

Escuela Técnica Superior de Ingeniería Universidad de Huelva

Grado en Ingeniería Informática

Trabajo Fin de Grado

Aplicaciones de estrategias de Deep Learning para la
detección de animales en imágenes de foto-trampeo

Carlos García Silva

¿?/05/2025

Resumen

La importancia de un futuro sostenible se refleja cada día con mayor claridad en nuestra sociedad. Es por ello por lo que los estudios de biodiversidad han cobrado un papel fundamental en la compresión, conservación y valoración de la vida en la Tierra.

Una de las técnicas ampliamente utilizada para la realización de dichos estudios es la técnica del foto-trampeo. Se basa en el despliegue de cámaras automáticas en el entorno a estudiar para poder capturar imágenes de animales en su hábitat natural. Uno de sus principales problemas es la gran facilidad para generar grandes volúmenes de imágenes, en las que realizar un análisis manual puede resultar costoso y laborioso.

Este Trabajo de Fin de Grado presenta una red neuronal convolucional capaz de clasificar imágenes de foto-trampeo para identificar aquellas donde existe presencia de animales.

El sistema propuesto ha sido desarrollado mediante el uso de la técnica de *transfer learning*, empleando un modelo previamente entrenado con los pesos de *ImageNet*, basado en la arquitectura *EfficientNet-B5*. La implementación de *MegaDetector* como preprocesamiento previo de las imágenes de entrada ha resultado en un rendimiento superior en comparación con otros modelos que no han hecho uso de dicha implementación.

El experimento se realizó utilizando una colección de 31.670 imágenes obtenidas de cámaras de foto-trampeo, de las cuales 21.670 presentaban evidencia de presencia animal. El proceso de entrenamiento, validación y evaluación de la red se llevó a cabo utilizando una proporción del 70%, 15% y 15% del conjunto de imágenes, respectivamente.

[TO-DO] [PÁRRAFO DE RESULTADOS].

[TO-DO] [PÁRRAFO FINAL DE CONCLUSIONES Y VALORACIÓN].

Abstract

The significance of a sustainable future is becoming increasingly evident within our society. Consequently, the study of biodiversity has assumed a pivotal role in the comprehension, preservation and estimation of life on Earth.

A prevalent technique in the execution of such research is the photo-trapping technique. This method involves the implementation of automated cameras within the designated study environment, with the objective of capturing images of animals within their natural habitat. A salient challenge associated with this approach pertains to the generation of voluminous image data, which renders manual analysis to be both costly and time-consuming.

The present Final Degree Project proposes a convolutional neural network with the capacity to classify photo-trapping images to identify those in which animals are present.

The proposed system has been developed using the transfer learning technique, employing a model previously trained with ImageNet weights, based on the EfficientNet-B5 architecture. The implementation of MegaDetector as a pre-processing of the input images has resulted in a superior performance compared to other models that have not made use of such implementation.

The experiment was conducted using a collection of 31.670 images obtained from photo-trapping cameras, of which 21.670 showed evidence of animal presence. The network training, validation and evaluation process was carried out using a proportion of 70%, 15% and 15% of the image set, respectively.

[TO-DO] [PÁRRAFO DE RESULTADOS].

[TO-DO] [PÁRRAFO FINAL DE CONCLUSIONES Y VALORACIÓN].

Índice

Capítulo 1.....	10
1.1 Motivación	10
1.2 Objetivos.....	11
1.3 Competencias.....	12
1.4 Hardware y Software.....	12
1.5 Organización de la memoria.....	13
Capítulo 2.....	15
2.1 Historia y evolución del aprendizaje profundo	15
2.1.1 Cibernética (1940 – 1960)	16
2.1.2 Conexiónismo (1980 – 1995)	17
2.1.3 Deep Learning (2006 – Actualidad)	19
2.2 Aprendizaje profundo en la actualidad.....	20
Capítulo 3.....	24
3.1 Base de datos	24
3.2 Conjuntos de datos	27
3.3 Métricas de evaluación	28
Capítulo 4.....	29
4.1 Arquitecturas de la red	29
4.2 Hiperparámetros	31
4.2.1 Umbral de MegaDetector	32
4.2.2 Versión 1: Batch size	32
4.2.3 Versión 2: Algoritmo de optimización	32
4.2.4 Versión 3: Función de pérdida	33
4.2.5 Versión 4: Hiperparámetros de optimización y pérdida	33
4.2.6 Versión 5: Data augmentation	33
4.2.7 Versión 6: Early stopping	34
4.2.8 Versión 7: Fine tuning.....	34
4.3 Fase de entrenamiento	35
4.3.1 Búsqueda de umbral de MegaDetector	36
4.3.2 Preprocesamiento de imágenes con MegaDetector.....	37
4.3.3 Entrenamiento de MegaClassifier_a	40
4.3.4 Entrenamiento de MegaClassifier_b	50
4.3.5 Entrenamiento de MegaClassifier_c	57
Capítulo 5.....	65
5.1 Resultados en el conjunto de prueba	66
5.1.1 Resultados MegaClassifier_a	67
5.1.2 Resultados MegaClassifier_b	75
5.1.3 Resultados MegaClassifier_c	82
5.2 Análisis y Discusión	90
Capítulo 6.....	91
6.1 Conclusiones técnicas.....	91

6.2 Trabajos futuros	91
6.3 Valoración personal	91
Referencias	92
Apartado 1.....	95
Apartado 2.....	100
 2.1 Fundamentos	100
2.1.1 Perceptrón	100
2.1.2 Funciones de activación	102
2.1.3 Tipo de capas	104
2.1.4 Aplicación a problemas de clasificación.....	105
2.1.5 Aplicación a imágenes.....	106
 2.2 Desarrollo de redes neuronales	107
2.2.1 Etapa de entrenamiento	108
2.2.2 Etapa de entrenamiento: Hiperparámetros	119
2.2.3 Etapa de entrenamiento: Control y seguimiento.....	124
 Apartado 3.....	128
 3.1 Introducción.....	128
 3.2 Tipos de capas	130
3.2.1 Capas de convolución	130
3.2.2 Capas de activación	132
3.2.3 Capas de pooling.....	133
3.2.4 Capas upsampling y transposed convolution.....	134
3.2.5 Capas de softmax.....	138
3.2.6 Capas Fully Connected.....	138
 3.3 Arquitecturas populares.....	139
3.3.1 AlexNet.....	139
3.3.2 VGG.....	139
3.3.3 GoogleLeNet.....	140
3.3.4 ResNet.....	141
3.3.5 DenseNet.....	141
 3.4 Arquitectura utilizada en este trabajo	142
 Apartado 4.....	145

Índice de figuras

Figura 1: Principales frameworks de desarrollo de modelos de Deep Learning	12
Figura 2: Arquitectura de procesador M3 de Apple.....	13
Figura 3: Frecuencia de aparición de conceptos en publicaciones a lo largo de la historia	16
Figura 4: Demostración de cómo la función lógica XOR no puede modelarse linealmente	17
Figura 5: Representación del perceptrón multicapa y el algoritmo Backpropagation.....	18
Figura 6: Arquitectura LeNet	19
Figura 7: Arquitectura AlexNet	20
Figura 8: Rendimientos empresariales derivados de la IA por regiones (en millones de dolares)	21
Figura 9: Niveles de conducción autónoma	21
Figura 10: Distribución de las clases en el dataset original.....	24
Figura 11: Distribución binaria de dataset adaptado	25
Figura 12: Ejemplo de foto diurna con presencia animal y encuadre muy abierto	25
Figura 13: Ejemplo de imagen diurna con animal camuflado resaltado.....	25
Figura 14: Ejemplo de imagen diurna con animal	25
Figura 15: Imagen nocturna con animal camuflado y resaltado	26
Figura 16: Imagen nocturna con presencia animal.....	26
Figura 17: Distribución binaria del dataset final.....	26
Figura 18: Esquema simplificado modelo MegaClassifier_c	30
Figura 19: Esquema simplificado modelo MegaClassifier_a	31
Figura 20: Esquema simplificado modelo MegaClassifier_b	31
Figura 21: Matriz de confusión ideal VS matriz de confusión umbral=0.0015 sobre el conjunto de entrenamiento	37
Figura 22: Ejemplo de generación de máscara binaria y su aplicación sobre una imagen con presencia animal	38
Figura 23: Explicación visual de la operación aplicada para el recorte ajustado.....	39
Figura 24: Ejemplo de preprocessado completo con MegaDetector sobre una imagen con presencia animal	39
Figura 25: Gráficas de entrenamiento de accuracy, auc, los y f1-score de MegaClassifier_a_v1.0	41
Figura 26: Gráficas de entrenamiento de accuracy, auc, los y f1-score de MegaClassifier_a_v2.0	42
Figura 27: Gráficas de entrenamiento de accuracy, auc, los y f1-score de MegaClassifier_a_v3.1	43
Figura 28: Gráfica teórica de LRFinder.....	44
Figura 29: Gráfica método LRFinder en MegaClassifier_a	45
Figura 30: Gráficas de entrenamiento de accuracy, auc, los y f1-score de MegaClassifier_c_v4.2	45
Figura 31: Gráficas de entrenamiento de accuracy, auc, los y f1-score de MegaClassifier_c_v5.0	46
Figura 32: Gráficas de entrenamiento de accuracy, auc, los y f1-score de MegaClassifier_c_v6.1	47
Figura 33: Gráficas de entrenamiento de accuracy, auc, los y f1-score de MegaClassifier_c_v7.3	48
Figura 34:Gráficas de entrenamiento de accuracy, auc, los y f1-score de MegaClassifier_c_v8.0	49
Figura 35: Etapas del proceso de Machine Learning.....	96
Figura 36: Esquema de aplicaciones del Machine Learning	98
Figura 37: Comparativa Machine Learning VS Deep Learning.....	99
Figura 38: Anatomía de una neurona biológica	101
Figura 39: Perceptrón y su proceso de aprendizaje.....	101

