalmente. Además, se calculan trayectorias y velocidades.

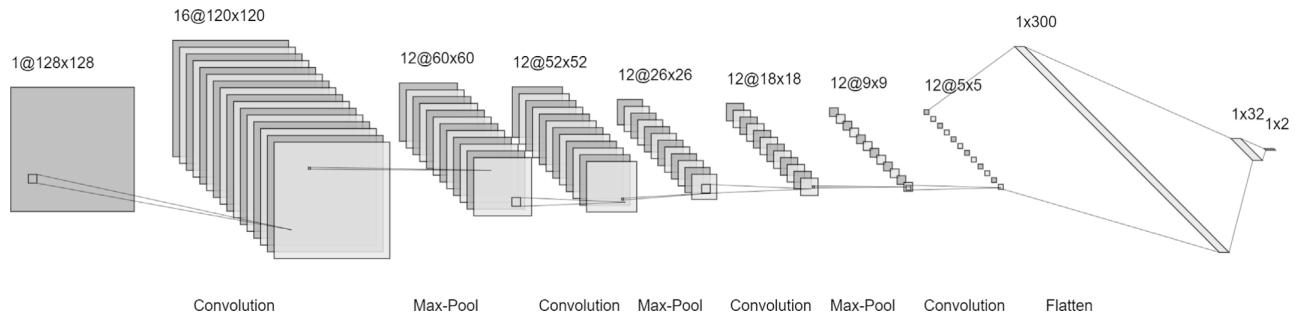


Figura 2.2: Arquitectura de la CNN con mejor rendimiento en [19].

Los datos utilizados son 56 230 imágenes en total, 22 444 etiquetadas como meteoro y 33 786 etiquetadas como no meteoro. El entrenamiento se ha realizado utilizando una

aceleración mediante tarjeta gráfica, en concreto la NVIDIA GTX 1080, que dispone de 8GB de memoria. En este caso el marco de trabajo utilizado es la implementación de Keras [20] dentro TensorFlow [21]. Este sistema tiene el mejor rendimiento hasta la fecha, con unos resultados excelentes como se muestra en la Tabla 2.2.

	Precisión	valor F1	Error
Conjunto de entrenamiento	99, 85%	0, 998	0, 009
Conjunto de validación	99, 78%	0, 997	0, 011
Conjunto de prueba	99, 79%	0, 997	0, 007

Tabla 2.2: Resultados de la CNN desarrollada en [19].

2.4 Estrategias de computación en Machine Learning

Mientras que el Aprendizaje Automático es una tarea intensiva a nivel computacional, los sistemas embebidos carecen de los recursos presentes en ordenadores de escritorio o servidores. Por ello, en este tipo de sistemas ciertas concesiones son necesarias entre consumo, recursos de memoria y rendimiento [22]. El desarrollo de técnicas con este enfoque es uno de los mayores desafíos del campo del Aprendizaje Automático, pues permitiría una migración de los sistemas desarrollados en el entorno científico, con recursos de computación casi ilimitados, a aplicaciones reales donde los recursos escasean y es preciso abaratar los costes.

2.4.1 Computación en la Nube

En los últimos años la computación en la nube ha permitido superar las barreras de la escalabilidad y flexibilidad [24, 23], proporcionando recursos ilimitados para entrenar y aplicar modelos de Aprendizaje Automático. Sin embargo, existen algunos inconvenientes al utilizar la Nube en sistemas de IA. El más destacado es que un sistema dependiente de la computación en la nube lo es también respecto a la conexión a internet para la transferencia de datos [25]. Una vez se interrumpe o retrasa la conexión a internet el sistema no funciona correctamente.

Por otra parte, la privacidad y seguridad se pueden ver comprometidos dado que los datos han de transmitirse al servidor [26, 27]. Por último, en aplicaciones críticas o de tiempo real, este enfoque puede no ser adecuado dada la latencia para transmitir y recibir datos a la nube.

2.4.2 Fog y Edge Computing

Los *Edge-devices* (dispositivos perimetrales) normalmente proporcionan el punto de entrada a la red. De modo que son responsables de recibir los datos de los dispositivos

finales, los desplegados en el campo, y enviarlos a la Nube. Sin embargo, también suelen almacenar datos de forma temporal, realizar un preprocesamiento de los mismos y garantizar la seguridad de la red. Estos dispositivos suelen ser pequeños ordenadores, que pueden ser programados en lenguajes de alto nivel como Python. De esta manera, exportar un modelo desde la Nube a estos dispositivos es un proceso sencillo.

Para reducir la latencia, el modelo se divide en una primera etapa que se realiza a nivel local y una segunda etapa que se realiza en la Nube [30]. El *fog-computing* [29] va un paso más allá y transfiere una parte del procesamiento al dispositivo final dividiendo el modelo en tres etapas. Debido a esto, la seguridad se ve reforzada dado que los datos no tienen que abandonar la red local.

El *fog-computing* también ayuda a bajar los costes, mejorar la eficiencia y facilitar el mantenimiento de los sistemas en las fábricas frente al uso exclusivo de la Nube para procesar los datos [31].

2.4.3 Computación local

En la mayoría de aplicaciones, debajo de la arquitectura de redes existen dispositivos finales que son los encargados de recolectar los datos, disponiendo de sensores para ello. Estos dispositivos se comunican con la red para enviar datos y recibir órdenes.

Normalmente estos dispositivos son alimentados mediante baterías dado que se encuentran en localizaciones remotas. Por lo tanto, un bajo consumo es crítico, así como la posibilidad de disponer de modos de suspensión y/o hibernación para situaciones donde no se requiera ninguna operación. Por otra parte, estos instrumentos deben ser capaces de ejecutar aplicaciones en tiempo real. En consecuencia, es necesario un Sistema Operativo de Tiempo Real (RTOS) [28].

2.4.4 Aprendizaje Automático en el dispositivo final

Dadas todas las ventajas de los métodos anteriores, parece innecesario procesar los datos de forma local en el dispositivo final. Sin embargo, esto se convierte en algo fundamental en algunos escenarios, especialmente cuando se trata de aplicaciones en tiempo real. Además, para comunicar los dispositivos finales con la red es necesario un canal de comunicaciones inalámbrico en el dispositivo final. No obstante, hay estudios que indican que hasta el 70% de la energía consumida en este tipo de dispositivos es debida a las comunicaciones [32, 33].

2.5 Machine Learning en sistemas embebidos

Diversos trabajos hablan sobre el uso del Machine Learning en sistemas embebidos [34, 35, 36, 37, 38]. En estos trabajos se utilizan principalmente dispositivos como Raspberry Pi o BeagleBone con posibilidad de ejecutar lenguajes de alto nivel como Python y, en consecuencia, se pueden exportar modelos entrenados en computadores de alto rendimiento a estos dispositivos de manera muy sencilla. En estos casos, mediante optimizaciones y uso de librerías específicas como TensorFlow Lite, una optimización de TensorFlow en C++ para dispositivos móviles y algunos sistemas embebidos, se consigue un rendimiento similar al de un equipo de escritorio moderno.

Por otra parte, algunos fabricantes de microcontroladores han lanzado al mercado dispositivos optimizados para tareas de Aprendizaje Automático. En 2017 Microchip ha integrado una GPU 2D en la familia de microcontroladores PIC32MZ DA [39]. Dado que las GPU son capaces de realizar cálculos paralelos, esto ha mejorado considerablemente el rendimiento de estos microcontroladores en tareas intensas de computación. En 2019 ARM ha anunciado la tecnología Helium, que va a estar presente en la próxima generación ARMv8.1 de sus microcontroladores. Esta tecnología puede llegar a lograr un rendimiento 15 veces mayor en aplicaciones de Aprendizaje Automático [40].

Diversos autores han integrado algoritmos de Aprendizaje Automático en microcontroladores con recursos muy limitados [41, 42, 43, 44, 45]. En la mayoría de estos casos se han logrado rendimientos aceptables con alrededor de un 70% en datos y algoritmos sencillos. La gran ventaja de estos modelos es la poca utilización de la RAM/SRAM, a lo sumo decenas de kB, así como su eficiencia energética. Sin embargo, para aplicaciones más sofisticadas en tiempo real este tipo de dispositivos no son capaces de ofrecer un rendimiento satisfactorio. Esto resulta lógico al comprar las especificaciones de estos dispositivos con otros sistemas embebidos como se muestra en la Tabla 2.3.

Dispositivo	Frecuencia de reloj	Flash	RAM
Arduino Uno (ATMega328P)	16 MHz	32 kB	2 kB
Arduino Mega (ATMega2560)	16 MHz	256 kB	8 kB
STM32L0 (Cortex-M0)	32 MHz	192 kB	20 kB
STM32F2 (Cortex-M3)	120 MHz	1 MB	128 kB
STM32F4 (Cortex-M4)	180 MHz	2 MB	384 kB
BeagleBone Black (Cortex-A8)	1 GHz	4 GB	512 MB DDR3
Raspberry Pi 4 (Quad core Cortex-A72)	1.5 GHz	MicroSD	2/4/8 GB LPDDR4

Tabla 2.3: Comparación de hardware de microcontroladores y placas de desarrollo.

Tras analizar los proyectos citados anteriormente, se observa que existen algunos desafíos recurrentes a la hora de implementar algoritmos de Aprendizaje Automático en sistemas embebidos. Para finalizar este capítulo, se presentan a continuación los más importantes.

2.5.1 Memoria

La capacidad reducida de las memorias es uno de los mayores desafíos a la hora de implementar algoritmos de Aprendizaje Automático en sistemas embebidos. Esto se debe a que la mayoría de estos dispositivos disponen de memorias del orden de los kB. Aunque estas capacidades se están aumentando en los últimos años, siguen sin ser óptimas para este tipo de sistemas.

Es importante diferenciar los distintos tipos de memoria presentes en este tipo de dispositivos. Existen dos grandes grupos, las memorias volátiles (RAM: SRAM y DRAM principalmente) y las no volátiles (especialmente Flash y EEPROM). La memoria RAM es la que ofrece un mayor rendimiento, pues la velocidad de lectura/escritura es muy alta. Sin embargo, este tipo de memoria tiene un alto coste, por ello se utiliza en tamaños pequeños. Las memorias no volátiles tienen la ventaja de ser mucho más económicas, además de que al ser no volátiles no se pierden los datos tras un apagado o reinicio. En cambio, la velocidad de lectura/escritura es mucho más baja.

2.5.2 Tiempo de ejecución

El tiempo de ejecución se mide en gran medida por la frecuencia de reloj del procesador, así como el número de procesadores. La mayoría de los microcontroladores tienen una frecuencia de reloj del orden de decenas de MHz y tienen únicamente un procesador. Esto dificulta en la mayoría de los casos lograr una ejecución en tiempo real, pues solo se dispone de un hilo de ejecución con una velocidad que, especialmente cuando se trata de Redes Neuronales densas, no es suficiente. Por ello, algunos dispositivos como Raspberry Pi 4, el elegido para este proyecto, son más adecuados dado que cuentan con cuatro núcleos de alto rendimiento, concretamente ARM Cortex-A72, que alcanzan los 1.5 GHz de frecuencia de reloj.

2.5.3 Consumo energético

El consumo energético es un elemento clave en este tipo de dispositivos, al estar en muchas ocasiones en localizaciones remotas o de difícil acceso. La comunicación inalámbrica puede suponer hasta el 70% del consumo en estos dispositivos [32, 33]. Implementar los algoritmos de forma local ayuda a mitigar este problema, dado que las comunicaciones se reducen de manera sustancial. Esto se debe a que en estos casos, solo se envía una porción de los datos capturados, los que resultan importantes tras su paso por el algoritmo. Incluso en algunos sistemas se puede prescindir del envío de los datos adquiridos, enviándose solamente los resultados derivados de estos.

No obstante, otros aspectos como un código optimizado y una buena gestión de memoria son también necesarios para mitigar este problema. En el caso de la memoria, dependiendo del tipo puede consumir más o menos energía. Además, al aumentar el tamaño de esta lo hace también el consumo.

2.5.4 Seguridad y privacidad

La seguridad y privacidad son elementos importantes en este tipo de sistemas dado que pueden estar monitorizando lugares públicos, casas o incluso seres humanos, como

es el caso de los relojes inteligentes, pudiendo adquirir datos personales. Por ello, aunque el procesamiento de los datos se realice de forma local, es necesario cifrar los resultados de este procesamiento antes de enviarlos a la Nube.

2.5.5 Escalabilidad y flexibilidad

Por último, a la hora de diseñar algoritmos con la intención de implementarlos en sistemas embebidos, es preciso tener en cuenta que para entrenar estos algoritmos se haga con un sistema y unos datos compatibles con los dispositivos concretos que se van a utilizar. Es frecuente desarrollar algoritmos con datos que no concuerdan con los que se van a adquirir en el campo. Además, es recomendable desarrollar algoritmos que se pueden implementar en sistemas distintos, ganando con ello flexibilidad.

Otro aspecto muy importante y, bajo mi punto de vista, crucial, es la posibilidad de actualizar el sistema de forma remota. Pudiendo así mejorar posibles errores e introducir mejoras de manera sencilla, lo cuál favorece la escalabilidad y flexibilidad del sistema.

Capítulo 3

Fundamentos y herramientas

En este capítulo se desarrollan los fundamentos teóricos en los cuales se basa este Trabajo Fin de Grado. Así como una explicación de las herramientas y tecnologías empleadas para llevarlo a cabo.

3.1 Machine Learning

El Machine Learning (ML) es una tecnología en auge en los últimos años. Una definición general de esta ciencia es la siguiente:

“El Aprendizaje Automático es el campo de estudio que otorga a los ordenadores la habilidad de aprender sin ser explícitamente programados.” Arthur Samuel, 1959.

Otra más específica:

“Se dice que un programa de ordenador aprende de la experiencia E, respecto a una tarea T y con un rendimiento P, si el rendimiento sobre la tarea T, medido por P, mejora con la experiencia E.” Tom Mitchell, 1997.

Como se indica en la Figura 3.1 este enfoque es el opuesto al de la programación tradicional, donde el sistema tiene como entradas datos y un conjunto de reglas, y al aplicar dichas reglas a los datos se obtienen respuestas. Por contra, en el Machine Learning el sistema tiene como entradas los datos y las respuestas, y, en este caso, se busca obtener las reglas que producen dichas respuestas a partir de los datos.

Esta ciencia se aplica a problemas donde las reglas no están claras o donde estando claras, son tan complejas que programarlas deja de ser práctico y cuyo mantenimiento es muy complicado. Un ejemplo de esto es un filtro de correos no deseados, donde la casuística es prácticamente ilimitada y cambia cada día. A continuación, se hace un repaso de los principales tipos de ML.

3.1.1 Aprendizaje supervisado y no supervisado

Los sistemas de Machine Learning pueden ser clasificados en cuatro categorías principales según el tipo de supervisión durante el entrenamiento: aprendizaje supervisado, aprendizaje no supervisado, aprendizaje semi-supervisado y aprendizaje por refuerzo.

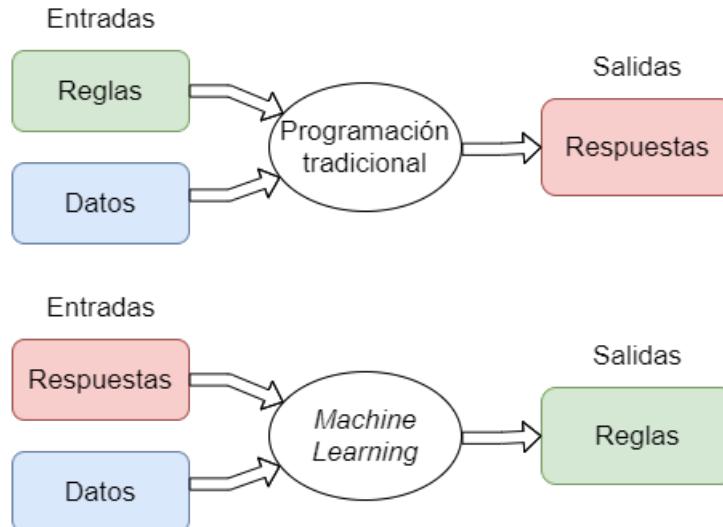


Figura 3.1: Comparación entre la programación tradicional y el Aprendizaje Automático.

Aprendizaje supervisado En este tipo de aprendizaje, los datos de entrenamiento están etiquetados, es decir, incluyen las soluciones. Un problema característico de este tipo de aprendizaje es la clasificación. Otro problema típico son tareas de predicción numéricas, como por ejemplo el precio de una casa dadas ciertas características de esta.

Algunos de los algoritmos con aprendizaje supervisado más importantes son los siguientes:

- *k-Nearest Neighbors*.
- Regresión lineal.
- Regresión logística.
- *Support Vector Machines* (SVMs).
- Árboles de decisión y Bosques Aleatorios.
- Redes Neuronales (algunas pueden ser no supervisadas).

Aprendizaje no supervisado En el aprendizaje no supervisado los datos no están etiquetados, por lo tanto el algoritmo trata de aprender por su cuenta. Este tipo de algoritmos se utilizan en problemas de agrupación o asociación de datos.

Aprendizaje semi-supervisado En este caso, el entrenamiento se realiza con una parte de los datos etiquetados y otra sin etiquetar. En estos algoritmos se suelen combinar las características de los dos anteriores. En primer lugar, se agrupan los datos mediante las técnicas utilizadas en el aprendizaje no supervisado. Y, en segundo lugar, se etiquetan estos grupos utilizando los datos etiquetados.

Aprendizaje por refuerzo El sistema de aprendizaje, denominado *agente*, puede observar el *entorno*, seleccionar y ejecutar acciones, y obtener recompensas o penalizaciones en función del resultado de las mismas. Por lo tanto, debe aprender por su cuenta

la mejor estrategia, denominada *política*, para obtener la mayor cantidad de recompensas. Enseñar a los robots a caminar es uno de los usos más extendidos de este tipo de aprendizaje.

3.1.2 Aprendizaje por lotes y en línea

Otro criterio para clasificar un algoritmo de Machine Learning es si dicho algoritmo puede aprender de un flujo entrante de datos.

Aprendizaje por lotes En este caso el sistema no es capaz de aprender de forma incremental, de modo que debe ser entrenado utilizando la totalidad de los datos. Esto requiere recursos computacionales y tiempo, por lo que generalmente se realiza fuera de línea. Para añadir nuevos datos al entrenamiento, es necesario volver a entrenar el sistema desde cero. Esta solución es adecuada para escenarios donde no existen cambios bruscos en los datos.

Aprendizaje en línea En cuanto al aprendizaje en línea, es posible entrenar el modelo de forma incremental mediante un flujo entrante de datos, bien de datos individuales o pequeños grupos denominados mini lotes (*mini-batches*). Cada paso en el entrenamiento es rápido y barato en términos computacionales. Este tipo de algoritmos es adecuado para sistemas con un flujo continuo de datos que necesitan adaptarse rápidamente a estos.

Un aspecto importante de estos sistemas es la velocidad de adaptación a los datos, esto se modela mediante un parámetro denominado ratio de aprendizaje. Con un ratio de aprendizaje alto, el sistema se adapta a nuevos datos de manera rápida. Sin embargo, también tiende a desechar lo aprendido con los datos antiguos con la misma velocidad.

3.1.3 Aprendizaje Basado en Instancias y Basado en Modelos

Finalmente, se puede categorizar los algoritmos de Machine Learning en función de como generalizan.

Aprendizaje Basado en Instancias El sistema aprende los ejemplos de memoria y generaliza a nuevos casos utilizando una medida de similitud.

Aprendizaje Basado en Modelos En este caso el sistema crea un modelo durante el entrenamiento. Posteriormente utiliza dicho modelo para hacer predicciones.

3.1.4 Principales desafíos del Machine Learning

En un sistema basado en Machine Learning hay dos tareas principales: recolectar datos y seleccionar el algoritmo con el que se van a entrenar. A continuación, se presentan los principales desafíos a la hora de llevar a cabo estas dos tareas principales.

Datos insuficientes Se necesita una gran cantidad de datos para que los algoritmos de Machine Learning tengan un buen rendimiento. Incluso en los casos más simples es

frecuente entrenar con miles de ejemplos. Para los problemas complejos, como el reconocimiento de imagen o voz, es habitual utilizar millones de ejemplos.

A partir de un artículo publicado en 2009 [46] Peter Norving extendió la idea de que los datos son más importantes que los algoritmos cuando se trata de problemas complejos.

Datos no representativos Para conseguir un modelo que generalice bien, es preciso utilizar datos con el mayor número de casos posible. Por ello, un conjunto de datos donde se cubren solamente una parte de los escenarios posibles no es representativo, y puede conducir a problemas de rendimiento cuando las entradas al modelo corresponden a escenarios no cubiertos.

Datos de baja calidad Si los datos contienen errores, valores atípicos o ruido, conseguir un rendimiento alto es más complicado. Por ello, organizar y filtrar los datos es una parte muy importante en todos los proyectos de Machine Learning.

Características intrascendentes Una parte crítica para el éxito de un proyecto de Machine Learning es seleccionar las características sobre las cuales se va a entrenar el modelo. Este proceso se denomina ingeniería de características y comprende los siguientes aspectos:

- Selección de características: elegir las características más útiles entre las que contiene nuestro conjunto de datos.
- Extracción de características: combinar características existentes para producir otras más apropiadas.
- Creación de nuevas características mediante la recopilación de nuevos datos.

Sobreajuste a los datos de entrenamiento (*overfitting*) El sobreajuste a los datos de entrenamiento se da cuando un algoritmo se ajusta muy bien a los datos donde entrena pero no es capaz de generalizar bien para nuevos datos. Como se observa en la Figura 3.2 en estos casos el modelo suele ser excesivamente complejo para adaptarse a la perfección durante el entrenamiento. Por otra parte, un conjunto de datos pequeño también puede llevar a un problema de sobreajuste.

Algunas soluciones a este problema pueden ser las siguientes:

- Simplificar el modelo. Esto se puede realizar utilizando menos parámetros (e.g., reduciendo el grado del polinomio si se trata de una regresión), reduciendo el número de atributos o características de los datos o introduciendo restricciones en el modelo.
- Utilizar más datos para entrenar el modelo.
- Organizar y filtrar los datos para reducir errores, valores atípicos y ruido.

Subajuste a los datos de entrenamiento (*underfitting*) En el caso del subajuste ocurre lo contrario, pues tiene lugar cuando el modelo es demasiado simple para detectar patrones en los datos. Como se observa en la Figura 3.2 un modelo lineal puede ser

demasiado simple para la naturaleza de los datos. En este caso el rendimiento tiende a ser bajo incluso en los datos de entrenamiento.

Las principales opciones para atajar este problema son:

- Utilizar un modelo más complejo, con más parámetros.
- Emplear mejores características en el algoritmo. Esto se puede conseguir utilizando las técnicas de ingeniería de características mencionadas anteriormente.
- Reducir las restricciones en el modelo (e.g., parámetros para evitar sobreajuste).

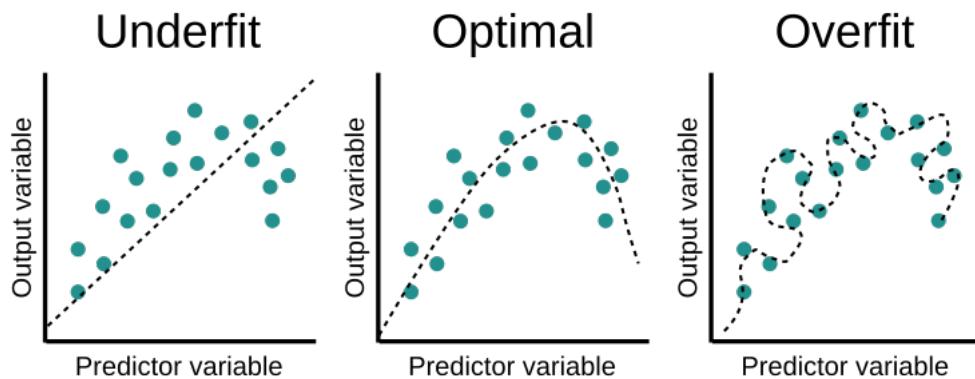


Figura 3.2: Subajuste, ajuste óptimo y sobreajuste. Fuente: <https://www.educative.io/>.

3.1.5 Métricas de rendimiento

Las métricas más habituales para evaluar el rendimiento de un algoritmo de Machine Learning son: precisión, exhaustividad, matriz de confusión, valor F1, curva ROC y AUC [47][48].

La tabla o matriz de confusión es una métrica utilizada en tareas de clasificación donde se representan las predicciones según el formato de la Tabla 3.1 en el caso de clasificadores binarios. Los valores en la tabla se definen de la siguiente manera:

- TP: verdaderos positivos. Datos clasificados como positivos correctamente.
- FP: falsos positivos. Datos clasificados como positivos incorrectamente.
- TN: verdaderos negativos. Datos clasificados como negativos correctamente.
- FN: falsos negativos. Datos clasificados como negativos incorrectamente.

	Positivo (clasificador)	Negativo (clasificador)
Positivo (real)	TP	FN
Negativo (real)	FP	TN

Tabla 3.1: Estructura de la matriz de confusión en un clasificador binario.

La precisión es la métrica más utilizada en el caso binario y multiclase, debido a su facilidad de cálculo y a que ofrece una buena evaluación del rendimiento del algoritmo [49]. Se calcula como se indica en la Ecuación 3.1.

$$Precisión = \frac{TP}{TP + FP} \quad (3.1)$$

El valor de la precisión se encuentra en el intervalo $[0,1]$, siendo 1 el valor ideal.

La exhaustividad es una métrica que permite calcular la proporción de casos positivos que han sido clasificados correctamente. Su valor se encuentra en el intervalo $[0,1]$, siendo 1 para todas las clases el valor ideal. Desde el punto de vista analítico se busca aumentar la exhaustividad sin afectar la precisión. Se calcula como se indica en la Ecuación 3.2.

$$exhaustividad = \frac{TP}{TP + FN} \quad (3.2)$$

La precisión y la exhaustividad son muy efectivas cuando se trabaja con datos equilibrados. No obstante, pierden efectividad al trabajar con datos desequilibrados [50].

La métrica valor F1 combina las dos anteriores, siendo directamente proporcional a ambas. Los valores altos del valor F1 indican que el algoritmo predice de mejor manera la clase positiva. En el caso del valor F1 se da la misma importancia a la precisión y a la exhaustividad, en caso de otorgar más peso a una de las dos métricas se utiliza el valor F2. La medida valor F1 se calcula según la Ecuación 3.3.

$$valorF1 = 2 \cdot \frac{Precisión \cdot exhaustividad}{Precisión + exhaustividad} \quad (3.3)$$

3.1.6 División de datos

La mejor forma de medir la generalización de un modelo es probarlo en nuevos casos. Una forma de hacer esto es implementar el algoritmo en el sistema real y monitorear su rendimiento. Sin embargo, esto puede resultar costoso si el modelo no rinde adecuadamente. Una alternativa menos arriesgada es dividir los datos en dos conjuntos: conjunto de entrenamiento y conjunto de prueba. De manera que el algoritmo se entrena en el conjunto de entrenamiento, y se pone a prueba con el conjunto de prueba para medir el rendimiento en nuevos casos. El error en nuevos casos se denomina error de generalización, al poner a prueba nuestro modelo con casos nuevos obtenemos una estimación de este error.

En caso de que el error en el conjunto de entrenamiento sea bajo pero el error en el conjunto de prueba es alto, tenemos un problema de sobreajuste. Normalmente se emplea el 80% de los datos para entrenar y el 20% restante para poner a prueba el modelo entrenado.

Consideremos el caso de entrenar un modelo con distintos hiperparámetros, como el ratio de aprendizaje o la regularización. Si se pone a prueba el modelo siempre con el mismo conjunto de prueba se puede producir un sesgo, adaptando el modelo y los

hiperparámetros tanto al conjunto de entrenamiento como al conjunto de prueba. Por lo tanto, estimar el error de generalización mediante el conjunto de prueba no es posible.

La solución más empleada para este problema es utilizar otro conjunto de datos denominado conjunto de validación. De este modo, se entranan diferentes modelos con distintos hiperparámetros, se selecciona el mejor modelo mediante el conjunto de validación, y finalmente se estima el error de generalización empleando el conjunto de prueba. En caso de utilizar tres subconjuntos de datos una división habitual es 60-20-20, utilizando el 60% de los datos para el entrenamiento, un 20% para la validación y el 20% restante para estimar el error de generalización [51].

3.2 Redes Neuronales Artificiales

Las Redes Neuronales Artificiales (ANN) son un tipo de Machine Learning basada en la arquitectura del cerebro. Las ANN son el núcleo del Aprendizaje Profundo (*Deep Learning*). Son versátiles, escalables y ofrecen un rendimiento muy alto en muchas tareas, por ello son la arquitectura más utilizada en Machine Learning para abordar problemas complejos. La clasificación de imágenes (e.g., Google), reconocimiento de voz (e.g., Apple) o recomendar videos a millones de usuarios (e.g., Youtube) son algunos de los ejemplos donde se emplea esta arquitectura [51].

Las neuronas parecen tener un comportamiento simple. No obstante, al haber miles de millones en el cerebro, y al estar cada neurona conectada a miles de otras neuronas, se pueden realizar tareas muy complejas. La arquitectura de las Redes Neuronales Biológicas se sigue investigando activamente. Sin embargo, algunas partes del cerebro han sido mapeadas y parece que las neuronas se organizan en capas consecutivas como se muestra en la Figura 3.3.

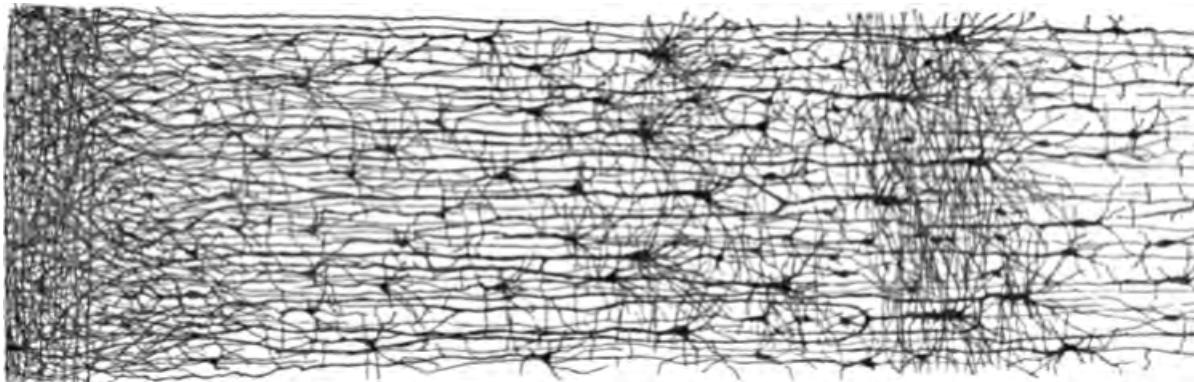


Figura 3.3: Múltiples capas en una Red Neuronal biológica. Fuente: https://en.wikipedia.org/wiki/Cerebral_cortex.

3.2.1 Perceptrón

El Perceptrón es una de las arquitecturas más básicas de las ANN. Ha sido creada por Frank Rosenblatt en 1957. Está basada en la Unidad de Umbral Lineal (LTU). En esta neurona, las entradas y salidas son números y cada conexión está asociada a

un peso. La LTU calcula la salida sumando la multiplicación de las entradas por los pesos correspondientes, después aplica la función escalón, normalmente escalón unitario o *Heaviside*, al resultado de dicho cálculo como se puede observar en la Figura 3.4. La función cuya entrada es la suma de la multiplicación de las entradas por los pesos y cuya salida es la salida de la neurona, en este caso la función escalón, se denomina función de activación.

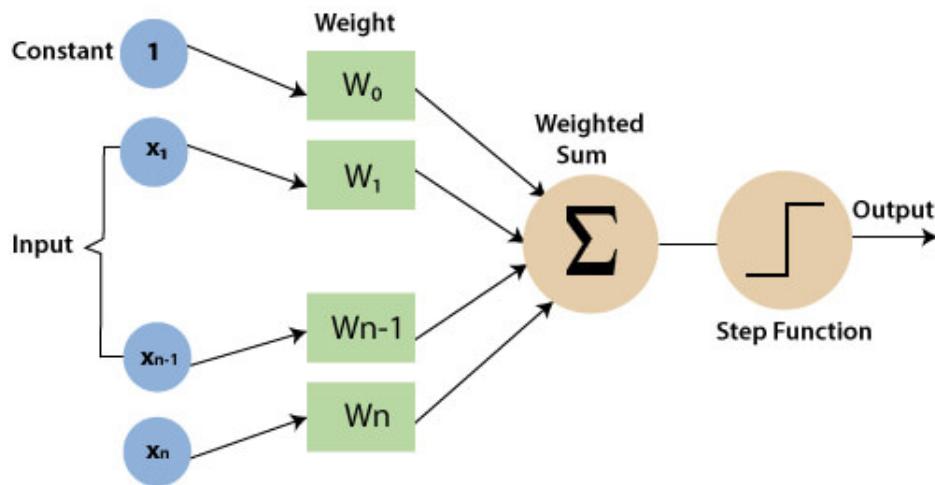


Figura 3.4: Arquitectura LTU. Fuente: <https://www.javatpoint.com/>.

Entrenar una LTU consiste en encontrar los valores adecuados de los pesos para realizar la tarea propuesta correctamente. Un Perceptrón es la composición de varias LTUs en una misma capa, donde cada neurona está conectada a todas las entradas. Esto se logra con las denominadas neuronas de entrada, cuya salida es igual que su entrada. Por otra parte, se añade una neurona de sesgo o *bias* cuyo valor de salida siempre es uno. En la Figura 3.5 se puede observar el esquema de un Perceptrón con dos entradas y tres salidas, formado por tres LTUs.

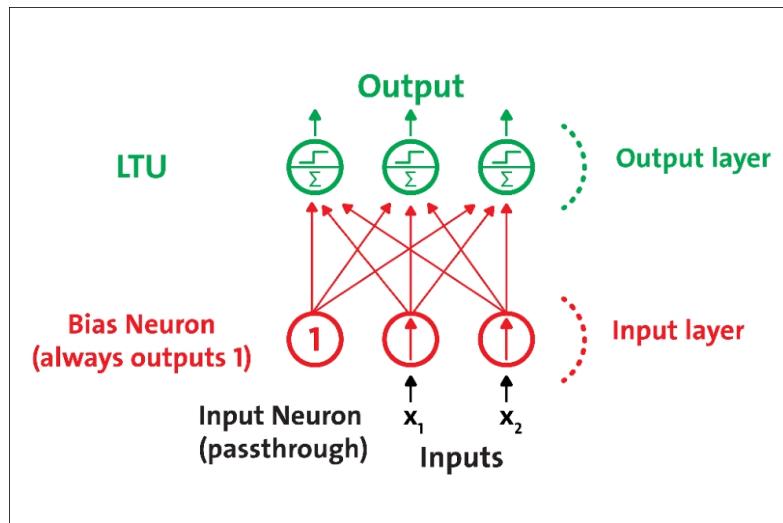


Figura 3.5: Esquema de un Perceptrón [52].

Entrenar una Perceptrón consiste en encontrar los pesos óptimos de las LTUs que lo componen. Para ello se utiliza la Ecuación 3.4

$$w_{i,j}^{(\text{siguiente iteración})} = w_{i,j} + \eta(\hat{y}_j - y_j)x_i \quad (3.4)$$

Donde:

- $w_{i,j}$ es el peso de la conexión entre la neurona de entrada i y la neurona de salida j .
- x_i es la entrada i para la instancia actual de entrenamiento.
- \hat{y}_j es la salida de la neurona de salida j para la instancia actual de entrenamiento, es decir, la salida estimada.
- y_j es la salida deseada de la neurona de salida j .
- η es el ratio de aprendizaje.

El Perceptrón tiene serios inconvenientes, el más destacado es el hecho de que no pueden resolver algunos problemas básicos, como la operación lógica XOR. Sin embargo, algunas de estas limitaciones se pueden eliminar mediante el uso del Perceptrón Multicapa (MLP).

3.2.2 Perceptrón Multicapa y Retropropagación

Un MLP está compuesto de una capa de entrada, una o más capas de LTUs, denominadas capas ocultas, y una capa final de LTUs denominada capa de salida (Figura 3.6). Todas las capas excepto la última incluyen una neurona de sesgo y están completamente conectadas con la siguiente capa. Cuando una Red Neuronal Artificial contiene dos o más capas ocultas se denomina Red Neuronal Profunda (DNN).

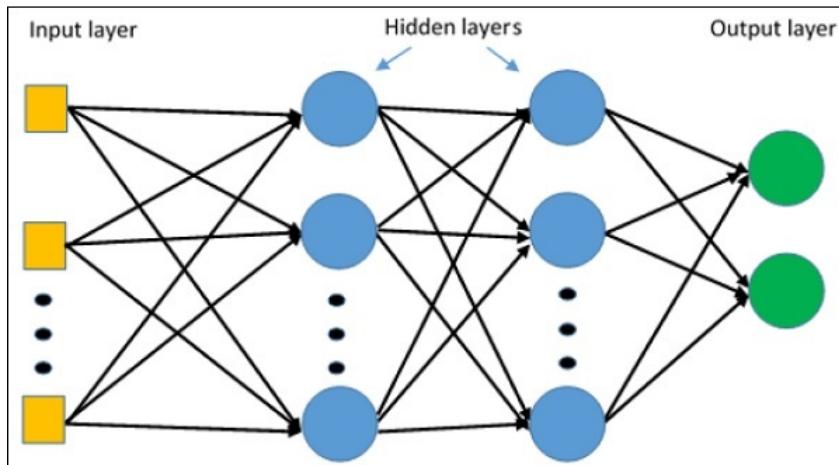


Figura 3.6: Perceptrón Multicapa. Fuente: <https://www.tutorialspoint.com/>.

Tras años donde los investigadores no habían conseguido un modelo para entrenar las MLP, D. E. Rumelhart et al. propusieron en [53] el entrenamiento mediante retropropagación. Este algoritmo de entrenamiento consiste en lo siguiente: para cada instancia de entrenamiento, el algoritmo la procesa y calcula la salida de cada neurona. Después mide el error de salida, es decir, la diferencia entre la salida deseada de la red y la salida obtenida. Con esto calcula la contribución a este error de cada neurona en la última capa oculta. Tras esto, calcula cuantas de estas contribuciones provienen de cada neurona de la capa anterior, y así hasta llegar a la capa de entrada. El paso inverso permite calcular el gradiente del error en todos los pesos de conexión.

Para que este algoritmo funcione correctamente los autores han reemplazado la función escalón por la función logística $1/(1 + e^{-z})$. Esto es esencial dado que la derivada de la función escalón es nula (excepto en el punto 0, donde no es derivable). En cambio, la derivada de la función logística es distinta de cero en todos los puntos, lo cual permite el uso de la técnica de Descenso de Gradiente. El algoritmo de retropropagación se puede utilizar con una gran variedad de funciones de activación que se discuten en el siguiente capítulo.

3.2.3 Redes Neuronales Convolucionales

En el año 1959, los neurofisiólogos David Hubel y Torsten Wiesel describieron las células simples y complejas en la corteza visual del cerebro. Propusieron que ambos tipos de células son usadas en el reconocimiento de patrones del cerebro. Las células simples detectan bordes y barras en orientaciones particulares, la Figura 3.7 es un filtro típico de una célula simple.

Las células complejas también detectan bordes y barras. Sin embargo, a diferencia de las células simples, los bordes y las barras pueden situarse en varias localizaciones y aún así ser detectadas. Las células simples podrían, por ejemplo, detectar bordes en la parte superior de la imagen. Mientras que las células complejas podrían detectarlos en la parte inferior, mediana y superior. Esta propiedad de las células complejas se denomina invarianza espacial [54]. En el año 1962, Hubel y Wiesel propusieron que las células complejas pueden lograr la invarianza espacial sumando la salida de varias células simples que detectan la misma orientación pero en diferentes partes (ver Figura 3.8). Esta

idea se ha extendido entre otros investigadores en los siguientes años [55].

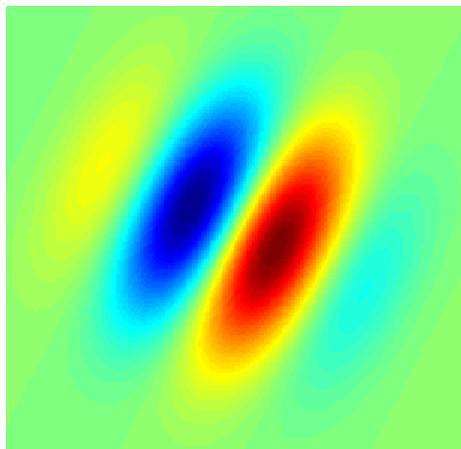


Figura 3.7: Campo receptivo tipo filtro de Gabor de una célula simple. Fuente: <https://towardsdatascience.com/>.

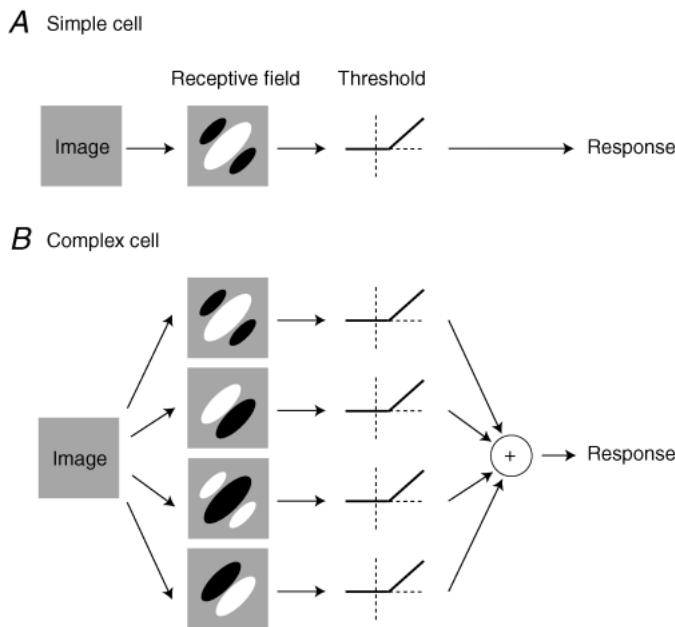


Figura 3.8: Modelos de células simples y complejas propuestos en [55]

En la década de 1980 el Dr. Kunihiko Fukushima propuso la idea del Neocognitrón [56], una Red Neuronal multicapa jerárquica para el reconocimiento de caracteres escritos a mano. Este modelo incluye componentes denominados *S-cells* y *C-cells*. Estos componentes representan operaciones matemáticas. Los *S-cells* se encuentran en la primera capa del modelo y las *C-cells* en la segunda capa. La idea es trasladar la idea de Hubel y Wiesel a un modelo computacional para el reconocimiento de patrones visuales.

El primer trabajo destacado en CNNs modernas, inspirado por el Neocognitrón, se publicó en el año 1998 [57]. En dicho artículo Yann LeCun et al. entrena una CNN que añade características simples progresivamente hasta detectar características complejas, logrando un rendimiento sin precedentes para reconocer caracteres escritos a mano.

El bloque más importante de una CNN es la capa de convolución. Las neuronas en la primera capa de convolución no están conectadas a todas las entradas (los píxeles de la imagen de entrada), sino que solamente a las entradas de su campo receptivo. Esto se puede observar en la Figura 3.9 que representa la red utilizada para el reconocimiento de caracteres escritos a mano mencionada anteriormente.

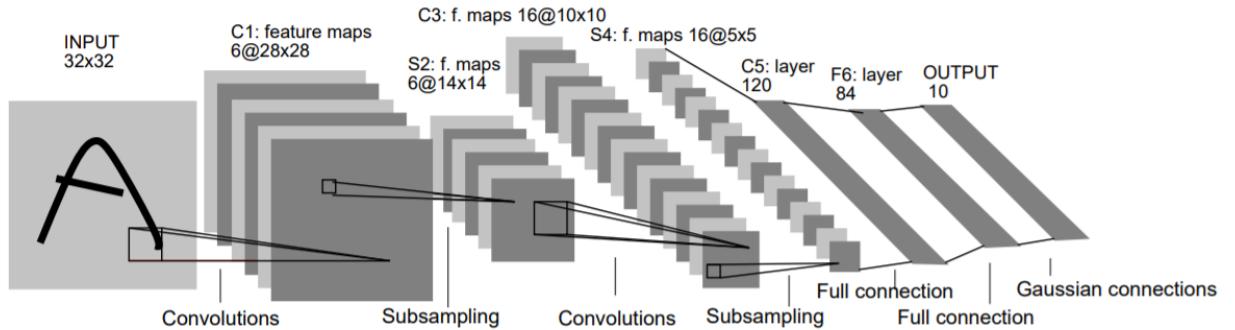


Figura 3.9: Arquitectura de la CNN LeNet-5 [57]

De esta manera, una neurona localizada en la fila i y columna j de una capa dada está conectada con las neuronas de salida de la capa anterior localizadas en las filas i a $i + f_h - 1$ y las columnas j a $j + f_w - 1$, donde f_h y f_w representan la altura y ancho del campo receptivo [51]. En aquellos lugares donde haga falta, se añade un relleno (*padding*) para conseguir que una capa tenga las suficientes filas y columnas para conectarse con todos las neuronas de la capa siguiente. Es frecuente utilizar ceros como relleno (ver Figura 3.10).