Figura 40: Estructura básica de una red neuronal multicapa	104
Figura 41: Transformación softmax y codificación one-hot.....	105
Figura 42: Ejemplo de flattening.....	106
Figura 43: Etapa de entrenamiento e inferencia de un modelo de machine learning	107
Figura 44: Ejemplo de matriz de confusión para 4 clases.....	109
Figura 45: Matriz de confusión considerando únicamente las clases positivas y negativas	110
Figura 46: Ejemplo de gráfica de Curva ROC.....	112
Figura 47: Representación gráfica de la dirección del gradiente.....	115
Figura 48: Comparación de algoritmos de optimización.....	117
Figura 49: Comparativa de underfitting y overfitting en una gráfica train vs validation de la métrica loss del entrenamiento de una red	121
Figura 50: Gráfica conceptual de los mínimos aplanados y los profundos, donde el eje-Y indica el valor de la función de perdida y el eje-X los parámetros	122
Figura 51: Efecto en un entrenamiento aplicando diferentes learning rate	122
Figura 52: Ejemplo de gráficas de loss y accuracy en un entrenamiento de 100 épocas.....	124
Figura 53: Áreas del lóbulo cerebral posterior.....	128
Figura 54: Ejemplo de extracción de características en capas de convolución	129
Figura 55: Detección de bordes en imagen mediante convolución.....	130
Figura 56: Operación de convolución	131
Figura 57: Operaciones maxpooling y averagepooling	133
Figura 58: Técnica de Nearest Neighbor	134
Figura 59: Técnica de Bed of nails	134
Figura 60: Técnica de max-pooling y max-unpooling.....	135
Figura 61: Operación de convolución traspuesta	137
Figura 62: Operación de convolución como multiplicación de matrices.....	137
Figura 63: Arquitectura AlexNet	139
Figura 64: Arquitectura VGG16	140
Figura 65: Arquitectura GoogleLeNet	140
Figura 66: Arquitectura ResNet-50	141
Figura 67: Arquitectura DenseNet	142
Figura 68: Diferentes tipos de módulos de EfficientNet.....	144
Figura 69: Arquitectura EfficientNet-B5	144

Capítulo 1

Propuesta de Proyecto

En el presente capítulo introductorio, se exponen las motivaciones que han fundamentado la realización de este trabajo, los objetivos que se han establecido y el sistema implementado para su consecución. Asimismo, se detalla el desarrollo de dichos objetivos y las diversas tecnologías empleadas.

1.1 Motivación

En años recientes, se han observado significativos avances en el ámbito de la tecnología, lo cual ha generado un impacto sustancial en la sociedad contemporánea. En cierta medida, se puede afirmar que la dependencia hacia esta área es una realidad en el contexto cotidiano. Diversos analistas consideran que la actual situación representa una nueva revolución industrial. En este sentido, se puede afirmar que, del mismo modo que sucedió con la industria textil en la primera revolución industrial, la electricidad en la segunda y la electrónica en la tercera, hoy en día se puede constatar que la informática se erige como uno de los pilares que han permitido identificar a esta nueva revolución como la denominada *Industria 4.0*.

En el campo de la informática, se ha observado un aumento significativo en la implementación de algoritmos y técnicas de inteligencia artificial en los últimos años. Este fenómeno se atribuye al incremento en la capacidad de procesamiento y al vasto volumen de datos disponibles en la actualidad, en contraste con la limitada disponibilidad de hace unos años. En particular, nos referimos a los algoritmos de aprendizaje profundo (*Deep Learning*)¹, que se fundamentan principalmente en redes neuronales artificiales.

Dentro del ámbito de la inteligencia artificial, los algoritmos han experimentado una notable evolución en su capacidad de procesamiento de datos. La implementación de estos algoritmos se lleva a cabo durante el proceso de entrenamiento, mediante la utilización de vastas cantidades de datos. Además, estos algoritmos se ejecutan en sistemas con capacidades de cómputo que, hasta hace una década, se consideraban inalcanzables para las máquinas. Los resultados obtenidos recientemente han superado las expectativas, logrando niveles de rendimiento que anteriormente eran impensables para las máquinas y, en algunos casos, incluso superando a los humanos en ciertas tareas. Este avance representa una significativa apertura a la capacidad de abordar nuevos y complejos problemas que, anteriormente, solo podían resolverse mediante el ingenio humano.

El propósito de este estudio es examinar y desentrañar la implementación del *Deep Learning* en un amplio espectro de aplicaciones tecnológicas. En este sentido, se centrará en la visión artificial para concebir un sistema de detección de animales en imágenes de foto-trampeo. El enfoque metodológico seleccionado se fundamenta en la utilización de *MegaDetector*², aprovechando sus capacidades y descartando las áreas de escaso interés de las imágenes.

¹ Debido a la popularidad del término Deep Learning, durante el resto del trabajo lo utilizaremos en sustitución del término aprendizaje profundo.

² Enlace oficial al modelo de detección: <https://github.com/microsoft/CameraTraps/tree/main>

Posteriormente, se implementa una red neuronal convolucional (CNN)³ para la clasificación de las imágenes, permitiendo distinguir entre aquellas que contienen animales y las que no. Este procedimiento se lleva a cabo con el propósito de mejorar la precisión y la eficiencia del proceso de monitoreo de la fauna en comparación con un sistema similar que no incorpora el preprocesamiento de las imágenes de entrada.

Este proyecto constituye una oportunidad para profundizar en el conocimiento y la aplicación de técnicas avanzadas de *Deep Learning*, que no se abordan en el plan de estudios del Grado en Ingeniería Informática. Además, posee una clara relevancia práctica en el ámbito de la conservación de la biodiversidad. La implementación de CNN en la clasificación de imágenes de foto-trampeo permite optimizar el monitoreo de la fauna, facilitando el análisis de datos con mayor eficiencia y precisión. Este avance no solo contribuye al desarrollo de nuevas metodologías en el procesamiento de imágenes, sino que también abre la posibilidad de generar herramientas de apoyo en estudios ecológicos y en la preservación de especies, lo que destaca la importancia de la inteligencia artificial en la resolución de problemas del mundo real.

1.2 Objetivos

En el marco de la presente investigación, se han planteado dos objetivos primordiales a alcanzar:

- Adquisición de conocimiento: Introducción y estudio teórico de las técnicas de *Deep Learning*, con el propósito de comprender su naturaleza, su evolución hasta la actualidad y los fundamentos necesarios para su aplicación en el problema específico de interés.
- Implementación: En segundo lugar, se implementará un sistema basado en CNN que permitirá llevar a cabo una detección y clasificación eficaces de la presencia de animales en imágenes de foto-trampeo.

Para una mayor profundización en el tema, se propone una subdivisión de los objetivos principales en los siguientes puntos: En primer lugar, es necesario realizar una exhaustiva revisión bibliográfica y un análisis de los conceptos teóricos básicos acerca del *Deep Learning*. En segundo lugar, se obtendrá, analizará y preparará un conjunto de imágenes⁴ para el entrenamiento, la validación y la evaluación de los modelos implementados. En tercer lugar, se procederá al diseño, implementación y evaluación de varios modelos entrenados para la detección de animales en imágenes de foto-trampeo. Por último, se llevará a cabo una búsqueda e implementación de métricas para la evaluación de los modelos, que permitirá cuantificar objetivamente la calidad de los resultados obtenidos.

³ Siglas del término inglés Convolutional Neural Network. Por motivos de simplicidad, durante el resto del trabajo utilizaremos dichas siglas en sustitución del término red neuronal convolucional.

⁴ Debido a su popularidad, a partir de ahora el término inglés *Dataset* sustituirá el término conjunto de datos / imágenes.

1.3 Competencias

El desarrollo de este proyecto ha permitido la aplicación y el refuerzo de diversas competencias del Grado en Ingeniería Informática, particularmente aquellas vinculadas con el aprendizaje computacional y el diseño e implementación de sistemas basados en inteligencia artificial. Específicamente, se ha concentrado en el estudio y la aplicación de las CNN para clasificación de imágenes de foto-trampeo. Este proceso ha implicado una revisión exhaustiva del estado del arte en *Deep Learning*, así como la experimentación con modelos diseñados específicamente para la detección de fauna en imágenes digitales.

En contraste, la imperativa de operar con *datasets* de un entorno real ha posibilitado la evolución de competencias en la extracción automática de información y filtrado de imágenes para potenciar la precisión de los modelos. En este sentido, el trabajo no solo ha servido como un ejercicio de profundización en técnicas de *Deep Learning* que no se abordan en profundidad en el plan de estudios, sino también ha permitido aplicar estos conocimientos a un caso práctico de relevancia en el ámbito de la conservación de la fauna, evidenciando la importancia del aprendizaje computacional en la solución de problemas en situaciones reales.

1.4 Hardware y Software

Para poder alcanzar las metas anteriormente establecidas de manera efectiva, resulta necesario el uso de un sistema cuyo hardware posea la capacidad de ejecutar algoritmos de *Deep Learning*. Además, dicho sistema debe estar equipado con un software especializado en el diseño y creación de modelos que hagan uso de estos algoritmos.

En lo que respecta al software empleado, la implementación del modelo se realizará en el lenguaje de programación Python 3.8 debido a su amplia popularidad en aplicaciones de *Machine Learning*⁵. (ver Figura 1).



Figura 1: Principales frameworks de desarrollo de modelos de Deep Learning

⁵ Término inglés cuyo significado se corresponde con el concepto de aprendizaje automático.

El marco de trabajo seleccionado para la ejecución de este proyecto es *TensorFlow*, debido a su exhaustiva documentación y activa comunidad de usuarios, lo que facilitará la implementación del proyecto. Desde su versión 2.0, TensorFlow incorpora de forma nativa la librería Keras, que permite la creación de redes neuronales desde un alto nivel gracias a su estructura de capas de abstracción. Esta integración proporciona una metodología eficiente para la concepción de modelos de *Deep Learning*, permitiendo descender en la abstracción solo cuando sea necesario y de forma más precisa.

Para la implementación y el entrenamiento del modelo de clasificación se utilizarán aplicaciones como Jupyter Notebook y Visual Studio Code, junto con el entorno de desarrollo proporcionado por Anaconda, que nos permitirá instalar y gestionar fácilmente las librerías necesarias, como Tensorflow, OpenCV, Numpy y otras más.

Para la implementación del proyecto, se ha utilizado un MacBook Pro de 14 pulgadas, equipado con el chip Apple M3 (ver Figura 2)⁶, 16 GB de memoria unificada y un almacenamiento SSD de 512 GB. Este dispositivo ha permitido la ejecución eficiente de los experimentos, gracias a su CPU de 8 núcleos, optimizada para tareas de alto rendimiento y eficiencia, así como su GPU de 10 núcleos, que incorpora tecnologías avanzadas como Dynamic Caching, Mesh Shading y Ray Tracing acelerado por hardware. Además, la librería TensorFlow ha experimentado actualizaciones que han posibilitado su compatibilidad nativa con la nueva arquitectura de procesadores Apple Silicon, lo que ha conducido a un notable incremento en su rendimiento. La autonomía del dispositivo, con una duración de hasta 22 horas de batería, ha facilitado la realización de entrenamientos de larga duración sin interrupciones, optimizando el flujo de trabajo durante el desarrollo del proyecto.

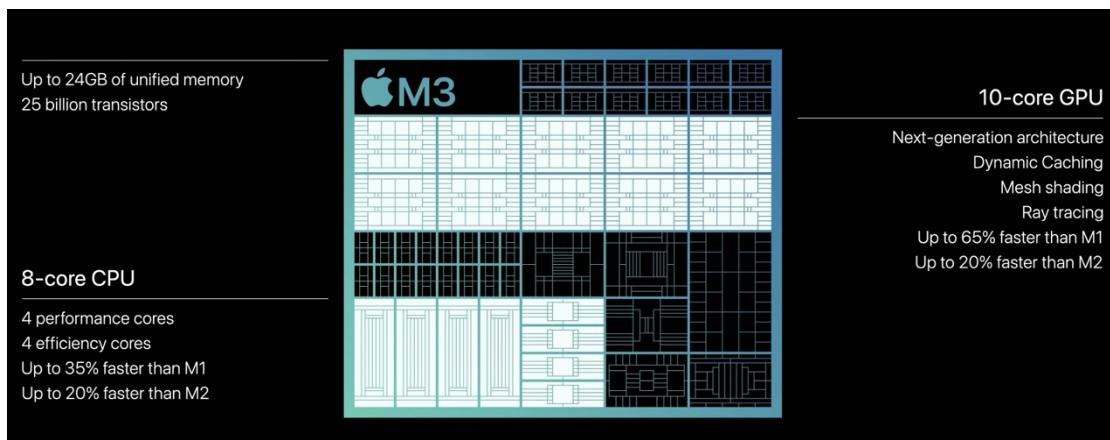


Figura 2: Arquitectura de procesador M3 de Apple

1.5 Organización de la memoria

Además del primer capítulo, en el que se presenta la propuesta, la motivación y los objetivos del trabajo, así como las competencias desarrolladas y los recursos hardware y software empleados, esta memoria se organiza en los siguientes capítulos:

⁶ Fuente: <https://www.notebookcheck.org/fileadmin/Notebooks/Apple/apple-m3-stats.jpg>

- **Capítulo 2 Introducción y estado de arte:** Se presenta un análisis exhaustivo de la implementación de técnicas de *Deep Learning* en el ámbito de la clasificación de imágenes. Este capítulo examina estudios previos y tecnologías relevantes en el contexto del reconocimiento de fauna mediante foto-trampeo.
- **Capítulo 3 Materiales:** Se aborda la descripción de los materiales empleados en el desarrollo del modelo. En este capítulo se describen los datos utilizados en la implementación del modelo, incluyendo la base de datos empleada, los conjuntos de datos generados para el entrenamiento, validación y evaluación, y las métricas utilizadas para medir el desempeño del sistema.
- **Capítulo 4 Metodología:** Como su propio nombre indica, se centra en la metodología empleada, detallando las estrategias implementadas en el desarrollo del modelo. En este sentido, se aborda la selección de la arquitectura de la red neuronal, los hiperparámetros utilizados y el proceso de entrenamiento del modelo.
- **Capítulo 5 Resultados y discusión:** Presenta los resultados obtenidos en el conjunto de evaluación, analizando su rendimiento en términos de las métricas establecidas y discutiendo las fortalezas y limitaciones del enfoque adoptado.
- **Capítulo 6 Conclusiones y trabajos futuros:** Ofrece una reflexión final sobre las implicaciones del estudio realizado y plantea posibles líneas de trabajo para futuras investigaciones. En este capítulo se presentan las conclusiones técnicas extraídas del estudio, se identifican posibles mejoras y líneas de trabajo futuro, y se incluye una valoración personal sobre la experiencia adquirida durante la realización del proyecto.

La memoria prosigue con el capítulo destinado a la bibliografía, en el cual se efectúan las referencias de los artículos, libros y recursos empleados para la elaboración del trabajo.