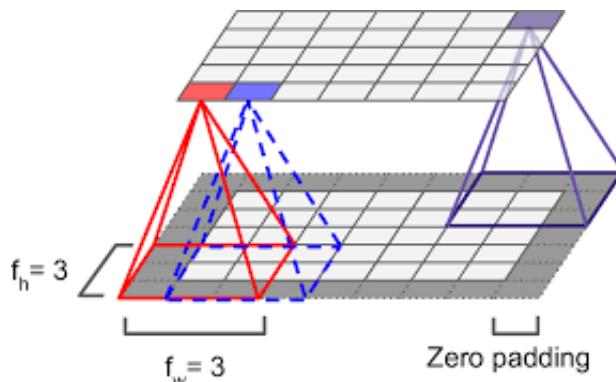


Figura 3.10: Conexiones entre dos capas consecutivas en una CNN [51].

Por otro lado, también es posible separar los campos receptivos para conectar una capa a otra de un tamaño muy superior. La distancia entre dos campos receptivos seguidos se denomina zancada (*stride*). En este caso, una neurona localizada en la fila i y columna j estará conectada a las salidas de las neuronas de las filas i a $i \times s_h + f_h - 1$ y de las columnas j a $j \times s_w + f_w - 1$. En la Figura 3.11 se puede observar un ejemplo de un campo receptivo de 3×3 y una zancada de 2×2 .

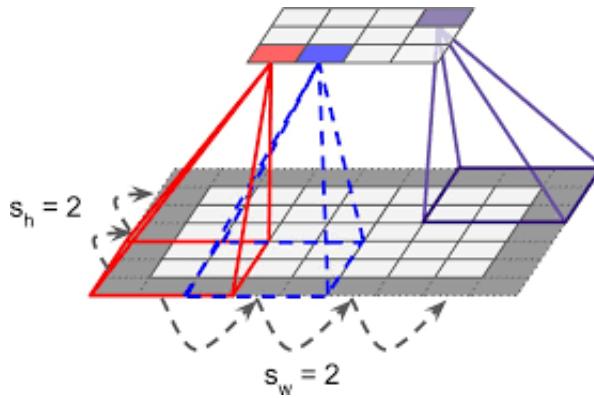


Figura 3.11: Conexiones entre dos capas consecutivas en una CNN con zancada [51].

En las Redes Neuronales Convolucionales, los pesos de una neurona son representados por una matriz del tamaño del campo receptivo. Un conjunto específico de pesos es lo que se denomina filtro o matriz de convolución. En el caso de un filtro 5×5 donde todas las columnas son ceros menos la central, cuyos elementos son unos, se obtiene un filtro vertical, dado que una neurona con estos pesos ignora todos los elementos de su campo receptivo a excepción de la columna central. Otros tipos de filtros se pueden obtener de forma análoga.

Una capa formada por neuronas que utilizan todos el mismo filtro crea un mapa de características donde se destacan las zonas de la imagen más similares al filtro. Durante el entrenamiento, las CNN encuentran los filtros más útiles para una tarea específica y aprende a combinarlos para crear patrones más complejos. La Ecuación 3.5 indica como se calcula la salida de una neurona en una capa de convolución.

$$z_{i,j,k} = b_k + \sum_{u=1}^{f_h} \sum_{v=1}^{f_w} \sum_{k'=1}^{f_{n'}} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{con } \begin{cases} i' = u \cdot s_h + f_h - 1 \\ j' = v \cdot s_w + f_w - 1 \end{cases} \quad (3.5)$$

- $z_{i,j,k}$ es la salida de la neurona localizada en la fila i , columna j en el mapa de características k de la capa de convolución l .
- s_h y s_w representan la altura y el ancho de la zancada, respectivamente.
- f_h y f_w representan la altura y el ancho del campo receptivo, respectivamente.
- $f_{n'}$ es el número de mapas de características de la capa anterior ($l - 1$).
- b_k es el término de sesgo para el mapa de características k en la capa l .
- $x_{i',j',k'}$ es la salida de la neurona localizada en la capa $l - 1$, fila i' , columna j' y mapa de características k' .
- $w_{u,v,k',k}$ es el peso de la conexión entre cualquier neurona en el mapa de características k de la capa l y su entrada localizada en la fila u , columna v (respecto al campo receptivo de dicha neurona) y mapa de características k' .

Tras estudiar y analizar los fundamentos de las Redes Neuronales en este apartado, en los siguientes se van a presentar las herramientas utilizadas para el desarrollo e implementación de la Red Neuronal utilizada en este trabajo.

3.3 Tecnología CUDA de NVIDIA

Entrenar un algoritmo complejo de Aprendizaje Automático puede consumir muchos recursos y, especialmente, tiempo. Esto se debe principalmente al limitado número de núcleos en las Unidades de Procesamiento Central (CPUs). La alternativa a las CPUs son las Unidades de Procesamiento Gráfico (GPUs) dado que cuentan con un número muy superior de núcleos.

Por otro lado, a la hora de entrenar Redes Neuronales Profundas (DNN) es frecuente manejar grandes cantidades de datos. Por lo tanto un mayor ancho de banda de memoria afecta positivamente al rendimiento. A continuación, se realiza una pequeña comparativa entre una CPU y una GPU de última generación.

Por un lado, la CPU es una Intel Core i7-11700K lanzada al mercado a finales de marzo de 2021. Por otro lado, la GPU es una NVIDIA GeForce RTX 3080, lanzada al mercado en septiembre del año 2020. En ambos caso se trata de componentes de gama alta doméstica. En cuanto al número de núcleos, la CPU cuenta con 8 núcleos y 16 subprocesos, mientras que la GPU posee 8704 núcleos CUDA. En lo referente al ancho de banda de memoria, la CPU soporta un máximo de 50 GB/s y la GPU 760.3 GB/s [59, 60]. Como se puede observar, la GPU cuenta con una cantidad muy superior de núcleos, así como de ancho de banda de memoria. Por todo ello, cuando se trata de sistemas complejos con una gran cantidad de datos, las GPUs son más utilizadas.

Para poder entrenar un algoritmo de Aprendizaje Automático en una GPU de manera práctica, entre otros usos, la empresa líder en el sector de las GPUs, NVIDIA, ha desarrollado la tecnología CUDA. Esta tecnología se describe en el siguiente apartado.

3.3.1 Tecnología CUDA

Compute Unified Device Architecture (CUDA) es una plataforma de computación paralela y *Application Programming Interface* (API) desarrollados por NVIDIA [61]. CUDA proporciona la posibilidad de acelerar la ejecución de código mediante el uso de GPUs compatibles con esta tecnología. Concretamente, las librerías (APIs) que proporciona permiten acceder al conjunto de instrucciones de la GPU, así como a elementos de computación paralelos.

CUDA está diseñada para trabajar con lenguajes de programación como C/C++ y Fortran, lo cuál la convierte en una tecnología que puede ser utilizada por programadores sin experiencia en programación de gráficos. También existen implementaciones de CUDA en lenguajes de alto nivel como Python, donde CuPy [63] y PyCuda [62] destacan.

A continuación, se exponen algunos de los conceptos principales detrás del modelo de programación CUDA en C++.

Kernels CUDA C++ extiende C++ al permitir la definición de funciones, denominadas *kernels*, que cuando se llaman son ejecutadas N veces por N hilos CUDA distintos. Un *kernel* se define mediante el especificador de declaración `__global__` y el número de hilos CUDA que ejecutan dicho *kernel* se especifican utilizando sintaxis de configuración de ejecución. Cada hilo que ejecuta el *Kernel* posee un identificador de hilo único acce-

sible desde dentro del *kernel*. En la Figura 3.12 se muestra un ejemplo sencillo obtenido de la documentación de NVIDIA.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Figura 3.12: Ejemplo de un *kernel* de CUDA [61].

Jerarquía de Hilos La variable `threadIdx` es un vector con tres componentes, por lo tanto los hilos pueden ser identificados utilizando una, dos o tres dimensiones de la variable `threadIdx`, formando bloques de hilos de una, dos o tres dimensiones. Esto proporciona una manera sencilla de invocar computación a través de elementos en dominios como un vector, matriz o volumen.

El índice de un hilo y su identificador se relacionan entre sí de manera directa. En el caso de un bloque unidimensional son el mismo; en el caso de un bloque bidimensional de tamaño (D_x, D_y) , el identificador de un hilo de índice (x, y) es $(x + y \cdot D_x)$; para un bloque tridimensional de tamaño (D_x, D_y, D_z) , el identificador de un hilo de índice (x, y, z) es $(x + y \cdot D_x + z \cdot D_x \cdot D_y)$. En la Figura 3.13 se puede ver un código que suma dos matrices A, B de tamaño $N \times N$ y guarda el resultado en una matriz C .

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                      float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Figura 3.13: Suma de dos matrices con CUDA mediante manejo de hilos [61].

Los hilos de un bloque residen en el mismo núcleo del procesador y comparten los recursos de memoria del mismo. Por ello, el número de hilos por cada bloque está limitado. Las GPUs de última generación pueden tener hasta 1024 hilos por bloque.

Los bloques se organizan en redes de bloques de una, dos o tres dimensiones. El número de bloques de hilos en una red generalmente está marcado por la cantidad de datos procesada, que normalmente es superior al número de procesadores en el sistema. En la Figura 3.14 se muestra una red de bloques de hilos bidimensional.

Jerarquía de Memoria Los hilos CUDA pueden acceder a diferentes regiones de memoria. Cada hilo tiene una memoria local privada. Por otro lado, cada bloque de hilos posee una región de memoria compartida que es visible a todos los hilos que lo forman. Además, todos los hilos tienen acceso a la memoria global.

Existen adicionalmente dos regiones de memoria de solo lectura accesibles para todos los hilos: las regiones para constantes y texturas. Las regiones de memoria visibles para todos los hilos, global, constantes y texturas, están optimizados para distintos usos. En la Figura 3.15 se ilustra la arquitectura de la memoria en CUDA.

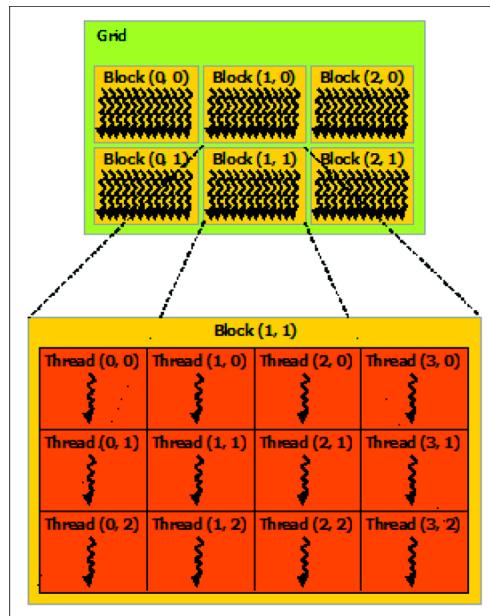


Figura 3.14: Red de Bloques de Hilos en una GPU de NVIDIA [61].

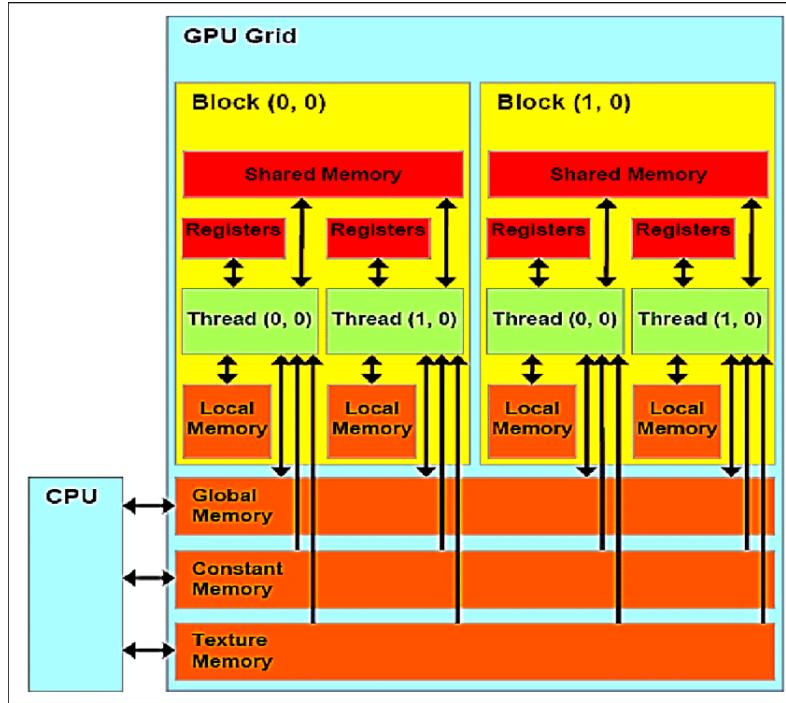


Figura 3.15: Jerarquía de Memoria en CUDA [64].

3.3.2 CUDA con Redes Neuronales Convolucionales

Los mayores inconvenientes de las CNNs son su compleja implementación, cuya solución es el uso de librerías de alto nivel como TensorFlow, y el tiempo de entrenamiento. Dado que el entrenamiento de las CNNs es intenso a nivel computacional y requiere muchos datos para un correcto rendimiento, entrenar una CNN puede llevar días o semanas. En este tipo de redes, hay un gran número de operaciones de coma flotante y una baja transferencia de datos en cada paso del entrenamiento. Por ello, las GPUs de Propósito General (GPGPU) son una herramienta adecuada para este tipo de tareas [65].

La gran desventaja a la hora de utilizar GPUs es que se requieren unos amplios conocimientos del hardware, al igual que de programación gráfica (OpenGL o DirectX, por ejemplo). Sin embargo, como hemos visto en el apartado anterior, la tecnología CUDA ha eliminado estas barreras, permitiendo utilizar toda la potencia de las GPUs sin necesidad de tener un gran conocimiento técnico.

Pese a que CUDA presenta algunas desventajas, como la limitación a variables de precisión simple, esto no afecta a la hora de implementar Redes Neuronales. En [65] D. Strigl et al. implementan una librería en C++ utilizando CUDA con el propósito de acelerar el entrenamiento de las CNNs. Pese a que este trabajo data del año 2010, cuando las GPUs todavía no contaban con tanta ventaja de núcleos y ancho de banda de memoria como en la actualidad, los resultados indican una clara superioridad respecto al entrenamiento con CPUs.

Además de la librería para la GPU, los modelos se han entrenado utilizando dos implementaciones para CPU, una sin optimizaciones y otra con optimizaciones mediante el uso de la librería IPP de Intel (*Intel Performance Libraries*). Los modelos utilizados en este estudio han sido SimardNet [58] y LeNet5 [57]. Los resultados de este estudio

se muestran en la Figura 3.16, donde se puede observar que la diferencia es más clara a medida que aumenta el tamaño de las imágenes de entrada.

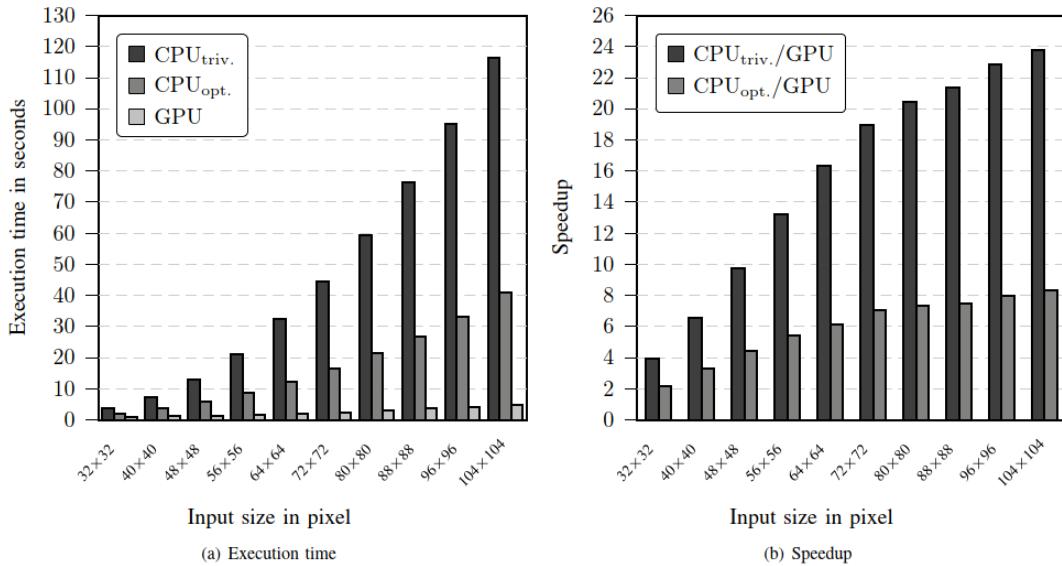


Figura 3.16: Tiempo de ejecución (a) y comparación (b) de las implementaciones en [65].

Si se compara la implementación optimizada con una CPU ($CPU_{opt.}$) y la realizada con CUDA (GPU), se puede observar que para un tamaño de imagen de 104×104 la implementación con CUDA es 8 veces más rápida. Además, dada la tendencia que se observa, se puede estimar que para tamaños de imagen mayores esta diferencia puede ser aún más grande.

En el estudio también se tiene en cuenta el consumo energético a la hora de entrenar. Aunque en la actualidad el consumo de una GPU normalmente es tres veces superior al de una CPU de la misma gama, esto se ve compensado dado que la GPU necesita mucho menos tiempo para entrenar, por lo que el consumo total es menor al utilizar la GPU.

Dada la gran ventaja que poseen las GPUs sobre las CPUs a la hora de entrenar, existen diversas implementaciones de librerías de Redes Neuronales basadas en CUDA. En [66] X. Li et al. realizan un análisis del rendimiento de distintas implementaciones de CNNs basadas en GPU. En este caso se han elegido Caffe [67], Torch-cunn [68], Theano-CorrMM, Theano-fft [69], cuDNN [70], cuda-convnet2 [71] y fbfft [72] como implementaciones representativas. En la Figura 3.17 se pueden observar los resultados de este trabajo. Donde se entrena diversas CNNs reales con distintos parámetros e hiperparámetros (tamaño del lote, tamaño de imágenes, número de filtros, tamaño del núcleo y zancada).

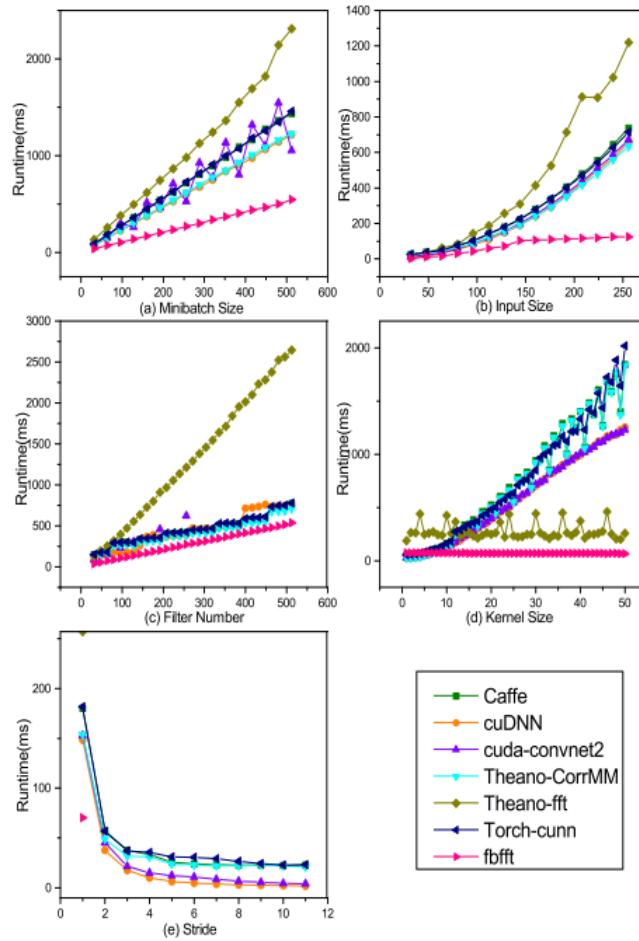


Figura 3.17: Comparación del tiempo de ejecución de las implementaciones de CNN estudiadas en [66].

La conclusión del estudio es que la implementación fbfft es la más rápida, siendo cuDNN la segunda mejor. Sin embargo, el consumo de memoria (ver Figura 3.18) muestra un consumo claramente superior en la implementación fbfft. Por ello, a la hora de elegir una implementación de CNN en CUDA con la intención de entrenar en una GPU con memoria limitada, buscando el mejor equilibrio entre consumo de memoria, velocidad y flexibilidad, cuDNN se convierte en la mejor opción.

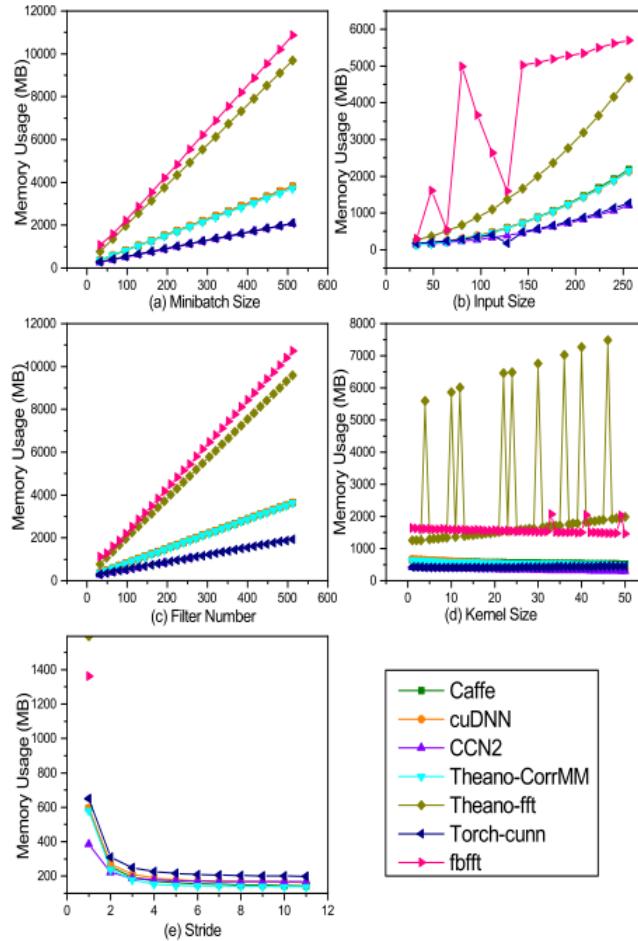


Figura 3.18: Memoria utilizada en las implementaciones de CNN estudiadas en [66].

3.4 TensorFlow

TensorFlow es una librería de código abierto desarrollada por Google, específicamente por el equipo Google Brain, principalmente para aplicaciones de Aprendizaje Profundo [21]. También soporta algoritmos de Aprendizaje Automático convencionales. TensorFlow acepta datos en forma de vectores multidimensionales denominados tensores. Por otro lado, funciona sobre la base de grafos de flujo de datos, con nodos y aristas. Esto simplifica su ejecución de manera distribuida a través de grupos de ordenadores usando GPUs.