Además de los capítulos principales, la memoria contiene un anexo teórico adicional que profundiza en los conceptos fundamentales del trabajo. Organizado de la siguiente manera:

- **Apartado 1 Introducción al Deep Learning:** Se presentan los principios básicos del *Deep Learning*, destacando su evolución y relevancia en la actualidad.
- **Apartado 2 Redes Neuronales:** En este apartado se abordan los fundamentos del funcionamiento de las redes neuronales, desde su desarrollo hasta las diferentes etapas del entrenamiento e inferencia, detallando aspectos como la configuración de hiperparámetros y el control del proceso de entrenamiento.
- **Apartado 3 Redes Neuronales Convolucionales:** Se analiza la estructura y funcionamiento de las CNN, describiendo los tipos de capas utilizadas, las arquitecturas más populares y la arquitectura específica empleada en este trabajo.
- **Apartado 4 MegaDetector:** Finalmente se presenta MegaDetector, utilizado para el preprocesamiento de las imágenes de foto-trampeo utilizadas como entrada para nuestro modelo clasificador, explicando su funcionamiento y su integración dentro del sistema desarrollado.

Capítulo 2

Introducción y estado del arte

Una vez presentadas las motivaciones y objetivos, se procederá a proporcionar una visión integral sobre el *Deep Learning*, destacándolo como una disciplina especializada dentro de un campo más amplio, el *Machine Learning*. Para ello, se iniciará con una introducción a los conceptos fundamentales del aprendizaje profundo, seguida de un recorrido histórico, en el que se podrán conocer fundamentos teóricos y la evolución experimentada desde los años cincuenta hasta las avanzadas aplicaciones actuales.

Finalmente, se ofrece una revisión general de las numerosas aplicaciones del *Deep Learning*, demostrando su impacto significativo en diversos campos y su potencial para seguir transformando la tecnología y la sociedad.

2.1 Historia y evolución del aprendizaje profundo

El *Deep Learning* ha emergido como una de las áreas más innovadoras dentro del campo de la inteligencia artificial, ya que permite el desarrollo de sistemas capaces de resolver problemas de gran complejidad sin necesidad de definir explícitamente cada uno de los pasos a seguir. En el anexo teórico ([Apartado 1](#))

Introducción al Deep-Learning) se exponen los fundamentos del Deep Learning, describiendo su funcionamiento y las distintas etapas que conforman el desarrollo de un modelo basado en este paradigma. A partir de estos conceptos, en este capítulo analizaremos la evolución histórica de las técnicas de aprendizaje automático, desde sus primeras aproximaciones hasta los avances más recientes que han permitido consolidar el aprendizaje profundo como una herramienta clave en múltiples ámbitos tecnológicos.

Al hablar de la historia del *Deep Learning*, también hay que abordar la historia del *Machine Learning*, ya que, como hemos comentado anteriormente, el aprendizaje automático sirve de base para el aprendizaje profundo. A lo largo de su evolución, el *Machine Learning* ha pasado por diferentes etapas, cada una de ellas marcada por el avance en nuevas técnicas y aplicaciones.

Aunque el actual auge del *Machine Learning* nos pueda llevar a confusión, su desarrollo no es reciente, sino que se remonta a hace muchos años (ver Figura 3)⁷, cuando no era fácil y su camino fue largo y complejo. Podríamos dividir su evolución en tres etapas clave, algunas de las cuales suscitan menos interés debido a las limitaciones de los modelos existentes en aquel momento. Estos altibajos son los que han provocado que en la actualidad se encuentre tan avanzado y diversificado, y sea una de las herramientas más poderosas, capaz de resolver problemas complejos, como el procesamiento de imágenes, que abordaremos en este trabajo de fin de grado.

⁷ Fuente: <https://books.google.com/ngrams/...>

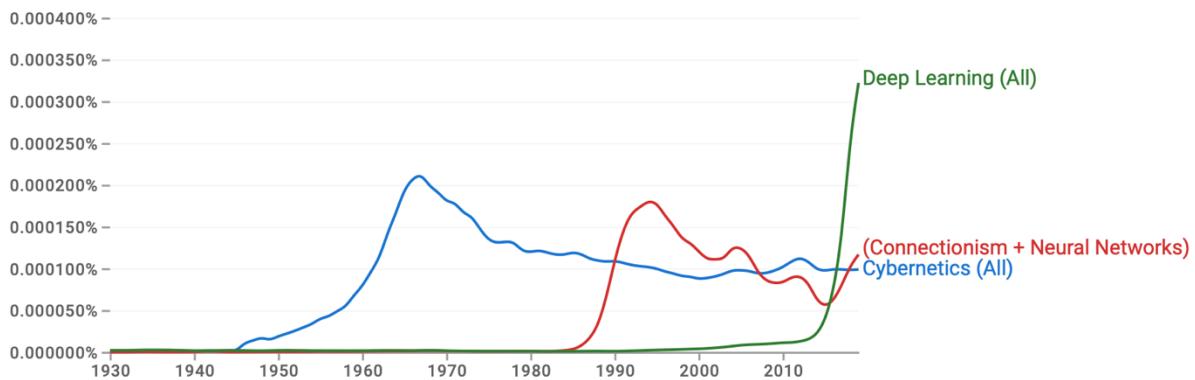


Figura 3: Frecuencia de aparición de conceptos en publicaciones a lo largo de la historia

2.1.1 Cibernetica (1940 – 1960)

En 1936, Alan Turing publicó un estudio en el que describió las máquinas de Turing [1], formalizando el concepto de algoritmo, lo que marcó el comienzo de la informática moderna.

En la década de 1940, Warren McCulloch y Walter Pitts presentaron un modelo matemático inspirado en las neuronas biológicas en 1943 [2]. Este modelo consistía en recibir un conjunto de entradas n ($x_1, x_2, x_3, \dots, x_n$) que se multiplicaban por pesos asociados w ($w_1, w_2, w_3, \dots, w_n$) y se sumaban. La salida dependía de una función de activación umbral que devolvía 1 si el valor era positivo y 0 si era negativo. Con un ajuste adecuado de los pesos, esta neurona podía realizar operaciones lógicas simples, como AND⁸, OR⁹ y NOT¹⁰, lo que permitió usarla para razonamientos lógicos sencillos.

En 1950, Alan Turing publicó un artículo en el que formulaba una pregunta fundamental que revolucionó el campo de la computación: “¿Puede una máquina pensar?”. En este artículo [3], Turing presentó su famoso “Test de Turing”. Gracias a este artículo, Alan Turing es considerado el padre de la informática.

Posteriormente, en 1958, Frank Rosenblatt propone el modelo del *perceptrón* (ver 2.1.1 Perceptrón) basándose en el modelo matemático de Walter Pitts y Warren McCulloch [4]. Se trataba del primer modelo matemático capaz de “aprender” a representar una función ajustando los pesos de sus entradas a partir de ejemplos dados.

Posteriormente, comenzaron a surgir modelos de redes neuronales que combinaban varios perceptrones, lo que permitía clasificar más de dos clases y asignar cada una a un perceptrón. En 1960, de la mano de Bernard Widrow, nació el modelo ADELIN [5], que tenía la peculiaridad de permitir cuantificar el error y ajustar los pesos durante el entrenamiento en función del error cometido.

⁸ Operación lógica de disyunción.

⁹ Operación lógica de conjunción.

¹⁰ Operación lógica de negación.

Pese a todo ello, los modelos seguían siendo lineales. En 1969, Marvin Minsky y Seymour Papert [6] demostraron que los perceptrones no eran capaces de resolver funciones no lineales, como la función lógica XOR, también llamada OR exclusiva. (ver Figura 4)¹¹.

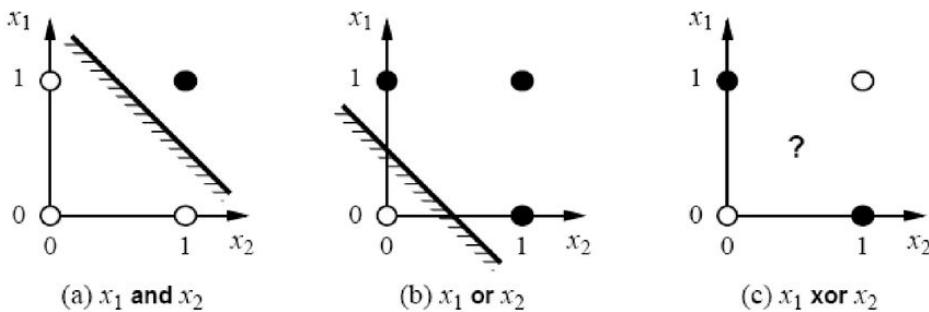


Figura 4: Demostración de cómo la función lógica XOR no puede modelarse linealmente

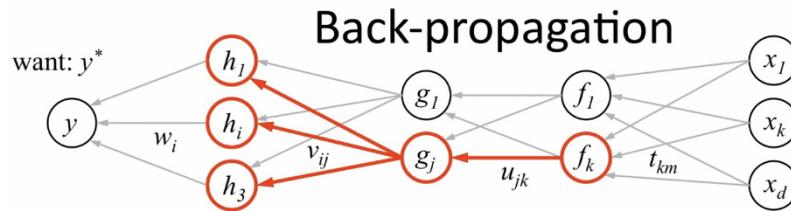
Esto ocasionó una gran decepción dentro de la comunidad y frenó el auge de esta tecnología, lo que conllevó un descenso del interés por su desarrollo en los años venideros.