TensorFlow está desarrollado en C++, Python y CUDA, y es posible integrarlo con otros lenguajes como Java y R. Además, dado que integra la tecnología CUDA, permite el uso de la GPU para entrenar los algoritmos. La primera versión de TensorFlow se lanzó en el año 2015. Sin embargo, a finales del año 2019 se lanzó TensorFlow 2.0, que integra Keras, permitiendo construir modelos de forma muy sencilla. La estructura de TensorFlow 2.0 se muestra en la Figura 3.19.

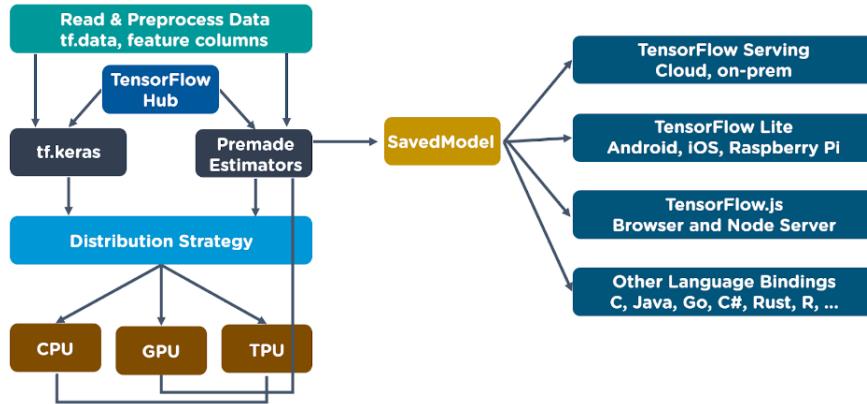


Figura 3.19: Arquitectura de TensorFlow 2.0. Fuente: <https://www.simplilearn.com>.

Por otro lado, existe también una versión de TensorFlow para dispositivos móviles (Android e iOS) y sistemas embebidos (Linux y microcontroladores), denominada TensorFlow Lite. TensorFlow Lite permite convertir un modelo de TensorFlow en un archivo comprimido FlatBuffer (.tflite) mediante TensorFlow Lite Converter, y posteriormente implementar dicho modelo en un sistema embebido o un dispositivo móvil. TensorFlow Lite optimiza el modelo original de TensorFlow para obtener un modelo con menos carga computacional manteniendo un rendimiento similar.

Para elegir la librería sobre la que se va a desarrollar este trabajo se han considerado otras opciones como PyTorch, Caffe y MXNet, que se describen a continuación.

- PyTorch: es una librería de aprendizaje profundo creada por Facebook y lanzada en el año 2016. Se puede utilizar con Python, Java y C++ [16]. Sus principales características son las siguientes:
 - Soporta entrenamiento con CPU, GPU y TPU (*Tensor Processing Unit*).
 - Entrenamiento distribuido en varias máquinas o GPUs.
 - Es más complejo que Keras, pero es una de las opciones más sencillas de utilizar.
 - Ofrece mucha flexibilidad para personalizar el modelo y el entrenamiento.
- Caffe: es una librería de aprendizaje profundo desarrollada por investigadores de la UC Berkeley en el año 2013. Originalmente estaba desarrollada en C++ pero también ofrece una interfaz en Python [67]. Sin embargo, esta librería está anticuada dado que PyTorch incluye una versión mejorada de esta librería, Caffe2. La gran ventaja de Caffe es la velocidad. A pesar de ello, su uso está muy limitado debido a que la documentación es compleja, la comunidad no es tan grande como en el caso de TensorFlow o PyTorch y además está orientada a investigadores expertos en la materia.
- MXNet: esta librería de aprendizaje profundo ha sido creada por la *Apache Software Foundation*. Soporta ocho lenguajes de programación y está integrado en servicios de computación en la nube como Amazon Web Services y Microsoft Azure [73]. MXNet presenta varias ventajas como la posibilidad de entrenar utilizando la GPU y de forma distribuida, además de tener una comunidad grande detrás. Sin

embargo, al proporcionar una interfaz más compleja que TensorFlow y carecer de algunas funcionalidades de PyTorch, su uso es muy inferior a estas.

Tras analizar las distintas opciones, los dos candidatos más claros han sido TensorFlow (que integra Keras) y PyTorch debido a que poseen una interfaz sencilla, un buen rendimiento al poder entrenar con GPUs, una gran variedad de módulos y una documentación oficial muy extensa y fácil de seguir. Se ha elegido TensorFlow dado que es más sencillo de utilizar, y por lo tanto una buena opción para un principiante. Por otro lado, TensorFlow utiliza internamente cuDNN y, como hemos analizado en el apartado anterior, ofrece el mejor equilibrio entre consumo de memoria, velocidad y flexibilidad. Finalmente, dispone de más documentación y una comunidad más grande que PyTorch dado que es el marco de trabajo más utilizado para desarrollar Redes Neuronales en la actualidad.

3.5 Raspberry Pi

La Raspberry Pi es un Ordenador de Placa Única (SBC) de bajo coste y cuyo tamaño es de 85.6 mm × 56.5 mm (ver Figura 3.20). El proyecto Raspberry Pi nace en Reino Unido con el objetivo de estimular la enseñanza de programación y ciencias de la computación. A pesar de su bajo precio, alrededor de 45€ para la Raspberry Pi 4B con 4 GB de RAM, ofrece la posibilidad de instalar Linux e interactuar con ella como si de un ordenador normal se tratara. Sin embargo, necesita de una unidad de almacenamiento externa, normalmente una tarjeta MicroSD, ya que no dispone de almacenamiento interno.

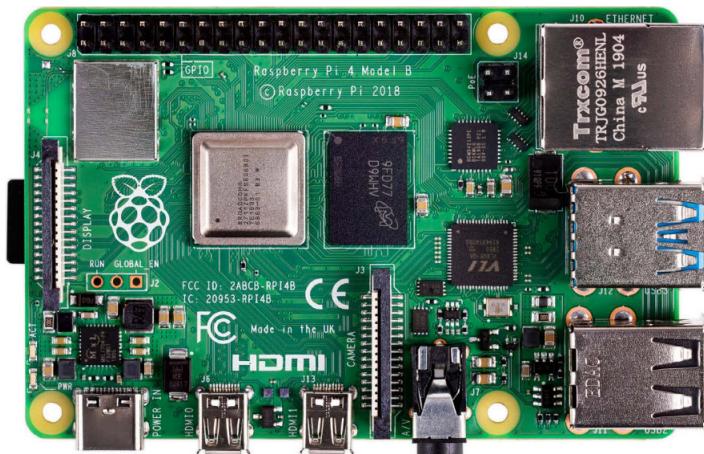


Figura 3.20: Raspberry Pi 4B. Fuente: <https://www.raspberrypi.org>.

Una de las principales ventajas que supone el uso de la Raspberry Pi es la gran comunidad de usuarios que existe y la inmensa cantidad de proyectos disponibles, desde servidores web hasta consolas.

En este proyecto se ha elegido la última versión disponible en este momento, la Raspberry Pi 4B de 4 GB. Esta placa cuenta con una CPU Broadcom BCM2711 Quad core Cortex-A72 a 1.5 GHz, 4 GB de SDRAM LPDDR4-3200, un puerto de cámara MIPI CSI de dos líneas, un puerto de pantalla MIPI DSI de dos líneas, dos puertos micro-

HDMI y cuatro puertos USB. Como sistema operativo se ha utilizado la distribución de Linux Raspberry Pi OS, anteriormente conocida como Raspbian.

3.6 Otro software empleado

Para desarrollar el código se ha empleado el Entorno de Desarrollo Integrado (IDE) PyCharm Community. Se trata de un IDE gratuito especializado para el lenguaje Python.

Para el control del desarrollo del código se ha empleado la herramienta Git Extensions, que ofrece una interfaz gráfica para interactuar con repositorios Git, así como Git Bash, que permite interactuar desde una consola. Esto se debe a que, si bien es cierto que Git Extensions ofrece la mayoría de las funciones propias de los repositorios Git, algunas acciones solamente pueden ser realizadas a través de una consola de Git.

Capítulo 4

Desarrollo

En este capítulo se detalla el desarrollo de este Trabajo Fin de Grado. Se organiza en tres bloques principales: recogida de datos, entrenamiento del modelo, e implementación en Raspberry Pi. Para el desarrollo del código se ha utilizado un repositorio Git desde el primer momento, facilitando la trazabilidad del mismo.

4.1 Datos para el entrenamiento

El primer paso para desarrollar un modelo de Machine Learning es conseguir los datos para poder entrenar y validar dicho modelo. Tras esto, es importante filtrar y adecuar los datos para comenzar a entrenar. A continuación, se describe como se han llevado a cabo estas tareas.

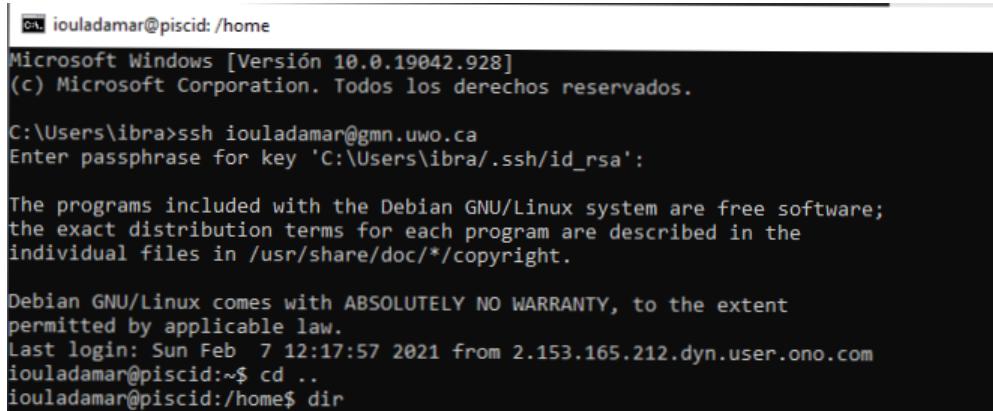
4.1.1 Adquisición de datos

Con el objetivo de entrenar el modelo con datos relevantes, es necesario que estos sean capturados por cámaras All-Sky. Esto se debe a que este trabajo persigue automatizar la detección de meteoros que se lleva a cabo en el observatorio Montegancedo de la UPM, perteneciente a la red de observatorios del Proyecto Europeo FAETON, donde se utilizan este tipo de cámaras para capturar los eventos.

Adquirir una cantidad significativa de imágenes de meteoros capturadas por cámaras All-Sky puede resultar muy complicado si se recopilan datos de distintas fuentes. Especialmente si los datos no están etiquetados. Para simplificar este proceso contacté con Denis Vida, un investigador del Departamento de Física y Astronomía en la Universidad de Ontario Occidental, Canadá. Este investigador me facilitó el acceso a un servidor de la Universidad de Ontario Occidental con datos de detecciones de meteoros en todo el mundo.

Acceder a este servidor se realiza de manera sencilla mediante SSH con una clave RSA como se puede apreciar en la Figura 4.1. En el directorio principal, *home*, se encuentran diversas carpetas. Los dos primeros caracteres del nombre de cada carpeta indican el país de donde proceden los datos, seguidos de estos caracteres encontramos un número para agrupar los datos de cada observatorio dentro de cada país. Dentro de

la carpeta de un observatorio de un país, existe una carpeta para cada día, donde se encuentran todos los archivos relacionados con las imágenes capturadas en dicha fecha.



```
iouladamar@piscid:/home
Microsoft Windows [Versión 10.0.19042.928]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\ibra>ssh iouladamar@gmn.uwo.ca
Enter passphrase for key 'C:\Users\ibra/.ssh/id_rsa':

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sun Feb 7 12:17:57 2021 from 2.153.165.212.dyn.user.ono.com
iouladamar@piscid:~$ cd ..
iouladamar@piscid:/home$ dir
```

Figura 4.1: Acceso al servidor de datos de la Universidad de Ontario Occidental.

En este caso, se ha trabajado únicamente con datos de observatorios de Bélgica (4), Alemania (1), Reino Unido (1) y Países Bajos (1). Dado que son los únicos datos clasificados manualmente por un científico.

Para descargar los datos desde el servidor y almacenarlos en el disco local se ha utilizado el comando SCP de Windows. En el siguiente apartado se describen los procedimientos para filtrar estos datos.

4.1.2 Filtrado de datos

Durante el filtrado de datos el primer paso ha sido eliminar los archivos no necesarios, conservando únicamente los archivos con extensión .fits, donde se encuentran las imágenes, y .txt. Este proceso se ha realizado utilizando el módulo os de Python.

El siguiente paso ha sido crear dos archivos de texto, donde uno de ellos contiene los nombres de todos los archivos .fits donde se detectan meteoros y el otro contiene los nombres de los archivos .fits donde no hay meteoros. En cada carpeta de una fecha se encuentran unos archivos .txt que comienzan con la cadena *FTPdetectinfo*. Si dichos archivos terminan en *pre-confirmation* esto indica que dentro se encuentran todos los archivos .fits que se han filtrado manualmente, tanto meteoros como no meteoros. En cambio, si no terminan de dicha manera indican que dentro se encuentran nombres de archivos .fits con meteoros. Para llevar esto a cabo se ha utilizado el módulo os de Python.

A continuación, se ha creado el conjunto de datos etiquetados para el entrenamiento. Para que TensorFlow sea capaz de generar las etiquetas automáticamente los ficheros de las imágenes deben estar separados en carpetas. Una carpeta con el nombre de la primera clase (*meteors* en nuestro caso) y otra con el de la segunda (*non_meteors*). Dentro de dichas carpetas se encuentran las imágenes correspondientes a la clase. El nombre de estos archivos debe comenzar también por el nombre de la clase (*meteors* o *non_meteors*). El formato de estos archivos es *Flexible Image Transport System* (FITS), siendo el más utilizado en el campo de la astronomía. En este caso, cada archivo representa una secuencia de 256 fotogramas, capturadas a 25 fotogramas por segundo, com-

primidas en cuatro imágenes siguiendo el formato *Four-frame Temporal Pixel* (FTP). Dichas imágenes se componen de la siguiente manera:

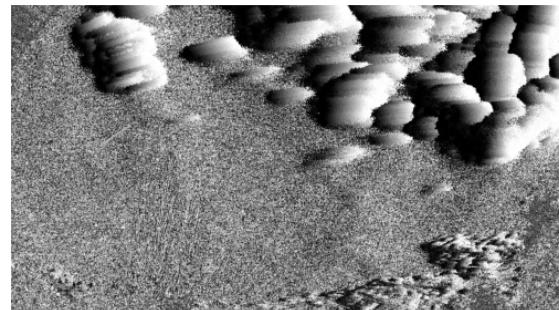
- MAXPIXEL: valor máximo de cada píxel en los 256 fotogramas.
- MAXFRAME: número de fotograma (0-255) del píxel de máximo valor.
- AVEPIXEL: media de los 256 fotogramas excluyendo el valor máximo.
- STDPPIXEL: desviación estándar en torno a la media.

Durante este trabajo, siguiendo las recomendaciones del investigador Denis Vida, se han utilizado las imágenes MAXPIXEL. Esto se debe a que los meteoros tienen un brillo muy alto, y al capturarse de noche, si se utiliza el valor máximo de cada píxel se puede detectar el meteoro con más facilidad. En la Figura 4.2 se puede observar un ejemplo de estas cuatro imágenes, donde claramente la imagen que más información aporta acerca del meteoro es la MAXPIXEL (4.2(a)).

Para extraer las imágenes MAXPIXEL de los archivos .fits se ha utilizado el módulo astropy, una colección de paquetes diseñados para su uso en astronomía. De esta manera, y mediante el uso del paquete imageio, una librería que ofrece una interfaz sencilla para leer y escribir imágenes, se ha creado el conjunto de datos base para poder entrenar el sistema.



(a) MAXPIXEL.



(b) MAXFRAME.



(c) AVEPIXEL.



(d) STDPPIXEL.

Figura 4.2: Ejemplo de archivos presentes en un archivo FITS siguiendo el formato FTP.

El siguiente paso ha sido visualizar y analizar los datos. En la Figura 4.3 se muestran algunos ejemplos. Lo primero a considerar es el tamaño de las imágenes, 1280x720

en este caso. Se trata de una resolución muy alta para un sistema en tiempo real, además de que entrenar un modelo con estos datos requiere unos recursos computacionales muy altos. Por ello, se ha modificado su tamaño con ayuda del módulo PIL, que permite la edición de imágenes de forma sencilla con Python. El tamaño final es de 640x360, reduciendo así el espacio que ocupan los datos en memoria. Por otra parte, cabe destacar que las imágenes son en blanco y negro. De esta manera, al poseer únicamente 8 bits de profundidad de color (escala de grises), ocuparán menos espacio en la memoria de la GPU a la hora de entrenar.



(a) Meteorito.



(b) Meteorito y no meteorito (avión).



(c) No meteorito (nubes).



(d) No meteorito (avión).

Figura 4.3: Ejemplos de los datos utilizados.

Por último, se han separado los datos en dos conjuntos, el de entrenamiento y el de validación. La separación se ha realizado de manera aleatoria, utilizando el módulo random de Python junto con el de shutil, que proporciona una interfaz de alto nivel para manejar archivos. El ratio de separación en un principio ha sido de un 80% para el conjunto de entrenamiento y un 20% para el de validación. Sin embargo, estos valores se han modificado a lo largo del desarrollo buscando obtener el máximo rendimiento. Como resultado de este procedimiento se ha obtenido la estructura de directorios que se representa en la Figura 4.4. En total, se han utilizado 89.263 imágenes, de las cuales 47.591 corresponden a meteoritos y las 41.672 restantes con no meteoritos.

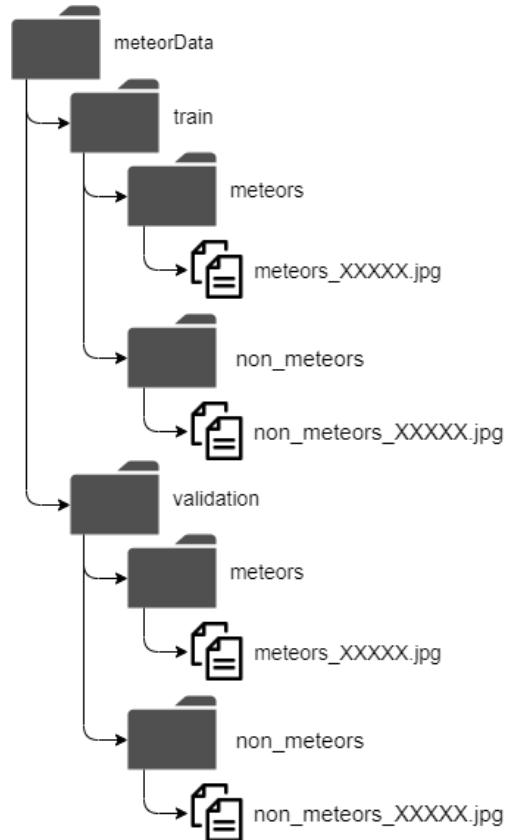


Figura 4.4: Estructura y nomenclatura de los datos filtrados.

4.2 Entrenamiento

En esta etapa, la más extensa de este trabajo, se ha desarrollado y entrenado el modelo que se va a implementar en el dispositivo embebido, una Raspberry Pi en este caso. La primera parte ha consistido en utilizar los datos filtrados en el anterior punto en el modelo, posteriormente se han ajustado diversos parámetros entrenando más de 40 modelos distintos, llevando en algunos casos hasta 24 horas.

4.2.1 Equipo utilizado durante el entrenamiento

Para entrenar tantos modelos es preciso disponer de componentes que agilicen este proceso. En este caso, todos los modelos se han entrenado con equipo doméstico. Este equipo cuenta con una CPU Intel i7 8700k, un procesador de 6 núcleos y 12 hilos con una frecuencia base de 3,70 GHz y una frecuencia máxima de 4,70 GHz. En cuanto a la GPU, un elemento clave para entrenar estos sistemas como se ha mencionado en el capítulo 3, se ha utilizado una NVIDIA GeForce GTX 1050 Ti con 4 GB de memoria interna. Esta GPU ha permitido entrenar de manera mucho más rápida, haciendo posible entrenar más modelos y, por lo tanto, mejorar el resultado final. Como memoria RAM, se ha contado con 16 GB DDR4 a 2400 MHz. Por último, los datos se han almacenado en una Unidad de estado sólido (SSD) Samsung 960 EVO para un acceso más rápido.

4.2.2 Datos de entrada

El primer paso para utilizar los datos de entrada ha sido crear un objeto de tipo *ImageDataGenerator* por cada conjunto de datos, entrenamiento y validación en este caso. La clase *ImageDataGenerator* está disponible en el paquete de preprocesamiento de Keras (*tensorflow.keras.preprocessing.image.ImageDataGenerator*). En esta clase se pueden definir las acciones que se deben realizar con las imágenes. En ambos conjuntos, entrenamiento y validación, se ha cambiado los valores de cada píxel (0-255) a un valor en el intervalo [0, 1] dividiendo el valor de cada píxel entre 255. Esto es conveniente dado que trabajar con valores altos puede disminuir el rendimiento de la red, ya que tardará más en aprender.

Por otro lado, en los datos de entrenamiento se han utilizado técnicas de aumento de datos. Algunas opciones disponibles son la rotación de la imagen desde 0 hasta 180 grados, desplazar horizontal y/o verticalmente entre un 0 y un 100%, girar la imagen a lo largo del eje horizontal en sentido contrario a las agujas del reloj, simulando una captura desde otra perspectiva, desde 0 hasta 180 grados, agrandar la imagen (zoom) un rango entre 0 y 1 o voltear la imagen horizontal y/o verticalmente.

Por último, se crean los objetos de tipo *DirectoryIterator*, que se utiliza como entrada de datos al modelo, mediante el método *flow_from_directory* de los objetos *ImageDataGenerator*. Al crear estos objetos se especifica el tamaño de la imagen, el modo de color, escala de grises siempre en nuestro caso, el modo de clases, binario en este caso y el tamaño del lote, que variará según el tamaño que ocupen las imágenes dada la limitación de memoria en la GPU. En los siguientes apartados se van a detallar todos los ajustes realizados hasta llegar al modelo final.

4.2.3 Cantidad y división de datos

Un aspecto importante a la hora de entrenar un algoritmo de Machine Learning es la cantidad de datos necesaria para conseguir una buena generalización. En principio cuantos más datos se utilicen mejor va a ser esta generalización. Sin embargo, es preciso evaluar este aspecto dado que cuantos más datos se utilicen más lento va a ser el entrenamiento. Por ello, se han realizado entrenamientos con distinta cantidad de datos, manteniendo siempre la proporción entre los datos de entrenamiento y validación. La conclusión que se ha obtenido es que para lograr una precisión alta, mayor de un 90%, así como para evitar el subajuste es necesario utilizar todos los datos.

Por otra parte, se han utilizado técnicas de aumento de datos en algunos entrenamientos dada la limitada cantidad de datos de los que se disponen y la naturaleza compleja del problema. En este caso, es importante utilizar estas técnicas de manera precisa pues de otra forma puede afectar negativamente al rendimiento del modelo. En la Figura 4.5 se puede observar como un desplazamiento lateral excesivo puede confundir a la red, dado que la etiqueta de la imagen no va a cambiar, aunque en la imagen ya no se encuentre el meteoro. Por ello, se han utilizado valores pequeños para los giros y desplazamientos, así como para agrandar la imagen. También, se han utilizado las opciones de voltear la imagen vertical y horizontalmente.

En cuanto a la división de datos se han utilizado distintas estrategias. El primer modelo se ha entrenado utilizando un 90% de los datos para el entrenamiento y el 10% restante para la validación. En los modelos posteriores se ha cambiado la proporción a un

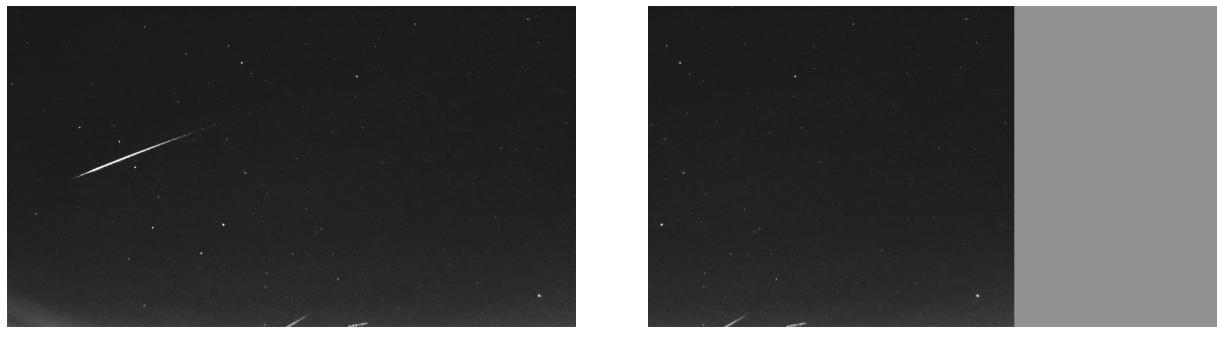


Figura 4.5: Aumento de datos con desplazamiento lateral excesivo.

80-20 dado que el primer modelo no ha generalizado bien. Tras el modelo 23, se ha observado que los datos de validación podrían ser insuficientes por lo que se ha cambiado la proporción a 70-30. Finalmente, en los últimos modelos se ha utilizado una proporción 85-15 dado que los datos para el entrenamiento eran escasos, esto ha llevado a una gran mejora de los resultados, sin tener el problema del sobreajuste.

4.2.4 Resolución de las imágenes

Un aspecto importante a la hora entrenar una Red Neuronal utilizando imágenes como datos de entrada es utilizar el mínimo tamaño posible, ya que de esta manera se puede entrenar de manera más rápida. Además, dado que el objetivo final de este trabajo es obtener una Red Neuronal que pueda ser implementada en un sistema embebido, el tamaño de las imágenes y de la red resulta crucial, pues en caso de utilizar una resolución alta y, por lo tanto, una red más compleja para procesar las imágenes, la implementación en un sistema embebido puede resultar inviable.

Durante el entrenamiento se han utilizado distintas resoluciones, comenzando por 300x300, tras lo cual se ha entrenado con una resolución de 640x360 para mantener la relación de aspecto de la imagen original. A continuación, se ha entrenado con una resolución de 480x480 y 432x432, dado que mantener la relación de aspecto no ha tenido una mejora significativa en los resultados. Finalmente, se ha utilizado una resolución de entrada de 256x256, pues se ha comprobado que permite desarrollar un modelo lo suficientemente complejo como para no tener un problema subajuste, manteniendo el equilibrio entre rendimiento y velocidad de proceso.

4.2.5 Optimizador y ratio de aprendizaje

El optimizador más básico utilizado en Redes Neuronales es el Descenso de Gradiente (*Gradient Descent*). Sin embargo, este optimizador no es el óptimo en la mayoría de ocasiones. Para escoger el optimizador y el ratio de aprendizaje utilizados durante este trabajo se ha monitorizado el ratio de aprendizaje mediante un objeto de la clase *LearningRateScheduler* del módulo *tensorflow.keras.callbacks*. Esta clase permite cambiar el ratio de aprendizaje a lo largo del entrenamiento. Debido a esto, ha sido posible ver hasta qué punto se podía incrementar el ratio de aprendizaje manteniendo la convergencia del modelo.

Para realizar esto se ha utilizado un modelo base dado que todos los modelos entrenados siguen la misma estructura. Como se puede observar en la Figura 4.6 se ha variado el valor del ratio de aprendizaje desde 10^{-6} hasta 1 durante 60 iteraciones utilizando la Ecuación 4.1. De esta manera, se puede comprobar a partir de qué valor del ratio de aprendizaje el error en el conjunto de entrenamiento aumenta.

$$lr = 10^{-6} \cdot 10^{(n/10)} \quad (4.1)$$

Donde:

- lr es el ratio de aprendizaje.
- n es la iteración actual.

Este proceso se ha realizado con tres optimizadores distintos, Adam, RMSprop y SGD, con el objetivo de compararlos. De la Figura 4.6 se puede concluir que el optimizador Adam es el que ofrece una convergencia más rápida así como el error más bajo. Por su parte, RMSprop tiene un rendimiento inferior a Adam, además de converger más lento. SGD, una versión mejorada del Descenso de Gradiente, no converge y se queda estancado en un error muy alto. El valor ideal del ratio de aprendizaje en el caso del optimizador Adam se sitúa entre 10^{-4} y 10^{-2} , por lo que se ha utilizado un valor de $5 \cdot 10^{-3}$ para asegurar la convergencia del modelo, así como un aprendizaje rápido.

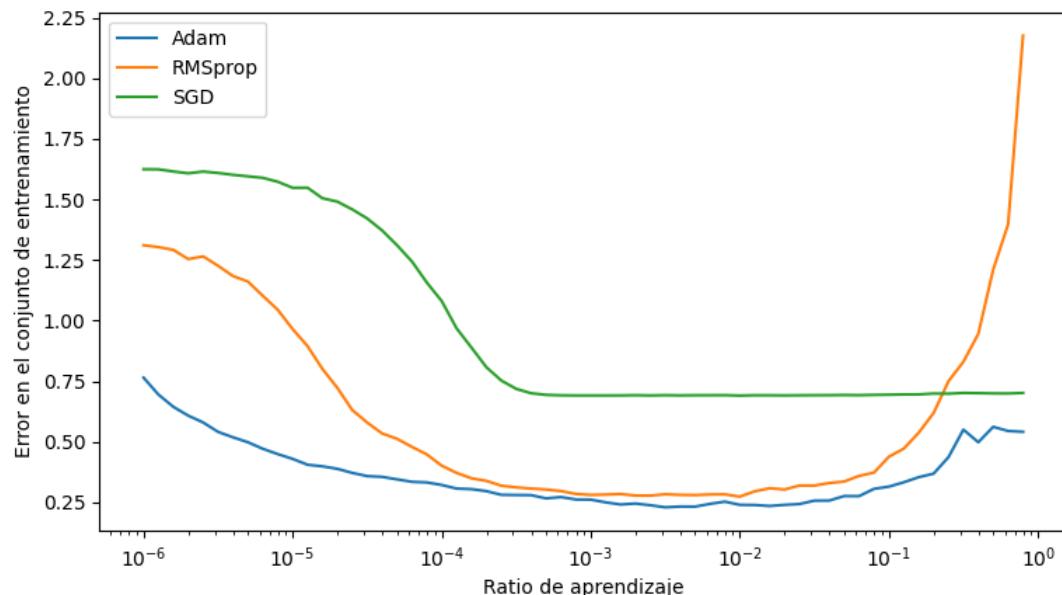


Figura 4.6: Error en conjunto de entrenamiento frente a ratio de aprendizaje con distintos optimizadores.

4.2.6 Función de activación e inicialización de pesos

En relación a la función de activación se han considerado diversas opciones. En el caso de una Red Neuronal, la función de activación es la que calcula la salida de una neurona, recibiendo como entrada las entradas a la neurona multiplicadas por su peso

correspondiente más la unidad de sesgo. Las funciones que se han probado en este caso han sido sigmoide, ReLu y ELU. La función sigmoide se ha utilizado en la neurona final dado que nuestro modelo es un clasificador binario. En la Ecuación 4.2 se representa la expresión matemática de esta función, que tiene una salida entre cero y uno, con saturación a partir de -4 y 4 respectivamente como se puede observar en la Figura 4.7.

$$\text{sigmoide}(x) = \frac{1}{1 + e^{-x}} \quad (4.2)$$

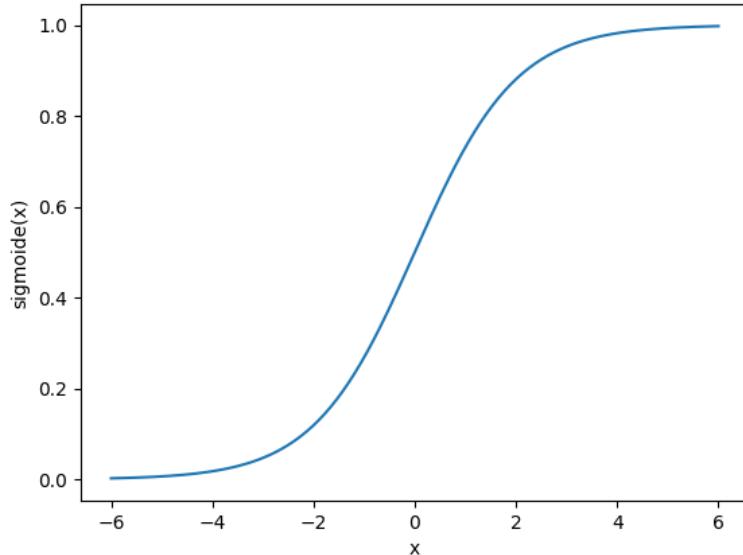


Figura 4.7: Representación gráfica de la función sigmoide.

En el caso del resto de la red se ha empleado la función ELU tras compararla con la función ReLu. La función ReLu únicamente considera los casos positivos como se refleja en la Ecuación 4.3, esto puede ser un problema dado que parte de la información no se está considerando. Esto lleva a que haya neuronas que siempre tengan como salida cero, por lo tanto se pueden considerar como neuronas muertas. En la Figura 4.8 se puede ver esto de forma más clara.

$$\text{ReLU}(x) = \begin{cases} x, & \text{si } x \geq 0 \\ 0, & \text{si } x < 0 \end{cases} \quad (4.3)$$

La función ELU, por su parte, ha demostrado obtener el mejor rendimiento, además de reducir el tiempo de entrenamiento dado que converge más rápido. La principal diferencia con respecto a la función ReLu es que esta no ignora los valores negativos. La expresión de esta función se puede observar en la Ecuación 4.4, donde α es un parámetro que marca el valor al cual tiende la función cuando la entrada es negativa. En este caso se ha utilizado el valor por defecto, 1. Esta función se representa en la Figura 4.9.

$$\text{ELU}(x) = \begin{cases} x, & \text{si } x \geq 0 \\ \alpha \cdot (e^x - 1), & \text{si } x < 0 \end{cases} \quad (4.4)$$

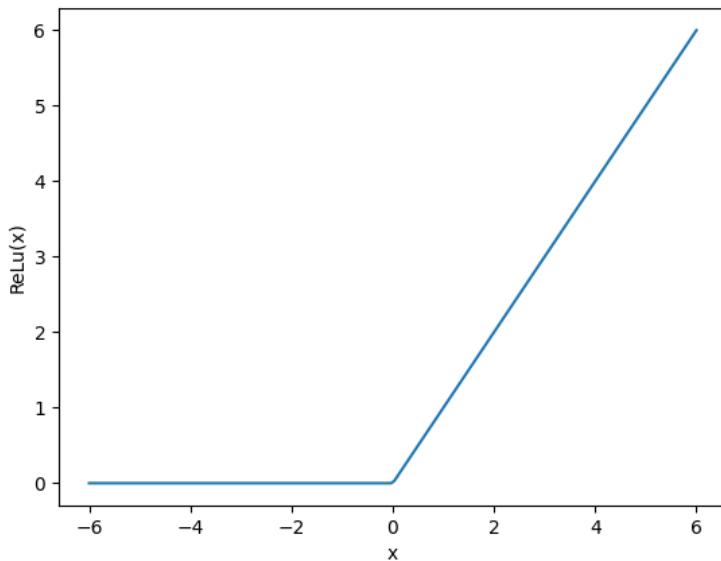


Figura 4.8: Representación gráfica de la función ReLU.

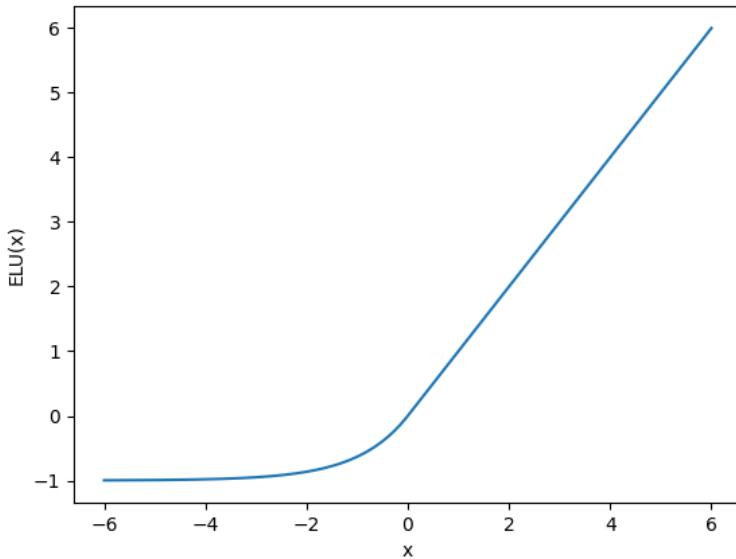


Figura 4.9: Representación gráfica de la función ELU.

Para inicializar los pesos se ha utilizado la inicialización de Xavier, que inicializa los pesos de una neurona según una distribución uniforme en este caso (para evitar tener una media nula), en función del número de neuronas conectadas a la entrada y a la salida de dicha neurona. Cuando esta inicialización se utiliza con la función de activación ReLU o una de sus variantes (incluyendo la ELU) se denomina inicialización He.

4.2.7 Regularización

Para evitar tener un problema de sobreajuste se han implementado distintas estrategias de regularización. La primera opción ha sido utilizar la estrategia de abandono

(*Dropout*). Esta técnica inutiliza algunas neuronas de manera aleatoria durante cada iteración, de tal forma que hace la red menos propensa a depender excesivamente de una parte. La cantidad de neuronas afectadas se puede cambiar en función de las necesidades de cada modelo. Para incluir esta capa en un modelo de TensorFlow simplemente se añade un objeto de la clase *Dropout* del módulo *tensorflow.keras.layers*. Esta técnica ha ayudado a disminuir ligeramente el sobreajuste durante algunos entrenamientos.

Otra estrategia utilizada es la normalización de lote (*Batch normalization*). Esta capa aplica una transformación que mantiene la media de la salida de una neurona cercana a cero con una desviación típica cercana a uno. Durante el entrenamiento esta capa normaliza la salida del lote actual. Por otro lado, durante la inferencia o uso normal de esta capa, normaliza utilizando una media móvil de la media y la desviación típica de los lotes utilizados durante el entrenamiento. Este recurso disminuye el sobreajuste y mejora el resultado global de la Red Neuronal. Para añadir esta capa al modelo basta con crear un objeto *BatchNormalization* del módulo *tensorflow.keras.layers*. Tras realizar comparativas entre este método y el abandono, se ha decidido continuar los últimos entrenamientos con la normalización de lote dado que se logran mejores resultados.

Por otro lado, también se han utilizado técnicas de regularización como restringir los pesos de cada neurona a tener norma unitaria. Esto se puede conseguir fácilmente utilizando la clase *unit_norm* del paquete *tensorflow.keras.constraints*. Finalmente, también se han probado los reguladores *l1* y *l2*, estos reguladores penalizan tener pesos de valor alto, de forma lineal en el caso de *l1* y cuadrática en el caso de *l2*. No obstante, esto solamente ha empeorado el rendimiento del modelo, por lo que no se ha seguido utilizando.

4.2.8 Capas del modelo

La parte más importante, y la que ha tomado más tiempo, ha sido averiguar la mejor estructura de capas. En este caso se ha trabajado principalmente con capas convolucionales bidimensionales (clase *Conv2D*) capas de neuronas totalmente conectadas (clase *Dense*) y capas de agrupación bidimensionales (clase *MaxPooling2D*) para reducir las dimensiones de una salida, además de las capas de abandono (clase *Dropout*) y normalización de lote (clase *BatchNormalization*) mencionadas anteriormente. Todas estas clases pertenecen al módulo *tensorflow.keras.layers* de TensorFlow.

Con respecto a las capas convolucionales, las más importantes en este caso, se ha variado su número para estudiar el efecto de la profundidad de la red. Se ha comprobado que al aumentar la profundidad de la red el modelo tiene a caer en el problema del sobreajuste. Por este motivo, se han empleado cuatro capas de convolución en el modelo final. Otro parámetro estudiado ha sido el tamaño de los filtros, considerando tamaños desde 1x1 hasta 11x11. Donde se ha comprobado que los filtros con tamaño mayor que 7x7 no mejoran el rendimiento de la red, y que a medida que la dimensión de la imagen disminuye hay que utilizar filtros más pequeños. Debido a esto se han utilizado filtros de tamaño 7x7, 5x5 y 3x3. La zancada también se ha estudiado, comprobando que la zancada unitaria en ambas direcciones es la que ofrece un mejor resultado.

Por otro lado, se han utilizado en algunos modelos capas de convolución anidadas, sin utilizar una capa de agrupación entre ellas. Esto ha mejorado el resultado en muchos casos. Sin embargo, añade complejidad, lo cual puede llevar a un problema de so-

breajuste e incrementar tiempo de computación. También, se ha variado el número de filtros en cada capa, utilizando valores entre 4 y 64. A medida que se incrementa el número de filtros en una capa se logran mejores resultados. No obstante, esto incrementa el tiempo de ejecución. Por ello, finalmente se han utilizado capas con 8 y 12 filtros. Otro aspecto estudiado es la influencia de utilizar agrupaciones de distinto tamaño, donde una combinación de agrupaciones de 9 elementos (3×3) y de 4 elementos (2×2) es la que ofrece los mejores resultados.

Finalmente, se ha estudiado el número de capas totalmente conectadas, así como su número de neuronas. Se ha comprobado que, a partir de tres capas, añadir más capas de este tipo no mejora el rendimiento de la red. Además de que incrementan notablemente el número de parámetros a entrenar. Por ello se han utilizado tres capas, la primera de 200 neuronas, la segunda de 16 y la última, la de salida, con únicamente una neurona. En la Figura 4.10 se puede observar un esquema del modelo final empleado.

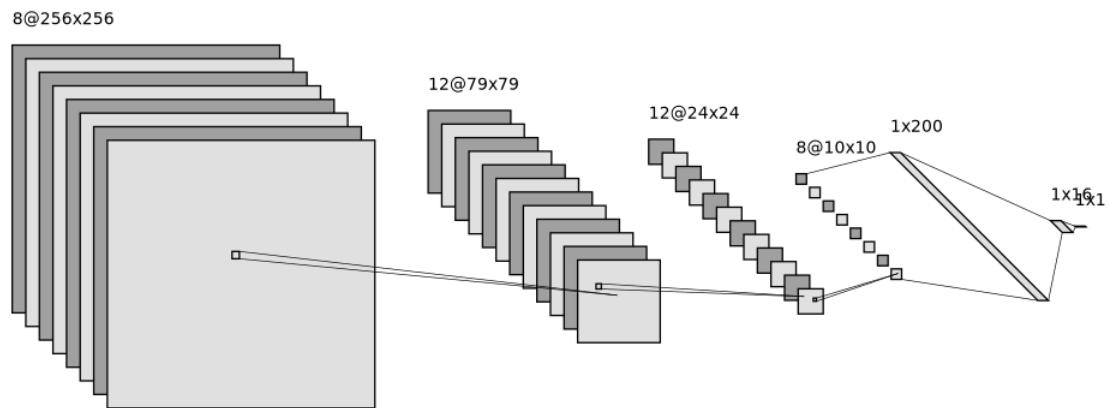


Figura 4.10: Esquema del modelo final.

4.3 Implementación en Raspberry Pi

A la hora de implementar el modelo en la Raspberry Pi el primer paso ha sido instalar el sistema operativo, concretamente la versión del 12 de diciembre del año 2020 de Raspberry Pi OS. El segundo paso ha sido instalar Python 3, así como las librerías requeridas, en este caso los módulos `tflite_runtime`, `NumPy`, `PIL`, `os` y `time`. El siguiente paso ha sido convertir el modelo final a un archivo `.tflite`. Esto se ha realizado utilizando la función `convert` de la clase `TFLiteConverter` del módulo `tensorflow.lite`. Este archivo `.tflite`, junto con las imágenes del conjunto de validación, se han transferido a la Raspberry Pi.

Finalmente, se ha escrito un programa en Python para ejecutar dicho modelo en las imágenes del conjunto de validación, así como para medir el rendimiento de la red. Los resultados de estas medidas se detallan en el capítulo de resultados. Para cargar y

utilizar el modelo se ha empleado la clase *Interpreter* del módulo *tflite_runtime*. En el caso de las imágenes se han abierto mediante el módulo PIL. Tras ello, se han convertido a un vector NumPy para su procesamiento. El procesamiento consiste en dividir los valores de los píxeles de la imagen entre 255, tal y como se ha realizado a la hora de entrenar el modelo, así como redimensionar dicha imagen para convertirla en una tupla de la forma (256, 256, 1).

En el siguiente capítulo se explican los resultados obtenidos a lo largo del desarrollo de este trabajo.

Capítulo 5

Resultados

En este capítulo se exponen y explican todos los resultados obtenidos a lo largo del desarrollo de este trabajo. En el primer punto se exponen los resultados intermedios, donde se explica a su vez las diferencias entre un modelo y otro. En el segundo punto se muestran los resultados finales, incluyendo el rendimiento del modelo en la Raspberry Pi.