2.1.2 Conexionismo (1980 – 1995)

A pesar del desinterés, las investigaciones en el campo del *Machine Learning* no se interrumpieron por completo. En 1986, David Rumelhart, Geoffrey Hinton y Ronald Williams redescubrieron el algoritmo de retropropagación, también conocido como backpropagation [7]. Gracias a este algoritmo, las redes multicapa podían entrenar eficazmente y modelar funciones no lineales, lo que supuso un gran avance en el desarrollo de este campo. La importancia de este hecho fue tal que hoy en día sigue considerándose uno de los algoritmos más viables para recalcular los pesos en redes neuronales (ver Figura 5)¹².

¹¹ Fuente: [https://slideplayer.com.br/slide/14283270/89/images/47/...](https://slideplayer.com.br/slide/14283270/89/images/47/)

¹² Fuente: https://miro.medium.com/v2/resize:fit:1400/1*_ZpFxEmEkigpz3AzrmZYAQ.png



1. receive new observation $x = [x_1 \dots x_d]$ and target y^*
2. **feed forward:** for each unit g_j in each layer $1 \dots L$
compute g_j based on units f_k from previous layer: $g_j = \sigma\left(u_{j0} + \sum_k u_{jk} f_k\right)$
3. get prediction y and error $(y - y^*)$
4. **back-propagate error:** for each unit g_j in each layer $L \dots 1$

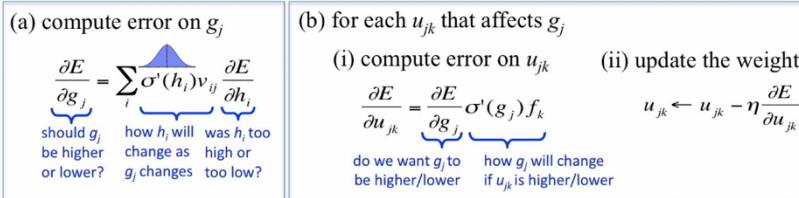


Figura 5: Representación del perceptrón multicapa y el algoritmo Backpropagation

En 1989, Kurt, Max y Halbert White, demostraron matemáticamente que las redes neuronales multicapa son aproximaciones universales [8], lo que prueba que con el uso de múltiples capas ocultas una red neuronal es capaz de modelar cualquier función matemática, incluida la función XOR. Este hecho hizo que resurgiera su popularidad dentro de la comunidad científica.

Ese mismo año, Yann LeCun y sus colaboradores presentaron una de las primeras aplicaciones de redes neuronales en el mundo real [9]. Creó una red denominada LeNet capaz de clasificar imágenes de dígitos escritos a mano con una tasa de error aproximada del 5 %.

La estructura de LeNet incluía una capa convolucional como primera capa oculta (ver Figura 6 para mejor comprensión)¹³. En lugar de asignar un peso a cada píxel, esta capa utilizaba un pequeño conjunto de pesos que formaban un filtro de convolución para extraer características de la imagen de entrada, como la detección de vértices o líneas. La siguiente capa se denomina *pooling* y reduce las características extraídas agrupando los píxeles por vecindad y comprimiéndolos en un solo valor con el objetivo de mantener solo la información más relevante.

Después de concatenar dos pares de estas capas, la red se conectaba a otra red multicapa más convencional en la que se clasificaban los dígitos basándose en las características extraídas, obteniendo en la salida una serie de nodos que también clasificaban los dígitos.

¹³ Fuente: https://miro.medium.com/v2/resize:fit:1400/1*bGjusyTjh2SACnkkMHU_hA.png

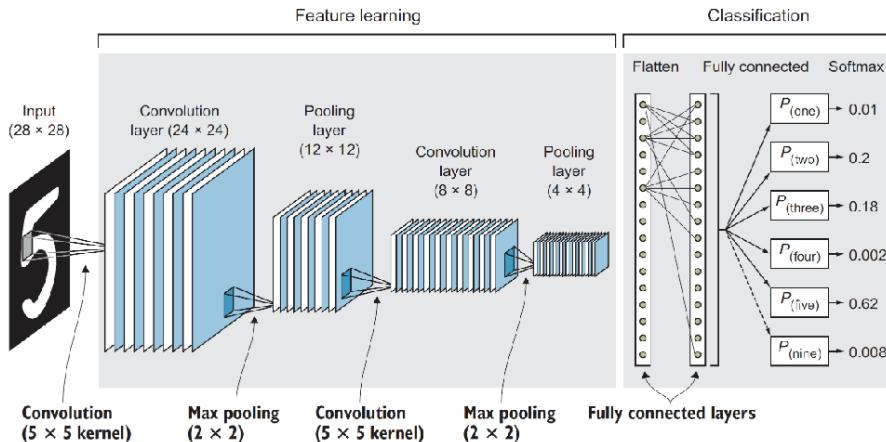


Figura 6: Arquitectura LeNet

Durante los años 90, las técnicas de *Machine Learning* avanzaron y las redes neuronales comenzaron a utilizarse en nuevas áreas. Se adoptó un enfoque orientado a programas de análisis de datos para extraer conclusiones de ellos con buenos resultados. El éxito que estaban presentando impulsó propuestas cada vez más ambiciosas, como la de aplicarlas al procesamiento del habla y la toma de decisiones. Sin embargo, en 1991, Sepp Hochreiter demostró que entrenar redes neuronales muy profundas era extremadamente complejo [10]. El problema radicaba en que el error que llegaba a las capas superficiales era tan insignificante que no permitía ajustar los pesos de forma efectiva.

La aparición de técnicas alternativas como los árboles de decisión y las máquinas vector soporte (SVM), que lograban buenos resultados sin el elevado coste computacional del entrenamiento de las redes neuronales, provocó que el machine learning volviera a caer en el desinterés durante varios años.

2.1.3 Deep Learning (2006 – Actualidad)

No fue hasta 2006 y 2007 que el machine learning volvió a resurgir. Geoffrey Hinton introdujo las Deep Belief Networks (DBN) [11] y utilizó por primera vez el término deep learning.

La propuesta de Hinton era un método innovador para entrenar de manera efectiva redes neuronales profundas mediante una estrategia llamada *greedy layer-wise pretraining*. Consistía en preentrenar cada capa de la red mediante un aprendizaje no supervisado utilizando *Restricted Boltzmann Machines* (RBM) [12], variantes de *Boltzmann Machines* con restricciones en las conexiones formando un grafo bipartito. Una vez preentrenado el modelo, se aplicaban los algoritmos de retropropagación para ajustar los pesos de la red de manera más eficiente.

El interés en las redes neuronales profundas, que había resurgido con el trabajo de Geoffrey, realmente despegó en 2012. Ese año, unos estudiantes de doctorado bajo la supervisión de Hinton presentaron AlexNet [13] (ver Figura 7)¹⁴ en la competición *ImageNet*. El objetivo de la competición era etiquetar imágenes en mil categorías diferentes a partir de un conjunto de datos con millones de imágenes. Lo impresionante de este modelo fue que logró reducir el error en la

¹⁴ Fuente: [https://www.researchgate.net/publication/320723863/figure/fig4/...](https://www.researchgate.net/publication/320723863/figure/fig4/)

clasificación de imágenes significativamente, bajándolo del 26 % a alrededor del 16 %. Este hito llamó la atención de muchos investigadores y grandes empresas.

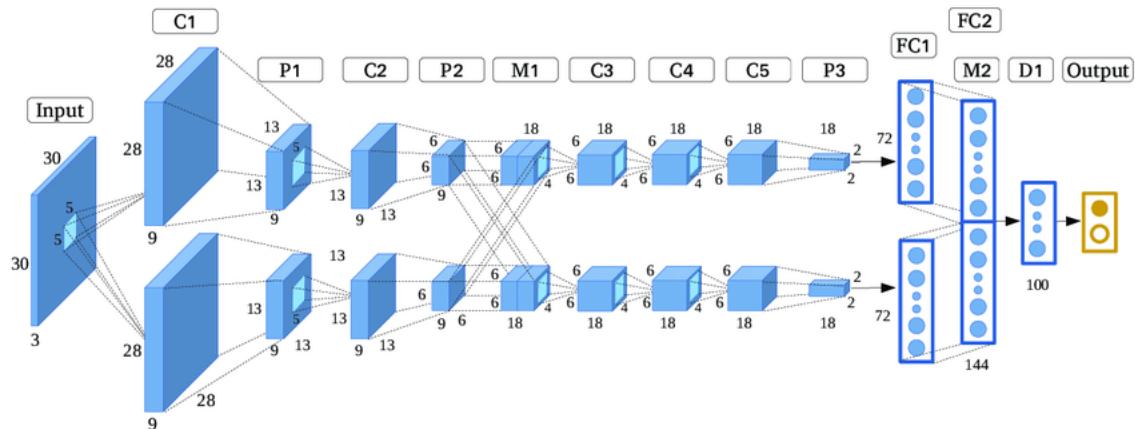


Figura 7: Arquitectura AlexNet

El interés no hizo más que crecer y, en 2014, Google presentó *GoogLeNet*, que incluía un módulo llamado *Inception* [14]. La novedad que este módulo aportaba era la posibilidad de realizar convoluciones de diferentes tamaños en paralelo, lo que ayudaba a encontrar el tamaño de kernel ideal. Así se consiguió reducir el error medio al 6,7 % en la competición de *ImageNet* de ese mismo año.

Luego, en 2015, Microsoft creó *ResNet* [15], una nueva red que introducía una modificación: conectar la capa de salida a otra capa no inmediata para permitir transmitir el error a capas más superficiales. Esto hizo que, en la *ImageNet* de ese año, el error medio se redujera al 3,6 %, siendo la primera vez que se obtenían valores inferiores al error humano, que se considera aproximado al 5 %.

2.2 Aprendizaje profundo en la actualidad

En la actualidad, el *Deep Learning* se ha convertido en una de las herramientas más utilizadas gracias a su capacidad para abordar una amplia variedad de problemas y su eficiencia.

Un factor clave para este hecho ha sido el desarrollo del hardware, en particular el de las tarjetas gráficas, que permiten procesar grandes cantidades de datos de manera eficiente y facilitan el entrenamiento de modelos complejos.

Una de las áreas donde ha tenido un impacto significativo es en la personalización del contenido y de las recomendaciones a los usuarios. Gracias al manejo de grandes cantidades de datos, lo que hoy en día se conoce como *Big Data*, muchas empresas han decidido diseñar modelos que se ajusten a las preferencias de sus usuarios y ofrecerles sugerencias personalizadas según el contenido que consumen.

El avance continuo de la potencia de cómputo y la reducción de costes ha hecho que el entrenamiento de modelos de *Deep Learning* sea cada vez más accesible, lo que impulsa aún más su integración en más sectores. En el ámbito del marketing, por ejemplo, el uso del *Big Data* para entrenar modelos que ajustan las ofertas y recomendaciones a los usuarios ha demostrado

ser extremadamente eficaz, lo que hace que servicios de *streaming* como *Netflix* sean un ejemplo del impacto positivo del *Deep Learning* en la experiencia de uso de los usuarios de estas plataformas. En la Figura 8¹⁵ se muestra el efecto del uso de la inteligencia artificial en los rendimientos empresariales.

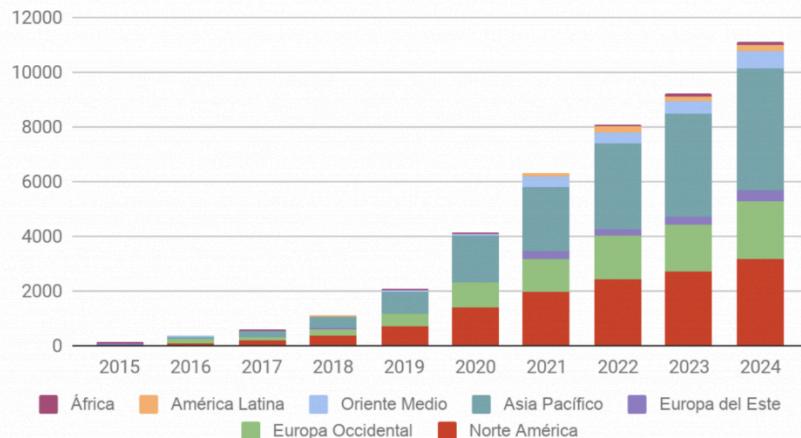


Figura 8: Rendimientos empresariales derivados de la IA por regiones (en millones de dólares)

Antes, todo en un videojuego era diseñado por equipos de personas: las mecánicas de los personajes, los entornos y los gráficos. Hoy en día, gracias a los avances en *Machine Learning*, la IA puede encargarse de muchas de estas tareas, lo que da lugar a mecánicas de juego más fluidas y entornos aleatorios. De este modo, los desarrolladores tienen mayor libertad, puesto que ahorrarían tiempo al no tener que diseñar los detalles manualmente. Además, la calidad de las imágenes ha mejorado, por ejemplo, NVIDIA ha desarrollado técnicas como DLSS 2.0 (*Deep Learning Super Sampling*) [16], que permiten a las tarjetas gráficas renderizar imágenes de menor resolución y luego escalarlas a alta resolución. Esto hace que la carga de trabajo de la tarjeta gráfica sea menor, aumenta la tasa de fotogramas por segundo y mantiene la nitidez en la imagen, por lo que el usuario tiene una mejor experiencia de juego sin sacrificar el rendimiento.

Otro sector en el que la inteligencia artificial ha cobrado importancia es el de la automoción, donde se ha producido una evolución significativa desde los sensores que detectan cuando un vehículo se sale de la vía hasta los coches que pueden estacionar solos. En la actualidad, hay una gran carrera entre las múltiples compañías automovilísticas para lograr el primer vehículo completamente autónomo, que eliminaría la necesidad de intervención humana. (ver Figura 9)¹⁶.

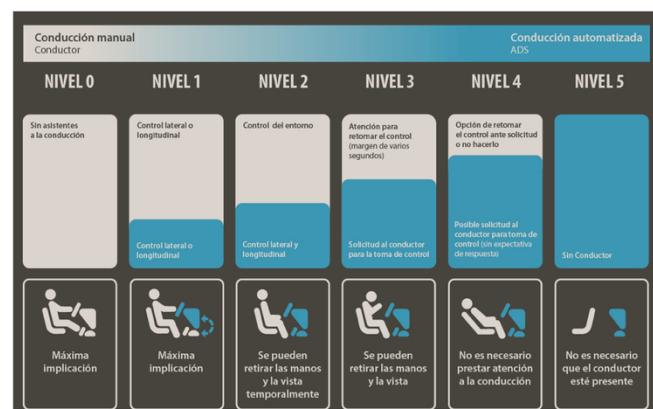


Figura 9: Niveles de conducción autónoma

¹⁵ Fuente: <https://ost.torrejuana.es/wp-content/uploads/2019/05/...>

¹⁶ Fuente: <https://www.km77.com/images/medium/5/7/9/5/2autonomo-km77-3-.335795.jpg>

Este sector está cada vez más cerca de alcanzar esta meta. Muchas marcas están desarrollando su propia inteligencia artificial con este propósito. Sin embargo, se trata de un tema delicado, ya que la seguridad y la vida de los ocupantes dependen de la precisión de la IA y no es posible prever ni entrenar todas las situaciones reales. De momento, solo se han implementado los primeros niveles, dependiendo del país y la legislación. De cara al futuro, aún se deben superar ciertos desafíos para que esto se convierta en una realidad común, algo que parece estar cada vez más cerca.

Los asistentes virtuales, como *Siri*, de Apple, *Alexa*, de Amazon, y *Google Assistant*, se han integrado profundamente en nuestra vida diaria gracias a los avances de la inteligencia artificial. Inicialmente, podían realizar tareas simples como poner una alarma o agregar una nota al calendario, pero han evolucionado enormemente y hoy en día pueden hacer llamadas para reservar establecimientos o controlar dispositivos de domótica en nuestros hogares. Su capacidad para entender y responder en lenguaje natural ha mejorado tanto que interactuar con ellos es casi como hablar con otra persona real. Esta evolución se debe a los constantes avances en el procesamiento del habla y a la recopilación continua de datos.