5.1 Resultados intermedios

Modelo 1

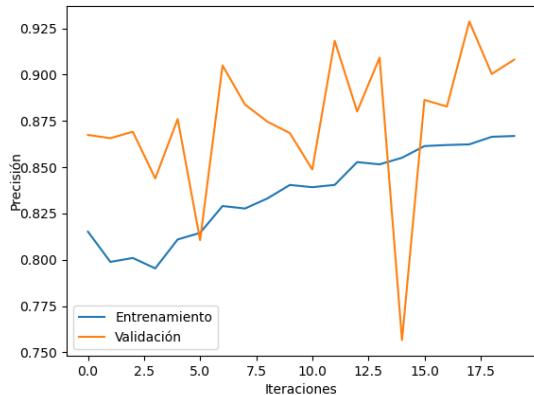
El primer modelo entrenado ha sido con una resolución de 300x300, utilizando el optimizador RMSprop con un ratio de aprendizaje de 10^{-3} . Respecto a los datos, se han utilizado 48.000 imágenes en el conjunto de entrenamiento y 12.000 en el de validación, utilizando por lo tanto una proporción 80-20. Otro aspecto importante relacionado con los datos es que en este caso se han utilizado técnicas de aumento de datos que se detallan a continuación:

- Rotación entre 0 y 15 grados.
- Desplazamiento vertical y horizontal entre un 0 y un 5% del tamaño de la imagen.
- Giro a lo largo del eje horizontal en sentido contrario a las agujas del reloj, simulando una captura desde otra perspectiva, desde 0 hasta 5 grados.
- Agrandar la imagen entre un 0 y un 10%.
- Voltear las imágenes vertical y/o horizontalmente.

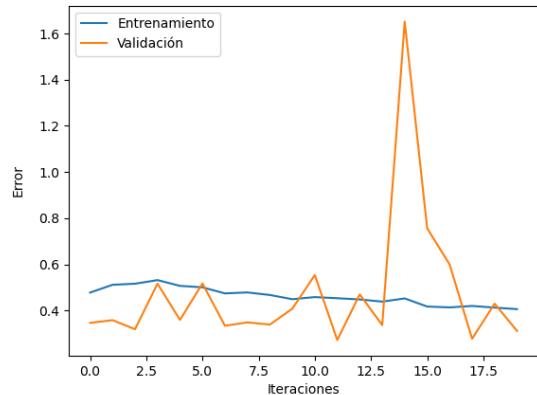
La topología utilizada ha sido una primera capa de convolución con 32 filtros de tamaño 3x3, una segunda convolución con 64 filtros y tamaño 3x3, cada una seguida de una capa de agrupación de tamaño 2x2, una capa de abandono del 20% y, por último, tres capas totalmente conectadas de 64, 16 y 1 neurona. La función de activación utilizada ha sido la función ReLu. Este primer modelo se ha entrenado durante 20 iteraciones.

Los resultados de este primer modelo se muestran en la Figura 5.1. La precisión aumenta de manera constante en el conjunto de entrenamiento, manteniéndose entre

un 80 y un 85%. y oscila en el de validación. En cuanto al error, desciende ligeramente aunque tiene un valor alto.



(a) Precisión.



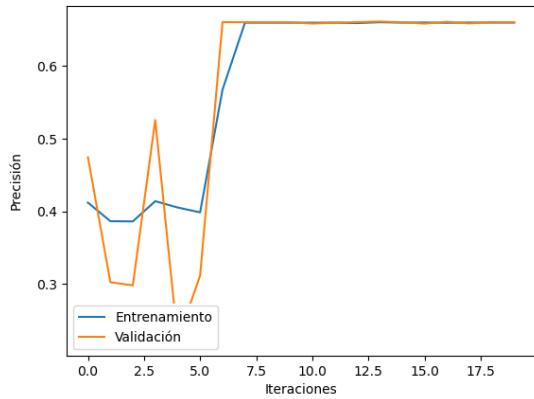
(b) Error.

Figura 5.1: Resultados del modelo 1.

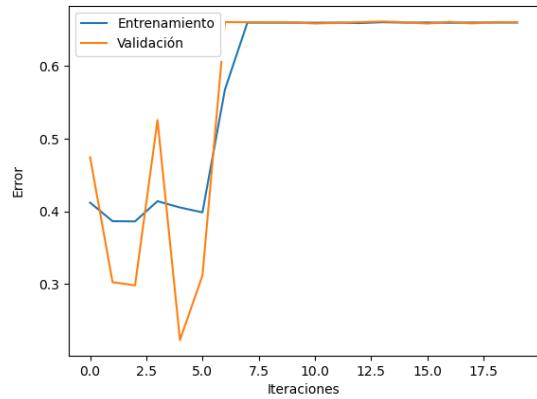
Modelo 2

El segundo modelo entrenado es muy similar al primero, únicamente se añade una tercera capa de convolución de 64 filtros de tamaño 3x3 y una capa completamente conectada de 4 neuronas antes de la última neurona.

En la Figura 5.2 se pueden ver los resultados de este modelo. Como se puede observar, tanto la precisión como el error se mantienen constantes en un valor aproximado de 0.68. Este comportamiento indica que la red se encuentra en un mínimo local y no es capaz de aprender.



(a) Precisión.



(b) Error.

Figura 5.2: Resultados del modelo 2.

Modelo 3

En el tercer modelo se cambia el número de filtros de en la primera capa de convolución, pasando de 32 a 16. Así como el número de neuronas de la primera capa totalmente conectada, pasando de 64 a 128. Por otro lado, se elimina la tercera capa de convolución añadida de en el segundo modelo. Finalmente, se elimina la capa de abandono entre las capas de convolución y las totalmente conectadas, y se añade una tras cada capa de agrupación.

En la Figura 5.3 se representan los resultados de este modelo. La precisión se mantiene alrededor de un 85% en el conjunto de entrenamiento y oscila entre un 80 y un 90% en el de validación. El error, por otro lado, se mantiene alrededor de 0.4 en el conjunto de entrenamiento y oscila de forma abrupta en el de validación. Esto indica que la red no es capaz de aprender de forma continuada, sino que oscila alrededor de un mínimo.

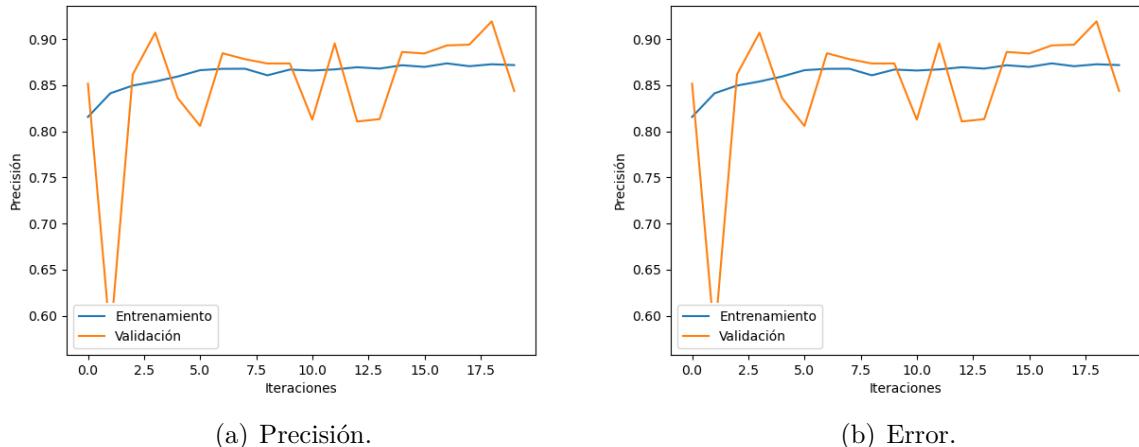


Figura 5.3: Resultados del modelo 3

Modelo 4

En este modelo se han realizado diversos cambios. El primero ha sido cambiar la resolución de las imágenes a 640x360. Otro cambio significativo ha sido utilizar menos datos, 29.600 para el entrenamiento y 7.400 para la validación. Por otro lado, se ha utilizado el optimizador Adam con un ratio de aprendizaje de $3 \cdot 10^{-3}$. La topología de la red se ha modificado drásticamente, empleando seis capas de convolución, cada una seguida de una de agrupación y una de abandono de un 20%. Todas las capas de convolución emplean en este caso filtros de tamaño 2x2, variando el número de filtros desde 32 en las capas iniciales hasta 12 en las últimas capas. Finalmente, se ha modificado el número de neuronas de las capas totalmente conectadas, pasando a ser de 432, 64, 8 y 1 respectivamente.

Como se puede ver en la Figura 5.4, esta complejidad de la red ha llevado a un problema de sobreajuste, dado que la precisión en el conjunto de validación es muy inferior a la del conjunto de entrenamiento. En el caso del error se observa una situación análoga, con un error de validación muy superior al de entrenamiento. Cabe destacar que incluso con un problema de sobreajuste, la precisión de entrenamiento no supera el 91%.

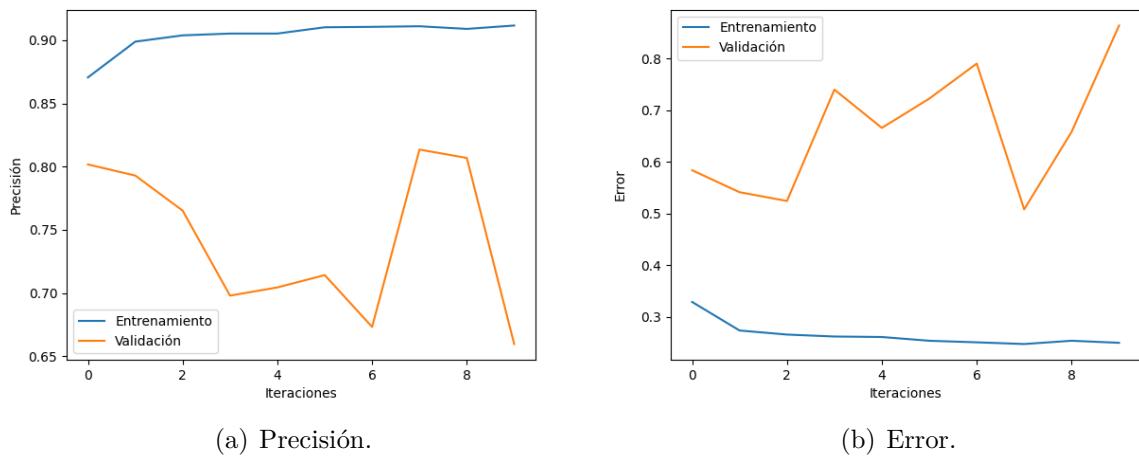


Figura 5.4: Resultados del modelo 4.

Modelo 5

Tras entrenar varios modelos similares al anterior, obteniendo resultados muy similares, se ha entrenado un modelo añadiendo más capas de convolución para estudiar su efecto. Concretamente, se han empleado once capas de convolución, todas con un tamaño de filtro de 2x2. El número de filtros ha ido variando desde 16 hasta 4. Se ha añadido una capa de agrupación de tamaño 2x2 y una capa de abandono del 20% tras cada capa de convolución. En cuanto a las capas totalmente conectadas, se ha cambiado el número de neuronas pasando a ser 48, 24, 12 y 1. Por último, el número de datos también ha variado, utilizando 40.000 imágenes para entrenar y 10.000 en la validación. Este modelo se ha entrenado durante 10 iteraciones.

Los resultados (ver Figura 5.5) en este caso muestran un comportamiento distinto. La precisión de ambos conjuntos se incrementa ligeramente a lo largo del entrenamiento, así como el error disminuye. Sin embargo, la diferencia entre ambos conjuntos sigue siendo destacada. Además, el rendimiento sigue sin ser mayor que los modelos anteriores.

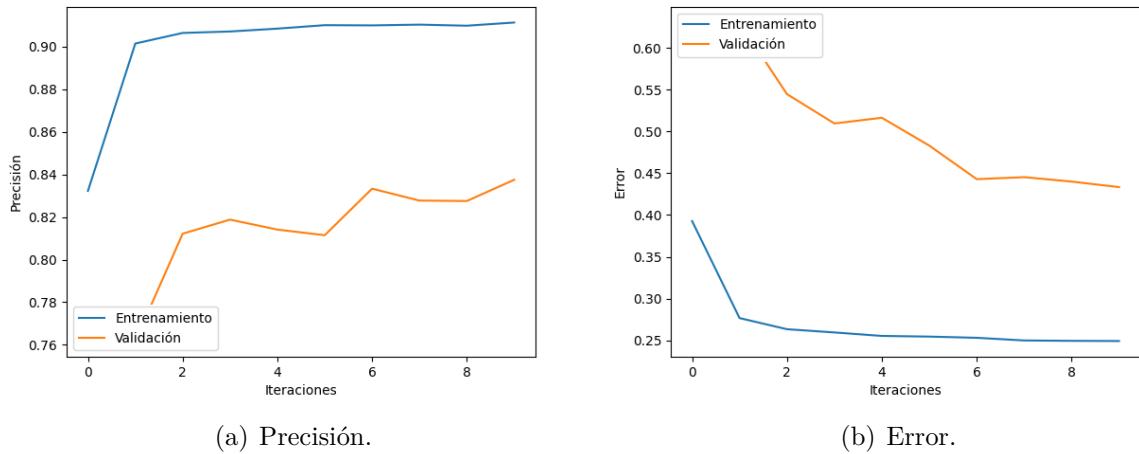


Figura 5.5: Resultados del modelo 5.

Modelo 6

En este modelo se ha cambiado el tamaño de la imagen, pasando a ser de 480x480. El ratio de aprendizaje se ha cambiado a un valor de $5 \cdot 10^{-3}$. En cuanto a la red se ha reducido el número de capas de convolución a siete, reduciendo por lo tanto la complejidad de la red. También, se han eliminado las capas de abandono para estudiar su impacto en el rendimiento. Por último, se han utilizado únicamente 3.840 imágenes para entrenar y 960 para la validación. Este modelo se ha entrenado durante 40 iteraciones.

Los resultados de este modelo (ver Figura 5.6) muestran una clara mejora con respecto al modelo anterior, ya que no existe una diferencia destacada entre la precisión de entrenamiento y la de validación. Por otro lado, el error de validación oscila entre 0.35 y 0.25, unos valores inferiores a los modelos anteriores. No obstante, estos resultados siguen sin ser satisfactorios, dado que parece que la red no es capaz de incrementar la precisión.

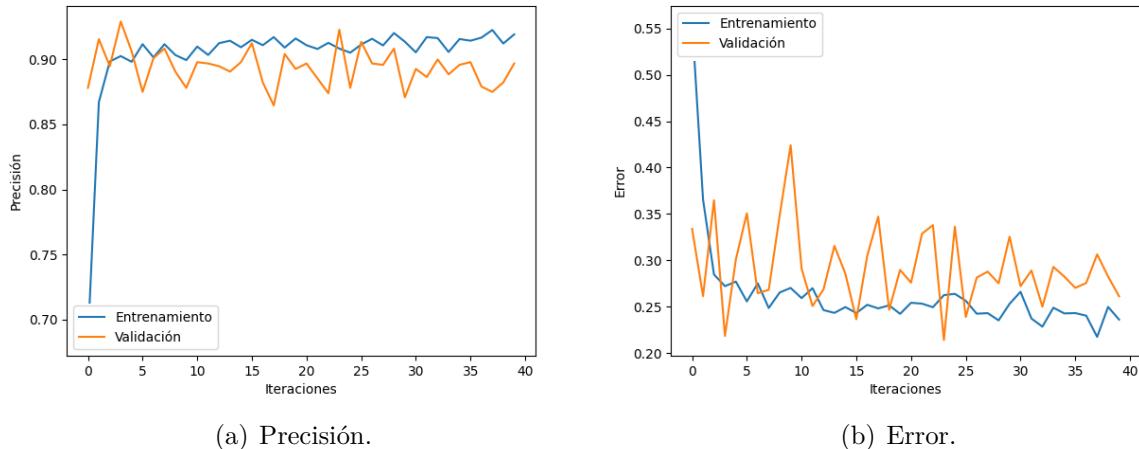


Figura 5.6: Resultados del modelo 6.

Modelo 7

Tras entrenar diversos modelos parecidos al anterior cuyo resultado no ha sido satisfactorio, se ha llegado a este modelo donde se introducen diversos cambios. El cambio más destacado en este caso ha sido utilizar la función de inicialización de Xavier o inicialización He en lugar de la inicialización por defecto. Por otro lado, se ha modificado ligeramente el valor del ratio de aprendizaje, pasando de $5 \cdot 10^{-3}$ a $4 \cdot 10^{-3}$, del mismo modo que la topología de la red, incrementando el tamaño de los filtros de 2x2 a 3x3 en las tres primeras capas de convolución y eliminando la última convolución, quedando así seis capas de convolución. Por último, se han utilizado más datos. 40.000 imágenes para el entrenamiento y 10.000 para la validación. Este modelo se ha entrenado durante 100 iteraciones.

Los resultados del modelo (ver Figura 5.7) indican una mejora en la precisión, rondando el 92.5% en el conjunto de entrenamiento y oscilando entre un 91 y un 87% en el de validación. Sin embargo, el error en la validación oscila de forma abrupta entre 0.4 y 0.25.

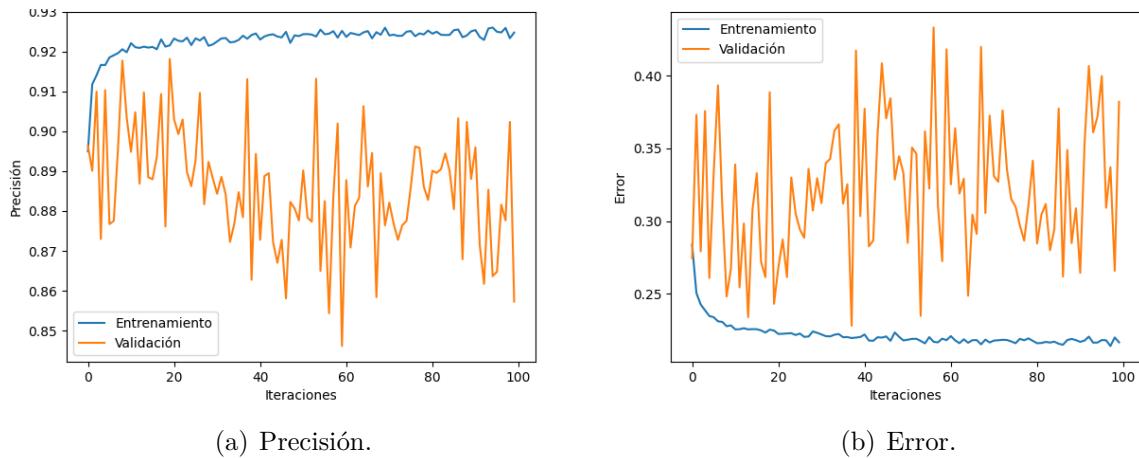


Figura 5.7: Resultados del modelo 7.

Modelo 8

En este modelo se ha decidido incrementar el tamaño y número de filtros, intentando maximizar la mejora de rendimiento lograda en el modelo anterior. Para ello, se han utilizado filtros de tamaño 7×7 en la primera capa y de tamaño 3×3 en las siguientes. También, se ha añadido una capa de convolución. En este caso el ratio de aprendizaje ha pasado a ser de $5 \cdot 10^{-4}$. Respecto a los datos, en este modelo no se han utilizado técnicas de aumento de datos. Por último, se ha modificado el número de capas completamente conectadas pasando a ser tres, de 24, 8 y 1 neurona respectivamente, entre las cuales se han añadido dos capas de abandono de un 25%. Este modelo se ha entrenado durante 150 iteraciones.

Los resultados de este modelo, como se puede observar en la Figura 5.8, indican un claro problema de sobreajuste. En este caso, la precisión en el conjunto de entrenamiento aumenta a lo largo de las 150 iteraciones, alcanzando un valor de un 95%. En el caso de la validación se observa de forma clara que la precisión disminuye. En el caso del error la situación es análoga.

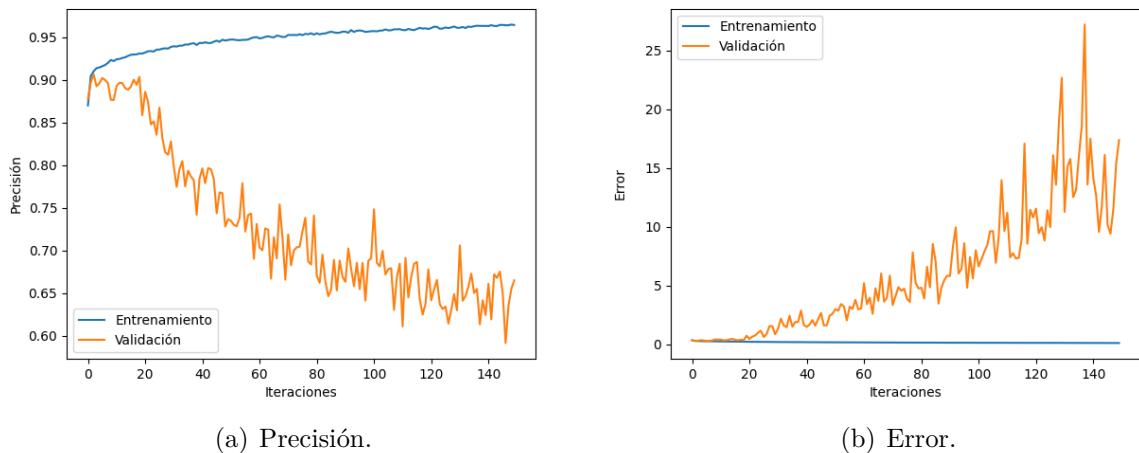


Figura 5.8: Resultados del modelo 8.

Modelo 9

Los cambios en este caso respecto al modelo anterior han sido utilizar menos capas de convolución, cuatro frente a siete, y el aumento del tamaño de los filtros. En este caso se ha utilizado un filtro de tamaño 11x11 en la primera capa de convolución, 7x7 en la segunda, 5x5 en la tercera y 3x3 en la cuarta. Otro cambio ha sido el uso de una capa de abandono de un 25% tras cada capa de agrupación. El número de neuronas de las capas totalmente conectadas ha pasado a ser 480, 16 y 1.

Como se observa en la Figura 5.9 la precisión en el conjunto de entrenamiento alcanza un valor muy alto, alrededor del 99%, y el error a su vez tiene un valor muy bajo. No obstante, en la validación los resultados muestran un claro caso de sobreajuste, dado que la precisión se mantiene entre el 83 y el 85% en la mayoría de los casos. Si bien la precisión no baja como en el caso anterior, la diferencia es muy grande entre entrenamiento y validación. Por ello, se ha decidido continuar con los entrenamientos hasta encontrar una solución mejor.