Además del avance en el procesamiento del lenguaje natural, la inteligencia artificial ha experimentado un crecimiento significativo en el reconocimiento y análisis de imágenes, lo que ha permitido a grandes empresas desarrollar soluciones innovadoras en múltiples sectores. Google ha implementado la IA en Google Photos y Google Lens, lo que permite identificar objetos, textos y rostros en imágenes con gran precisión. Amazon, con su servicio Rekognition, ha optimizado la identificación de productos, rostros y contenido en imágenes y vídeos, beneficiando tanto a empresas como a organismos de seguridad. Microsoft, a través de su iniciativa AI for Good Lab, ha desarrollado MegaDetector, un modelo especializado en la detección de fauna en imágenes de fototrampeo, ampliamente utilizado en proyectos de conservación, disponible en su repositorio Pytorch-Wildlife¹⁷ disponible en la plataforma GitHub [17].

Aprovechando estos avances, este trabajo propone una solución que combina la potencia de *MegaDetector* como preprocesador de imágenes con una red neuronal convolucional basada en la arquitectura *EfficientNet-B5*, optimizada mediante *Transfer Learning* con pesos preentrenados en *ImageNet*. *MegaDetector* ha demostrado su gran eficacia en la identificación de fauna en imágenes de foto-trampeo. Por otro lado, *EfficientNet-B5* ha recibido el reconocimiento generalizado de la comunidad científica por su capacidad para mejorar la precisión en la clasificación de imágenes y optimizar el uso de los recursos computacionales.

La integración de estos enfoques optimiza el proceso de clasificación de imágenes de fototrampeo, ya que mejora la precisión en la detección de animales y reduce la carga computacional del sistema. En este trabajo, *MegaDetector* se utilizará como un preprocesador, cuyo objetivo principal será identificar y descartar las zonas de la imagen que carecen de interés, de modo que la red *EfficientNet-B5* se centre únicamente en las regiones relevantes donde puede haber fauna. De esta manera, la CNN recibirá imágenes depuradas y optimizadas, lo que evitara el ruido innecesario en el entrenamiento y mejorará su capacidad para clasificar correctamente las imágenes como “con presencia animal” o “sin presencia animal/vacía”. A diferencia de otros enfoques, en los que la red neuronal debe analizar la imagen completa, este método facilita una mejor extracción de características, lo que optimiza tanto el rendimiento como la eficiencia del modelo.

¹⁷ Enlace del repositorio: <https://github.com/microsoft/CameraTraps/>

En estudios previos, se ha reconocido el preprocesamiento de imágenes para resaltar la región de interés como un paso fundamental en el rendimiento de las redes neuronales convolucionales. Sin embargo, muchos de los enfoques existentes realizan esta segmentación manualmente, como en el caso del trabajo de identificación de especies de roedores [18], lo que implica un proceso laborioso y poco escalable. Otros enfoques recurren a una CNN entrenada específicamente para la detección de regiones de interés, como en el estudio de identificación y conteo de animales salvajes [19], lo que implica una fase adicional de entrenamiento que incrementa la complejidad computacional.

A diferencia de estas aproximaciones, este trabajo propone la automatización del preprocesamiento mediante el uso de *MegaDetector*, un modelo previamente entrenado para la detección de animales en imágenes de foto-trampeo. Esto permite filtrar automáticamente las zonas irrelevantes de las imágenes y proporcionar únicamente la información relevante para el entrenamiento de la CNN a utilizar. Como resultado, se reduce significativamente la carga computacional y se optimiza el uso de recursos al evitar la necesidad de entrenar una CNN adicional para la detección. Además, la red clasificatoria se entrena mediante *transfer learning*, lo que acelera el proceso de convergencia y mejora la generalización del modelo con menos datos y en menos tiempo.

La ventaja clave del transfer learning consiste en que los modelos preentrenados ya han aprendido representaciones generales de las características visuales (bordes, texturas, formas) a partir de grandes volúmenes de datos, como *ImageNet*, lo que permite reutilizar ese conocimiento y adaptarlo a una tarea específica con un ajuste mínimo, lo que se conoce como *Fine Tuning*, argumento expuesto en muchos artículos de la comunidad [20]. Esto no solo reduce drásticamente el tiempo y los recursos computacionales necesarios para el entrenamiento, sino que también mejora la capacidad de generalización y evita problemas cuando se trabaja con conjuntos de datos más pequeños o especializados.

En conjunto, esta metodología supone una mejora respecto a los enfoques tradicionales, ya que automatiza el preprocesamiento, optimiza el flujo de entrenamiento y reduce los requerimientos computacionales sin comprometer la precisión de la clasificación. Esta innovación facilita la implementación de modelos de detección de fauna más escalables y eficientes, lo que permite su aplicación en estudios de biodiversidad de manera más accesible y reproducible.

Capítulo 3

Materiales

En el presente capítulo se aborda la descripción de los recursos empleados durante el desarrollo del trabajo, iniciando con la base de datos que sirve como punto de acceso al modelo, detallando su procedencia, clases disponibles y distribución de instancias. Posteriormente, se especifica la división de los datos en conjuntos de entrenamiento, validación y prueba, así como los criterios seguidos para dicha partición. Por último, se enumeran las métricas de evaluación empleadas para valorar el rendimiento del modelo propuesto, tanto durante la fase de entrenamiento como en los experimentos finales.

3.1 Base de datos

Para la implementación de este modelo, capaz de clasificar imágenes con o sin presencia animal, se ha proporcionado un conjunto de datos compuesto por imágenes de foto-trampeo previamente clasificadas.

Sin embargo, esta categorización inicial no se alineaba con nuestro objetivo, ya que se basaba en la clasificación taxonómica de las especies presentes en las imágenes. Por consiguiente, fue necesario realizar una adaptación minuciosa para ajustarla a las características específicas de nuestro caso de estudio.

Originalmente, se nos proporcionó un conjunto de datos que contenía 19 clases taxonómicas distintas, sumando un total de 31670 imágenes, como se muestra en la Figura 10. Sin embargo, dado que nuestro problema de clasificación se basa en una dicotomía binaria, mantendremos la clase denominada "Vacía" para indicar la ausencia de animales en la imagen. Por otro lado, fusionaremos las demás clases, correspondientes a diferentes especies, para formar una única categoría llamada "Animal". Como se evidencia en la Figura 11, se obtiene la proporción de aproximadamente 30% del conjunto de imágenes con ausencia de animales y un 70% con

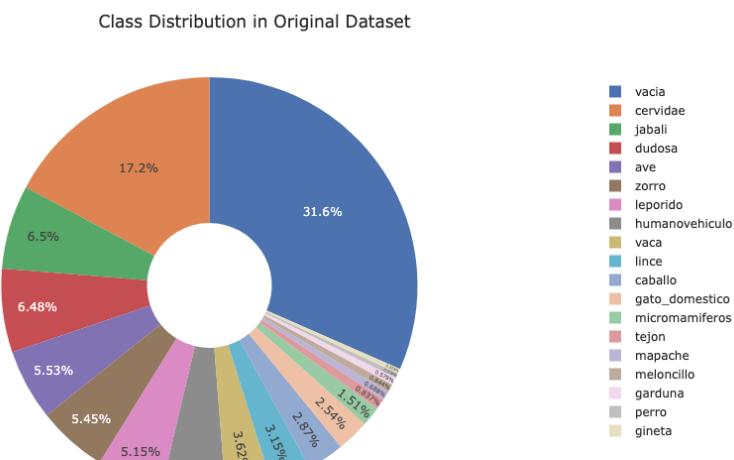


Figura 10: Distribución de las clases en el dataset original

presencia. Estos datos ya nos adelantan que tendremos que trabajar con técnicas que tengan en cuenta un desbalance entre las clases.

Class Distribution (Empty VS Animal) in Original Dataset

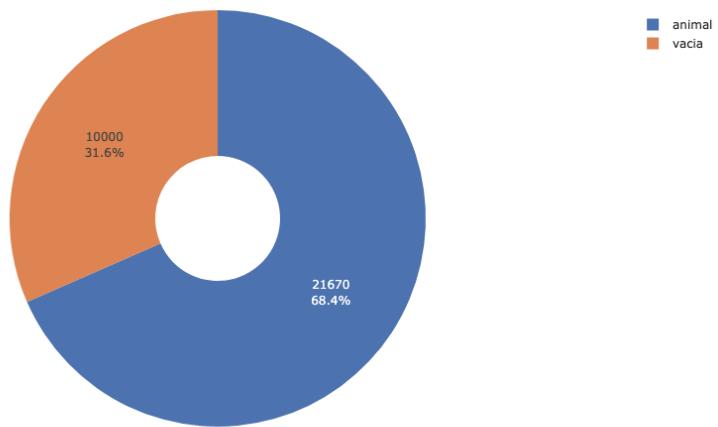


Figura 11: Distribución binaria de dataset adaptado

Mediante el análisis de la composición de las imágenes, se evidencia la presencia de una diversidad de elementos. En primer lugar, se observa que algunas de las imágenes fueron capturadas durante el día (como la Figura 14), en las cuales el animal ocupa una porción significativa del plano. En segundo lugar, se identifica que, en aquellas imágenes capturadas durante el día, puede ser difícil discernir la ubicación del animal, ya sea debido a su camuflaje en el entorno (como se muestra en la Figura 13) o a la comparación de su tamaño con el encuadre del plano (como se observa en la Figura 12).