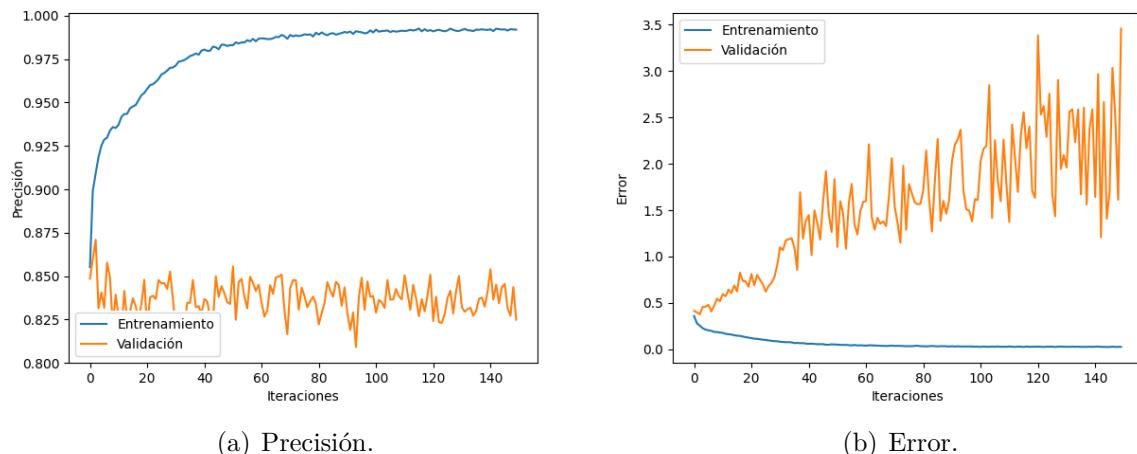


Figura 5.9: Resultados del modelo 9.

Modelo 10

Se han entrenado varios modelos similares al anterior, variando el tamaño y el número de filtros, así como las neuronas de las últimas capas. Sin embargo, estos cambios han conducido a resultados muy parecidos al modelo 9. En este caso se han utilizado todos los datos disponibles con una proporción 70-30. En consecuencia, se han utilizado 62.500 imágenes para el entrenamiento y 26.800 para la validación, empleando un tamaño de 432x432. Por otro lado, se han empleado catorce capas de convolución, donde algunas se han anidado. El tamaño de los filtros se ha variado desde 7x7 en las primeras dos capas hasta 3x3 en las últimas seis. En cuanto a las capas totalmente conectadas, se han utilizado 864, 16 y 1 neurona. Por último, se ha utilizado la función de activación ELU en todas las capas. Este modelo se ha entrenado un total de 15 iteraciones.

Los resultados de este entrenamiento (ver Figura 5.10) indican una mejora respecto al modelo anterior, dado que en esta ocasión la diferencia entre el conjunto de entrenamiento y el de validación ha disminuido. A pesar de esto los resultados siguen sin ser buenos, dado que el error es considerable y la red es muy compleja, lo que imposibilita su uso en un sistema embebido.

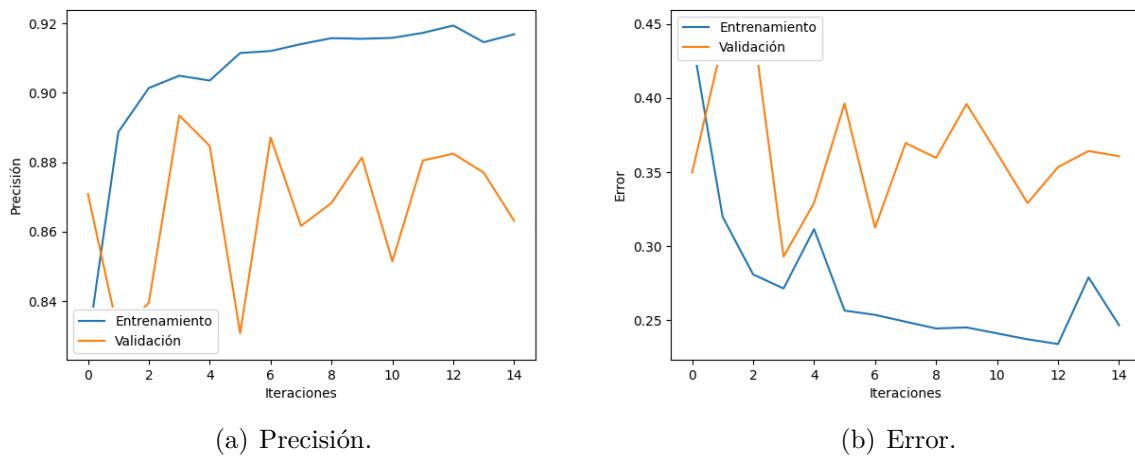


Figura 5.10: Resultados del modelo 10.

Modelo 11

La única diferencia entre este modelo y el anterior es la función de activación, que en este caso es la función ReLu. Como se puede observar en la figura 5.11, este modelo logra un peor rendimiento que el anterior. La precisión, tanto en el entrenamiento como en la validación es más baja. El error en este caso oscila de forma más abrupta y tiene un valor mayor. De esto se puede concluir que la función de activación ELU es mejor en este problema.

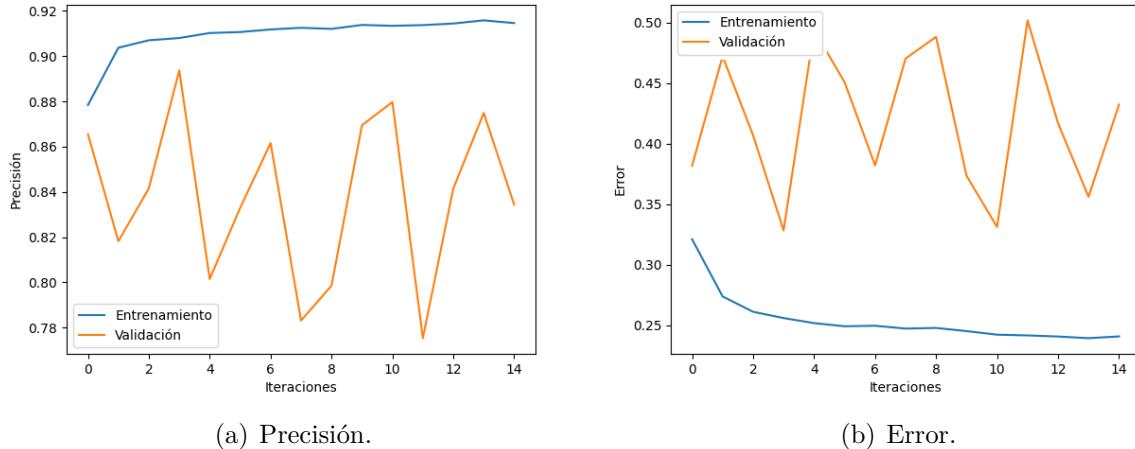
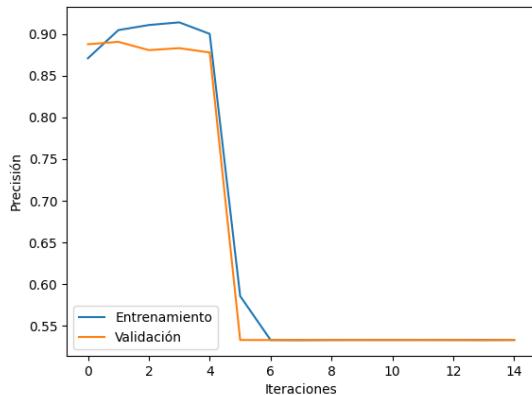


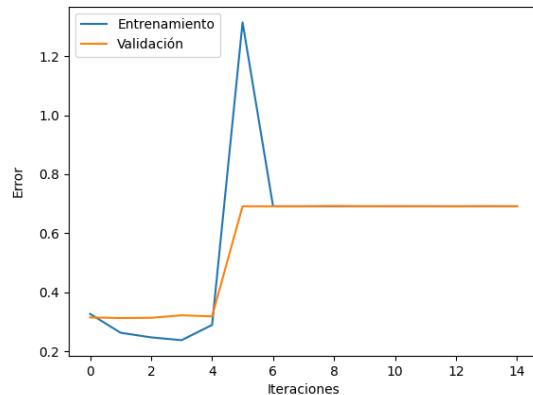
Figura 5.11: Resultados del modelo 11.

Comparación de porcentaje de abandono

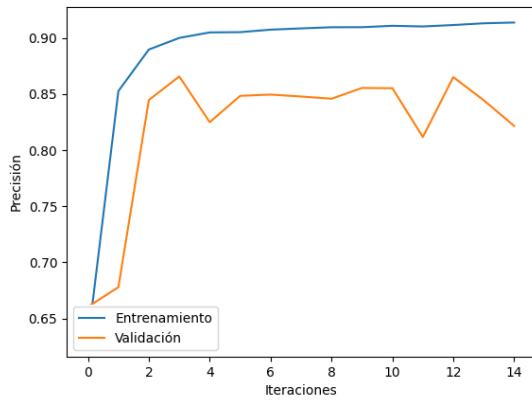
Con el objetivo de estudiar el impacto de las capas de abandono de forma más clara, se ha entrenado el modelo 10 variando el porcentaje de abandono. Originalmente se ha entrenado con una tasa de abandono del 25%. Se ha entrenado además con una tasa del 0 y del 50%. En la Figura 5.12 se pueden observar los resultados con estas últimas tasas. El error es mayor al utilizar una tasa de un 50% frente a utilizar una del 25%. En el caso de no utilizar capas de abandono el modelo no sale de un mínimo local, manteniendo un error muy alto. Tras esta comparación se concluye que una tasa de abandono del 25% es la ideal.



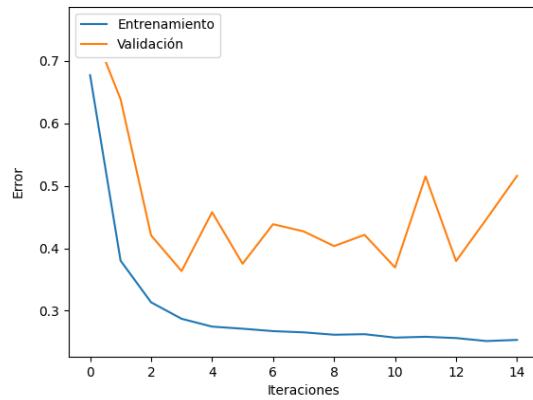
(a) Precisión con tasa de abandono del 0%.



(b) Error con tasa de abandono del 0%.



(c) Precisión con tasa de abandono del 50%.



(d) Error con tasa de abandono del 50%.

Figura 5.12: Resultados del modelo 10 con distintas tasas de abandono.

Modelo 12

En este modelo se ha empleado la normalización por lotes en lugar de las capas de abandono como método de regularización. Por otro lado, se han utilizado doce capas de convolución, utilizando una capa de agrupación y otra de normalización por lotes tras cada dos capas de convolución. Este modelo se ha entrenado durante 15 iteraciones.

Los resultados de este modelo (ver Figura 5.13) indican un mejor rendimiento que el modelo 10. En este caso, la precisión de entrenamiento supera el 94% y la de validación

oscila entre el 84 y el 90%. En cuanto al error, se observa que en la validación el error aumenta, por lo tanto se tiene un problema de sobreajuste.

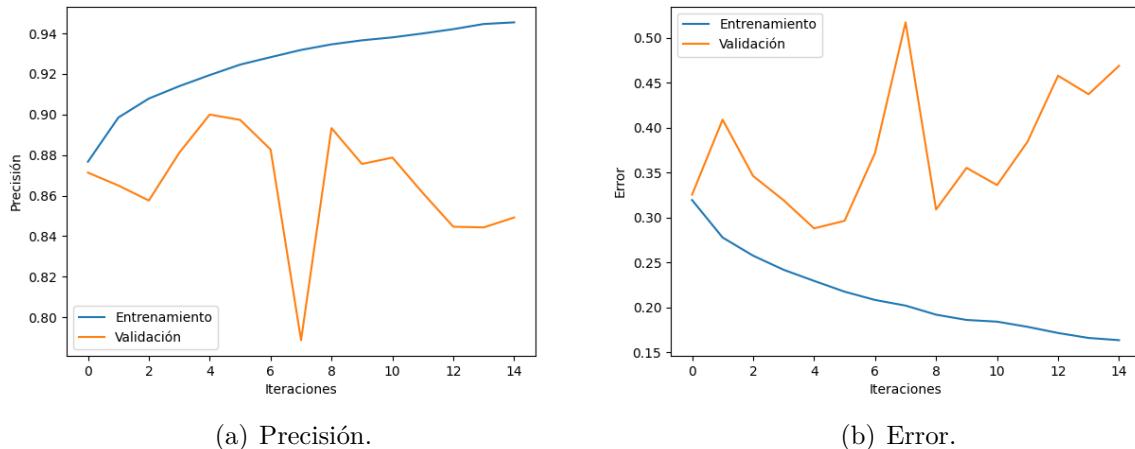


Figura 5.13: Resultados del modelo 12.

Comparación de técnicas de regularización

Dado el problema del sobreajuste en el modelo 12 se ha decidido emplear distintas técnicas de regularización, además de la normalización por lotes, con el objetivo de encontrar una solución. La primera técnica utilizada ha sido la regularización l_1 y l_2 , para penalizar los pesos grandes. En las Figuras 5.14(a) y 5.14(b) se puede notar que esta técnica disminuye drásticamente la precisión en el conjunto de validación, incrementando a su vez el error. De esta manera, podemos descartar el uso de este tipo de regularizaciones.

Por otro lado, se ha empleado la restricción de pesos a la norma unidad para cada capa del modelo. En este caso, como se puede observar en las Figuras 5.14(c) y 5.14(d), esto ha llevado a mejores resultados que en el caso anterior. A pesar de esto, los resultados son peores que en el modelo original.

Modelo 13

En este modelo se han utilizado menos capas de convolución con el objetivo de obtener un modelo más simple. Además, no se han utilizado capas de convolución anidadas, reduciendo la complejidad de la red. De esta manera, se han utilizado cinco capas de convolución de entre 4 y 16 filtros, con filtros de tamaño 7×7 y 5×5 en las primeras dos capas, 3×3 en las siguientes dos y 2×2 en la última. También, se ha empleado normalización por lote y restricción de peso a norma unidad. Por otro lado, se han utilizado dos tamaños de imagen, 300×300 y 256×256 , con el objetivo de evaluar su impacto en el rendimiento.

Como se observa en la Figura 5.15 el tamaño de la imagen no afecta al rendimiento. En cuanto a la precisión, cabe destacar que el modelo no es estanca en un mínimo local, logrando mejorar la precisión en el conjunto de entrenamiento tras cada iteración. No obstante, el modelo tiene problemas de sobreajuste, por lo que es necesario mejorarlo.

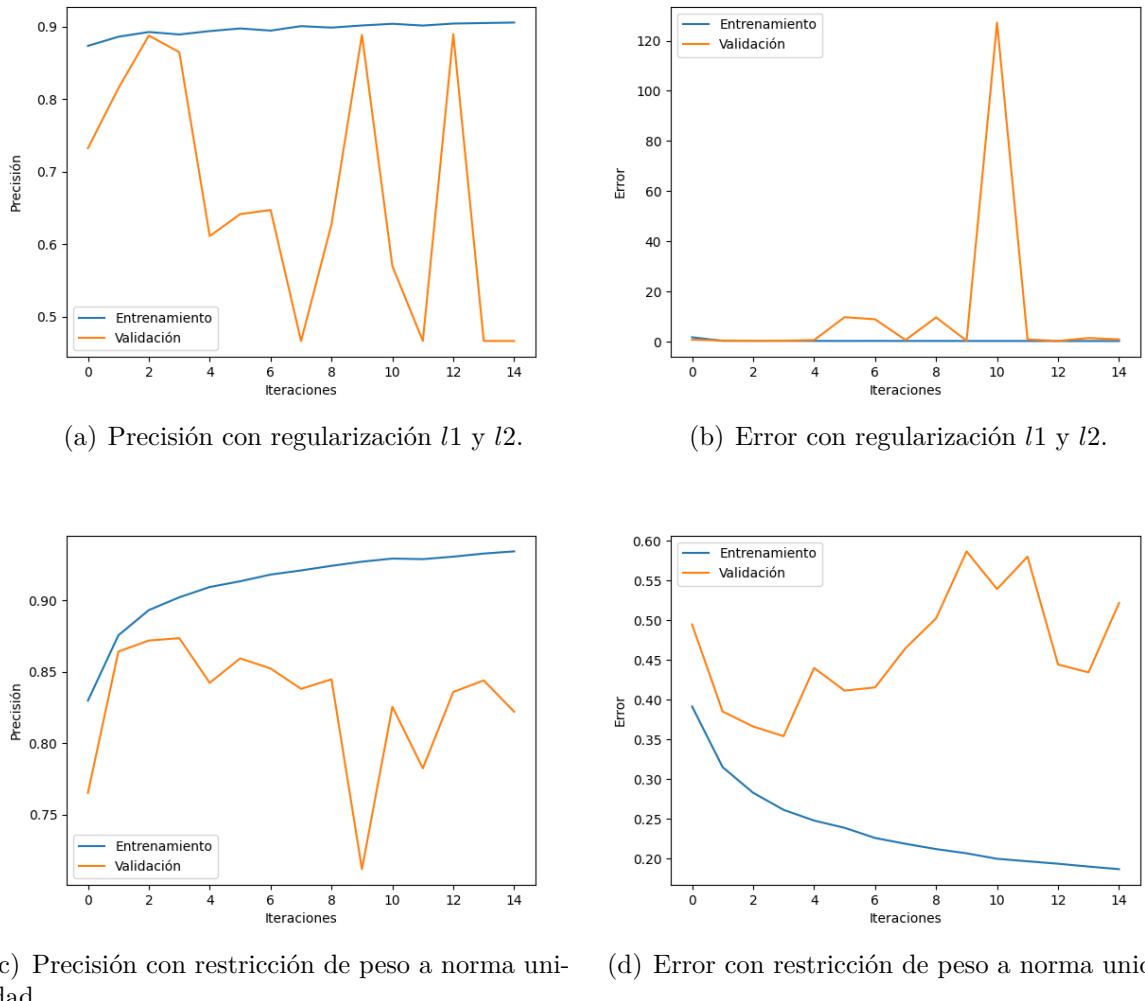


Figura 5.14: Resultados del modelo 12 con distintas técnicas de regularización.

Modelo 14

En este caso se ha simplificado el modelo con el objetivo de evitar el problema del sobreajuste. Además, se ha cambiado la proporción de datos a 85-15, utilizando de esta manera 75.900 imágenes para entrenar y 13.400 para la validación. Las neuronas en las capas totalmente conectadas son 200, 16 y 1. Este modelo se ha entrenado durante 75 iteraciones.

Los resultados de este modelo (ver Figura 5.16) mejoran claramente todos los anteriores. La precisión en el conjunto de entrenamiento ronda el 98% y en el de validación oscila entre un 87 y un 89%. La situación del error es análoga. A pesar de todo esto se sigue con el problema de sobreajuste, por lo que el siguiente paso es volver a utilizar técnicas de aumento de datos con el objetivo de evitarlo.

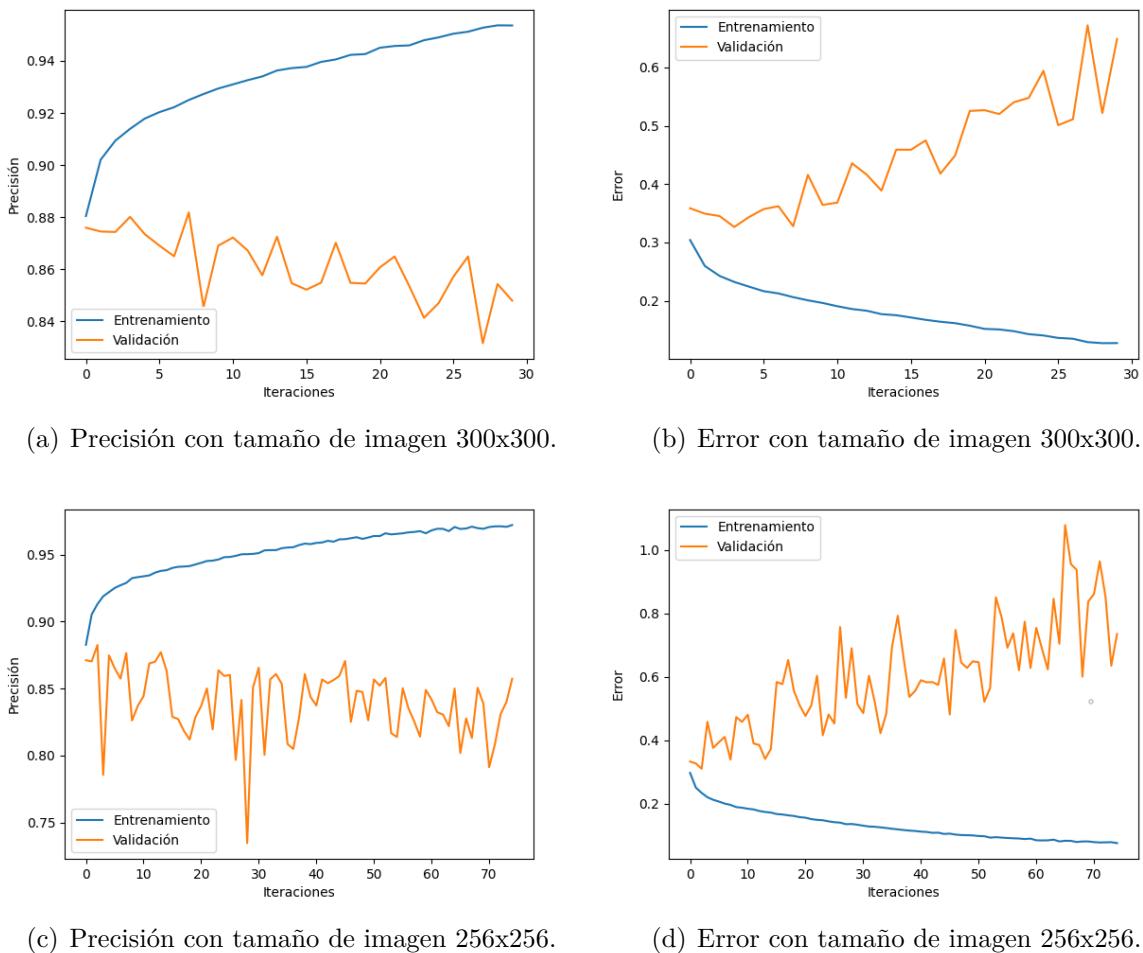


Figura 5.15: Resultados del modelo 13.

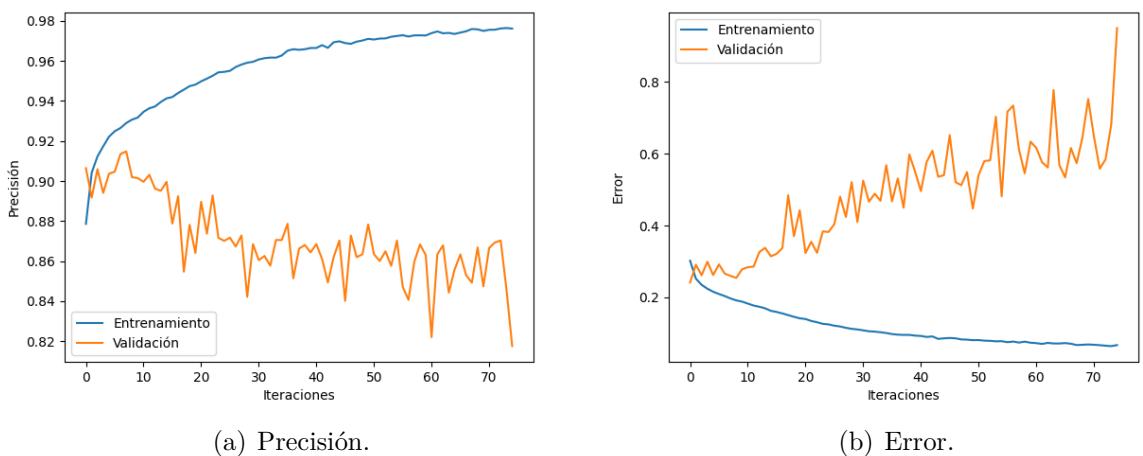


Figura 5.16: Resultados del modelo 14.

5.2 Resultados finales

La única diferencia entre este modelo y el anterior es el uso de técnicas de aumento de datos. Los valores para el aumento de datos son los mismos que los utilizados en el primer modelo. Este modelo se ha entrenado durante 200 iteraciones.

En la Figura 5.17 se puede apreciar que tanto la precisión como el error tienen valores parecidos en ambos conjuntos. Sin embargo, a partir de la iteración 35 aproximadamente se observa como el modelo comienza a caer en un problema de sobreajuste. Por ello, se utilizan los pesos en la iteración 35 como pesos finales del modelo. La matriz de confusión resultante para los conjuntos de validación y entrenamiento son las mostradas en las Tablas 5.1 y 5.2 respectivamente. A partir de estas matrices se han calculado los valores de la precisión, exhaustividad y valor F1 representados en la Tabla 5.3.

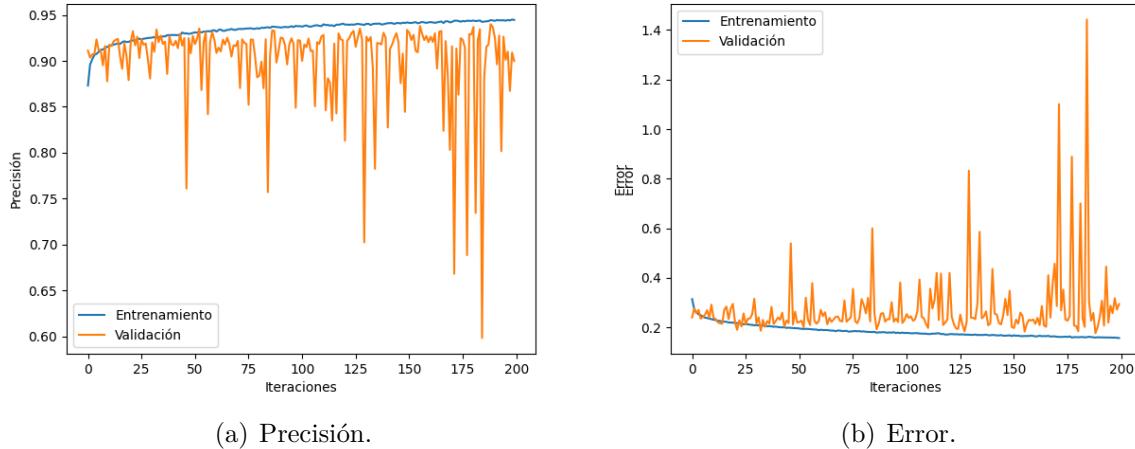


Figura 5.17: Resultados del modelo final.

	Clasificados como meteoros	Clasificados como no meteoros
Meteoros	5880	371
No meteoros	435	6704

Tabla 5.1: Matriz de confusión del modelo final en el conjunto de validación.

	Clasificados como meteoros	Clasificados como no meteoros
Meteoros	33673	1748
No meteoros	3546	36906

Tabla 5.2: Matriz de confusión del modelo final en el conjunto de entrenamiento.

	Precisión	Exhaustividad	Valor F1
Conjunto de entrenamiento	90,47%	95,07%	0,927
Conjunto de validación	93,11%	94,07%	0,936

Tabla 5.3: Métricas de rendimiento del modelo final.

Estos resultados se consideran muy buenos, especialmente si se tiene en cuenta el coste computacional del modelo. Este rendimiento supera el de algunos sistemas del estado del arte, como [11], obteniendo una precisión y exhaustividad más alta con menos datos y una red menos compleja. Esto se debe a que en dicho sistema se utilizan vídeos como datos de entrada, lo cual conlleva una mayor carga computacional.

Por otro lado, este modelo iguala el rendimiento de otros modelos del estado del arte como el método alternativo basado en cuatro etapas en [2] o el modelo creado a partir de aprendizaje de transferencia en [12].

Por último, se ha medido el rendimiento del modelo en la Raspberry Pi con el objetivo de asegurar que el modelo funciona correctamente. La matriz de confusión obtenida en la Raspberry Pi es la misma que la obtenida en el ordenador donde se ha entrenado, lo cual indica una correcta conversión e implementación del modelo. Respecto a los tiempos, el tiempo medio para leer, procesar y clasificar una imagen es de 56,03 ms. El tiempo medio empleado únicamente en clasificar una imagen es de 42,65 ms. Esto permitiría clasificar aproximadamente 17,86 imágenes por segundo. Estos tiempos se consideran excelentes, dado que una imagen comprime los datos de aproximadamente 10 segundos.

Capítulo 6

Conclusiones

6.1 Conclusiones

A nivel personal este trabajo ha servido como introducción a la Inteligencia Artificial, concretamente al Aprendizaje Profundo, así como al lenguaje Python y al ecosistema de Raspberry Pi.

El desarrollo de este trabajo, que se ha extendido durante diez meses, ha sido apasionante, logrando despertar un gran interés por mi parte en el campo Aprendizaje Profundo. Por todo ello, y teniendo como base los conocimientos aquí adquiridos, se va a seguir investigando y estudiando esta tecnología tras la finalización de este TFG, con el objetivo de dedicarse profesionalmente a la investigación, desarrollo y despliegue de sistemas basados en Aprendizaje Profundo.

En cuanto al trabajo, se concluye que las Redes Neuronales tienen un enorme potencial en tareas de reconocimiento de imágenes, debido a su gran rendimiento y flexibilidad. Además, mediante el uso de TensorFlow junto con Python es posible desarrollar y desplegar modelos de Aprendizaje Profundo de manera sencilla y rápida. Por otro lado, se ha manifestado la gran importancia de utilizar herramientas de control de versiones, en este caso GitHub, para monitorizar el progreso, además de documentar el proceso de desarrollo del código.

Por otro lado, el desarrollo de este trabajo ha resaltado la importancia de los datos a la hora de desarrollar modelos de Machine Learning. Esta importancia se puede dividir en dos aspectos principales: la cantidad y la calidad de los datos. La cantidad de los datos es muy importante a la hora de generalizar un modelo. Por otro lado, la calidad de dichos datos resulta crucial, pues de nada sirve disponer de una gran cantidad de datos si contienen errores o no son lo suficientemente variados.

El entrenamiento del modelo ha manifestado algunos aspectos importantes. El primero es la importancia de contar con una o varias GPUs, debido a que aceleran de forma notable el entrenamiento. Por otro lado, cabe mencionar que a medida que la red es más profunda, es decir, tiene más capas, tiende a caer en problemas de sobreajuste, consumir más tiempo para el entrenamiento y tardar más tiempo en ejecutarse, especialmente en sistemas con limitados recursos como los embebidos.

Respecto a los resultados obtenidos, se concluye que son excelentes dado que igua-

lan algunos modelos del estado del arte, desarrollados por investigadores experimentados. Estos resultados podrían mejorarse en caso de disponer de más datos, así como varias GPUs que permitan entrenar de forma más rápida. Finalmente, se concluye que se han logrado con creces todos los objetivos propuestos en el primer capítulo de este TFG.

6.2 Desarrollos futuros

Dada la gran flexibilidad de los modelos de Aprendizaje Automático y las grandes ventajas que ofrece el ecosistema de Raspberry Pi, existe un gran abanico de posibilidades para trabajos e investigaciones futuras. A continuación, se mencionan algunas de estas posibilidades.

1. Despliegue del modelo con el objetivo de medir el rendimiento del mismo en un entorno real, dado que las condiciones climáticas pueden afectar a los resultados. Por otro lado, esto sería conveniente para capturar imágenes y vídeos y, de esta manera, disponer de más datos para los próximos entrenamientos.
2. Mejora del modelo mediante el uso de nuevas técnicas y herramientas que surgirán en los próximos años debido al auge del Aprendizaje Profundo. Además, es posible desarrollar un modelo más ligero con un rendimiento similar al actual en caso de disponer de más datos.
3. Desarrollar un software para almacenar y enviar las detecciones de cada Raspberry Pi a un servidor. Esto facilitaría la realización de los dos puntos anteriores.
4. Añadir la posibilidad de actualizar el modelo desplegado en las Raspberry Pi de manera remota. Esto añadiría flexibilidad al sistema, permitiendo una mejora continua del mismo.
5. Desarrollo de un modelo con el mismo objetivo con otras librerías, por ejemplo PyTorch y comparar los resultados obtenidos. Así mismo, se podría desarrollar un modelo con la misma finalidad en otros lenguajes como C++, Java o MATLAB.

Bibliografía

- [1] NASA (2019). *Meteors and Meteorites*. [En línea]. Disponible en: [https://solarsystem.nasa.gov/asteroids-comets-and-meteors/meteors-and-meteorites/overview/](https://solarsystem.nasa.gov/asteroids-comets-and-meteors/meteors-and-meteorites/overview)
- [2] Novoselnik, F., Grbić, R. & Slišković, D. (2016). *Image Based Meteor Detection and Path Estimation*.
- [3] Zhang, D., Mishra, S., Brynjolfsson. E., Etchemendy, J., Ganguli D., Grosz, B., Lyons, T., Manyika, J., Niebles, J. C., Sellitto, M., Shoham, Y., Clark, J. & Perrault, R. (2021). *The AI Index 2021 Annual Report*. AI Index Steering Committee, Human-Centered AI Institute, Stanford University.
- [4] Vida, D. & Novoselnik, F. (2011). *Croatian Meteor Network: data reduction and analysis*. In Proceedings of the International Meteor Conference.
- [5] Gural, P. S. (2007). *Algorithms and Software for Meteor Detection*. In Advances in Meteoroid and Meteor Science.
- [6] Gural, P. S. & Šegon, D. (2009). *A new meteor detection processing approach for observations collected by the Croatian Meteor Network (CMN)*. Journal of the IMO, vol. 37, no. 1, pp. 28-32.
- [7] Trigo-Rodriguez, J. M., Madiedo, J. M., Gural, P. S., Castro-Tirado, A. J., Llorca, J., Fabregat, J., Vítek, S. & Pujols, P. (2008). *Determination of Meteoroid Orbits and Spatial Fluxes by Using High-Resolution All-Sky CCD Cameras*.
- [8] Davis, E. R. (1990). *Machine Vision: theory, algorithms, practicalities*. Academic Press (Caps. 8-14).
- [9] Illingworth, J. & Kittler, J. (1988). *A survey of the Hough Transform, Computer Vision Graphics and Image Processing*, vol. 44, pp. 87-116.
- [10] Bishop, C. M. (2006). *Pattern recognition and machine learning, Second*. Springer New York.
- [11] De Cicco, M. (2017). *Artificial intelligence techniques for automating the CAMS processing pipeline to direct the search of long-period comets*.
- [12] Galindo Y. & Lorena, A. C. (2018). *Deep Transfer Learning for Meteor Detection*.
- [13] <http://press.exoss.org>.
- [14] Loshchilov I. & Hutter, F. (2016). *SGDR: Stochastic Gradient Descent with restarts*.

- [15] Kingma D. P. & Ba, J. (2014). *Adam: A method for stochastic optimization.*
- [16] <https://pytorch.org>.
- [17] Gural, P. S. (2019). *Deep learning algorithms applied to the classification of video meteor detections*
- [18] Zoghbi, S. et al. (2017). *Workshop on Deep Learning for Physical Sciences (DLPS 2017).*
- [19] Cecil, D. & Campbell-Brown, M. (2020). *The application of convolutional neural networks to the automation of a meteor detection pipeline.*
- [20] Chollet, F. et al. (2015). *Keras.* <https://keras.io>,
- [21] Google, (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* <https://www.tensorflow.org/>.
- [22] Roth, W. et al. (2020). *Resource-Efficient Neural Networks for Embedded Systems.*
- [23] National Institute of Standards and Technology (NIST), USA. (2011). *The NIST Definition of Cloud Computing.* Technical Report.
- [24] Coutinho, E. F., De Carvalho Sousa, F. R., Rego, P. A. L., Gomes, D. G. & De Souza, J. N. (2015). *Elasticity in cloud computing: A survey.*
- [25] Naeski, M. (2019). *Bringing Machine Learning to Embedded Systems;* White Paper; Texas Instruments.
- [26] Chen, D. & Zhao, H. (2012). *Data security and privacy protection issues in cloud computing.*
- [27] Tari, Z. (2014). *Security and Privacy in Cloud computing.*
- [28] Lim, Q., Hristu-Varsakelis, D. & Levine, W. S. (2005). *Fundamentals of RTOS-Based Digital Controller Implementation.*
- [29] O'Donovan, P., Gallagher, C., Bruton, K. & O'Sullivan, D. T. (2018). *A fog computing industrial cyber-physical system for embedded low-latency machine learning Industry 4.0 applications.* Manufacturing Letters 15, pp. 139–142.
- [30] Teerapittayanon, S., McDanel, B. & Kung, H. T. (2017). *Distributed Deep Neural Networks over the Cloud, the Edge and End Devices.* In Proceedings of the International Conference on Distributed Computing Systems (ICDS), pp. 328–339.
- [31] Aazam, M., Zeadally, S. & Harras, K. A. (2018). *Deploying Fog Computing in Industrial Internet of Things and Industry 4.0.* In IEEE Transactions on Industrial Informatics, vol. 14, no. 10, pp. 4674-4682.
- [32] Xiao, S., Dhamdhere, A., Sivaraman, V. & Burdett, A. (2009). *Transmission Power Control in Body Area Sensor Networks for Healthcare Monitoring.* In IEEE Journal on Selected Areas in Communications, vol. 27, no. 1, pp. 37-48.
- [33] Kazemi, R., Vesilo, R., Dutkiewicz, E. & Liu, R. (2011). *Dynamic Power Control in Wireless Body Area Networks Using Reinforcement Learning with Approximation.* In Proceedings of the 2011 IEEE 22nd International Symposium on Personal, Indoor and Mobile Radio Communications, pp. 2203–2208.

- [34] Fischer, M., Scheerhorn, A. & Tönjes, R. (2019). *Using Attribute-Based Encryption on IoT Devices with instant Key Revocation*. In Proceedings of the 2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), pp. 126–131.
- [35] Khan, A. M., Umar, I. & Ha, P. H. (2018). *Efficient Compute at the Edge: Optimizing Energy Aware Data Structures for Emerging Edge Hardware*. In Proceedings of the 2018 International Conference on High Performance Computing Simulation (HPCS), pp. 314–321.
- [36] Shafique, M., Theocharides, T., Bouganis, C., Hanif, M. A., Khalid, F., Hafiz, R. & Rehman, S. (2018). *An overview of next-generation architectures for machine learning: Roadmap, opportunities and challenges in the IoT era*. In Proceedings of the 2018 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 827–832.
- [37] Yang, Y., Chen, A., Chen, X., Ji, J., Chen, Z. & Dai, Y. (2018). *Deploy Large-Scale Deep Neural Networks in Resource Constrained IoT Devices with Local Quantization Region*. arXiv:1805.09473.
- [38] Chauhan, J., Seneviratne, S., Hu, Y., Misra, A., Seneviratne, A. & Lee, Y. (2018). *Breathing-Based Authentication on Resource-Constrained IoT Devices using Recurrent Neural Networks*. Computer, vol. 51, no. 5, pp. 60–67.
- [39] Microchip. (2017). *Microchip Introduces the Industry's First MCU with Integrated 2D GPU and Integrated DDR2 Memory for Groundbreaking Graphics Capabilities*. [En línea]. Disponible en: <https://www.microchip.com>.
- [40] Arm. (2019). *Next-generation Armv8.1-M architecture: Delivering Enhanced Machine Learning and Signal Processing for the Smallest Embedded Devices*. [En línea]. Disponible en: <https://www.arm.com/company/news/2019/02/next-generation-armv8-1-m-architecture>.
- [41] Chirag, G. (2017). *ProtoNN: Compressed and Accurate kNN for Resource-scarce Devices*. In Proceedings of the 34th International Conference on Machine Learning, Sydney, Australia, 2017; Volume 70, pp. 1331–1340.
- [42] Kumar, A., Goyal, S. & Varma, M. (2017). *Resource-efficient Machine Learning in 2 KB RAM for the Internet of Things*. In Proceedings of the 34th International Conference on Machine Learning (ICML 2017), volume 70, pp. 1935–1944.
- [43] De Almeida Florencio, F., Moreno Ordóñez, E. D., Teixeira Macedo, H., Paiva De Britto Salgueiro, R. J., Barreto Do Nascimento, F. & Oliveira Santos, F. A. (2018). *Intrusion Detection via MLP Neural Network Using an Arduino Embedded System*. In Proceedings of the 2018 VIII Brazilian Symposium on Computing Systems Engineering (SBESC), pp. 190–195.
- [44] Szydlo, T., Sendorek, J. & Brzoza-woch, R. (2018). *Computational Science—ICCS 2018*; Springer International Publishing: Berlin/Heidelberg, Germany, 2018; Volume 10862, pp. 682–694.
- [45] Leech, C., Raykov, Y. P., Ozer, E. & Merrett, G. V. (2017). *Real-time room occupancy estimation with Bayesian machine learning using a single PIR sensor and*

- microcontroller.* In Proceedings of the 2017 IEEE Sensors Applications Symposium (SAS), pp. 1–6.
- [46] Halevy, A. Y., Norvig, P. & Pereira, F. (2009). *The Unreasonable Effectiveness of Data*. IEEE Intelligent Systems, 24, pp. 8-12.
- [47] Catal, C. (2012). *Performance Evaluation Metrics for Software Fault Prediction Studies*. Acta Polytechnica Hungarica, 9(4), pp. 193-206.
- [48] Borja-Robalino. R., Monleón-Getino, A., & Rodellar J. (2020). *Estandarización de métricas de rendimiento para clasificadores Machine y Deep Learning*. Revista Ibérica de Sistemas e Technologias de Informação: risti, 30, pp. 172-184.
- [49] Hossin, M., Sulaiman, M., Mustapha, A., Mustapha, N., & Rahmat, R. (2011). *A hybrid evaluation metric for optimizing classifier*. Malasya. 2011 3rd Conference on Data Mining and Optimization (DMO), pp. 165-170.
- [50] Perotte, A., Pivovrov, R., Natarajan, K., Weiskopf, N., Wood, F. & Elhdad, N. (2014). *Diagnosis code assignment: models and evaluation metrics*. Journal of the American Medical Informatics Association: JAMIA, 21(2), pp. 237-237.
- [51] Géron, A. (2017). *Hands-on machine learning with scikit-Learn and Tensorflow: concepts, tools, and techniques to build intelligent systems*. Sebastopol, CA: O'Reilly Media, pp. 253-365.
- [52] Zaccone, G. & Karim, Md. R. (2018). *Deep Learning with TensorFlow - Second Edition*.
- [53] Rumelhart, D., Hinton, G. & Williams, R. (1986). *Learning Internal Representations by Error Propagation*.
- [54] Carandini M. (2006). *What simple and complex cells compute*. The Journal of physiology, 577(Pt 2), 463–466. [En línea]. Disponible en: <https://doi.org/10.1113/jphysiol.2006.118976>.
- [55] Movshon, J. A., Thompson, I. D. & Tolhurst D. J. (1978). *Nonlinear spatial summation in the receptive fields of complex cells in the cat striate cortex*. J Physiol 283, 78–100.
- [56] Fukushima K. (1980). *Neocognitron: a self organizing neural network model for a mechanism of pattern recognition unaffected by shift in position*. Biological Cybernetics, 36(4), pp. 193-202.
- [57] LeCun, Y., Bottou, L., Bengio, Y. & Haffner, P. (1998). *Gradient-based learning applied to document recognition*. In Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324.
- [58] Simard, P. Y., Steinkraus, D. & Platt, J. C. (2003). *Best practices for convolutional neural networks applied to visual document analysis*. Seventh International Conference on Document Analysis and Recognition. Proceedings, pp. 958-963.
- [59] Especificaciones de productos de Intel. [En línea]. Disponible en: <https://ark.intel.com/>.

- [60] Tarjetas gráficas NVIDIA GeForce. [En línea]. Disponible en: <https://www.nvidia.com/es-es/geforce/graphics-cards>.
- [61] NVIDIA, CUDA Toolkit. (2020). [En línea]. Disponible en: <https://developer.nvidia.com/cuda-toolkit>.
- [62] Klöckner, A., Pinto, N., Yunsup, L., Catanzaro, B., Ivanov, P. & Fasih, A. (2012). *PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation*. Parallel Computing. Volume 38, Issue 3, pp. 157-174.
- [63] Okuta, R., Unno, Y., Nishino, D., Hido, S. & Loomis, C. (2017). *CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations*. Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS).
- [64] Chauhan, M., Hammoshi, M. & Abdallah, B. (2016). *Recent Trends in Parallel Computing Accelerating High Arithmetic Intensity Storm Surge Model using CUDA*. Recent Trends in Parallel Computing. Volume 38, Issue 3, pp. 9–21.
- [65] Strigl, D., Kofler, K. & Podlipnig, S. (2010). *Performance and Scalability of GPU-Based Convolutional Neural Networks*. 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, pp. 317-324.
- [66] Li, X., Zhang, G., Huang, H. H., Wang, Z. & Zheng, W. (2016). *Performance Analysis of GPU-Based Convolutional Neural Networks*. 45th International Conference on Parallel Processing (ICPP), pp. 67-76.
- [67] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S. & Darrell, T. (2014). *Caffe: Convolutional architecture for fast feature embedding*. arXiv preprint arXiv:1408.5093.
- [68] Collobert, R., Kavukcuoglu, K. & Farabet, C. (2011). *Torch: A matlab-like environment for machine learning*. In BigLearn, NIPS Workshop.
- [69] Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde- Farley, D. & Bengio, Y. (2010). *Theano: a cpu and gpu math expression compiler*. In SciPy, volume 4, pp. 3.
- [70] Sharan, C. et al. (2014). *cudnn: Efficient primitives for deep learning*. arXiv preprint arXiv:1410.0759.
- [71] Krizhevsky, A. (2014). *One weird trick for parallelizing convolutional neural networks*. arXiv preprint arXiv:1404.5997v2.
- [72] Vasilache, N., Johnson, J., Mathieu, M., Chintala, S., Piantino, S. & LeCun, Y. (2015). *Fast convolutional nets with fbfft : A GPU performance evaluation*. arXiv preprint arXiv: 1412.7580.
- [73] Chen T. et al. (2015). *MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems*. arXiv preprint arXiv:1512.01274.