Figura 14: Ejemplo de imagen diurna con animal



Figura 13: Ejemplo de imagen diurna con animal camuflado resaltado



Figura 12: Ejemplo de foto diurna con presencia animal y encuadre muy abierto

En oposición a las imágenes previamente mencionadas, se presentan aquellas que exhiben condiciones diametralmente opuestas. Dichas imágenes fueron capturadas durante el período nocturno, lo que implica que fueron tomadas en condiciones de baja luminosidad y, por consiguiente, con una marcada ausencia de color en la fotografía. En este tipo de imágenes se observan situaciones similares al caso anterior, donde el animal puede cubrir una porción significativa del encuadre (ver Figura 16) o también animales camuflados en el entorno nocturno (caso de la Figura 15Figura 16).



Figura 16: Imagen nocturna con presencia animal



Figura 15: Imagen nocturna con animal camuflado y resaltado

La disponibilidad de un conjunto de datos de foto-trampeo que abarque una amplia gama de situaciones, composiciones, condiciones ambientales y tipos de terreno constituye una ventaja significativa en el entrenamiento de modelos de aprendizaje profundo. Además, la presencia de diferentes niveles de iluminación, ángulos de captura, fondos y especies animales permitirá al modelo aprender patrones más robustos y desarrollar una mayor capacidad para enfrentarse a casos complejos.

No obstante, durante el análisis de las diferentes casuísticas que acabamos de exponer, se observó que las imágenes pertenecientes a la clase original denominada como “dudosa”, suponían unas situaciones extremas y, por tanto, demasiado complejas para que un ser humano pudiera catalogarlas, de hecho, el propio nombre de la clase lo autodescribe. Dichos hallazgos apuntan hacia la posibilidad de que surgieran problemas relacionados con su uso, por lo que su implementación ha sido descartada.

Además de la eliminación de la clase dudosa, se observó la presencia de rutas de imágenes duplicadas, por lo que se llevó a cabo una meticulosa limpieza, teniendo como resultado una colección con un total de 28567 imágenes, distribuidas según se muestra en la Figura 17.

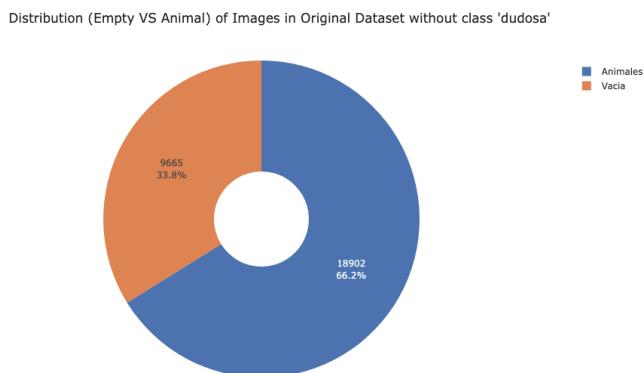


Figura 17: Distribución binaria del dataset final

3.2 Conjuntos de datos

Una vez concluido el análisis de los datos obtenidos en el paso anterior, se procede a la división del conjunto de datos en tres subconjuntos. Dividir el conjunto de datos en tres subconjuntos es una práctica fundamental en el desarrollo de modelos de redes neuronales convolucionales. Si bien se recomienda como mínimo la división de dos subconjuntos, hacerlo en tres permite una mejor evaluación exhaustiva y objetiva del rendimiento del modelo, contribuyendo así a su optimización y mejora continua.

Como se explica con mayor detalle en la sección 2.2.1 Etapa de entrenamiento del anexo teórico, el subconjunto de entrenamiento se emplea para ajustar los pesos de la red durante el proceso de aprendizaje. Por su parte, el de validación supervisa el comportamiento del modelo durante el entrenamiento y permite tomar decisiones como el uso de *early stopping*, la elección de hiperparámetros o la detección de posibles problemas de sobreajuste. Por último, el conjunto de prueba proporciona una evaluación final, imparcial e independiente del modelo ya entrenado, midiendo su capacidad real de generalización sobre datos nunca antes vistos.

La implementación de este procedimiento garantiza la integridad de los resultados obtenidos, al evitar la influencia de los datos empleados durante el proceso de entrenamiento. De este modo, se consigue una representación más precisa del rendimiento esperado en escenarios del mundo real.

En nuestro caso, se implementó una estrategia de subdivisión que asignó el 70 % de las imágenes al subconjunto de train, el 15 % al de validation y el 15 % restante al de test. En el proceso de división, se preservó la proporción de imágenes con y sin presencia animal en los conjuntos de validation y test, manteniendo la misma distribución observada en el conjunto de datos completo.

Tabla 1: Número de imágenes de cada clase en cada subconjunto de datos

Conjunto	Porcentaje de imágenes disponibles	Número total de imágenes	Número de imágenes con presencia animal	Número de imágenes vacías
Entrenamiento	70%	19.995	13.230	6.765
Validación	15%	4.286	2.836	1.450
Prueba	15%	4.286	2.836	1.450

Se evidencia una discrepancia significativa en los resultados, con una predominancia de casos positivos que alcanzan aproximadamente el 66%, en contraste con los casos negativos que representan el 34% del total.

3.3 Métricas de evaluación

Con el propósito de evaluar la eficacia del modelo de clasificación binaria propuesto, se ha implementado un conjunto de métricas que facilitan la observación de su comportamiento desde múltiples perspectivas.

- **Accuracy:** mide el porcentaje global de predicciones correctas.
- **Loss:** refleja el error durante el entrenamiento y la validación
- **Precision:** ofrece información más detallada sobre la calidad de las predicciones positivas, indica la proporción de predicciones positivas que resultaron ser correctas.
- **Recall:** también ofrece información detallada sobre la calidad de las predicciones positivas, en este caso, la proporción de instancias positivas reales que fueron identificadas correctamente.
- **F1-Score:** integra ambas métricas (precision y recall) en una única medida equilibrada.
- **FN rate:** tasa de falsos negativos.
- **FP rate:** tasa de falsos positivos.
- **AUC:** Área Bajo la Curva ROC evalúa la capacidad del modelo para distinguir entre clases.
- **Curva ROC:** complementada con la métrica anterior, en este caso ilustra gráficamente la capacidad de distinguir entre clases.
- **Matriz de confusión:** para visualizar de forma detallada la distribución de aciertos y errores en las predicciones.

Todas estas métricas se encuentran mejor definidas en el apartado 2.2.1 Etapa de entrenamiento del anexo teórico.

Capítulo 4

Metodología

En el presente capítulo, se procederá a la exposición de la metodología empleada en el desarrollo del proyecto, así como las distintas fases que han sido planteadas y las arquitecturas implementadas para la implementación del modelo, con el propósito de alcanzar el objetivo propuesto.

Cuando nos enfrentamos a un problema como el presente, podemos recurrir a una serie de herramientas, y, en la actualidad, gracias a los avances continuos en el ámbito de la inteligencia artificial, disponemos de un amplio abanico de ellas. En el proceso de planteamiento de la solución al problema en cuestión, se propuso el empleo de la herramienta MegaDetector como etapa de preprocesamiento de las imágenes de entrada de la CNN a implementar, y la posterior comparación de los resultados obtenidos con una CNN implementada sin la asistencia de dicha herramienta, con el objetivo de realizar un análisis exhaustivo de las ventajas y desventajas de ambos enfoques. Por consiguiente, a lo largo del proyecto se han gestionado tres arquitecturas divergentes.

Para una mejor comprensión del resto del presente apartado, se recomienda la consulta del anexo teórico completo, especialmente la sección Apartado 3

Redes Neuronales Convolucionales, en la que se exponen los conceptos teóricos que serán empleados a partir de este momento.

4.1 Arquitecturas de la red

Como se ha mencionado anteriormente, en el marco de este proyecto se gestionan tres arquitecturas distintas. La primera de ellas se basa en una CNN de clasificación binaria, fundamentada en la arquitectura EfficientNet-B5, la cual se ha detallado en el apartado 3.4 Arquitectura utilizada en este trabajo.

Originalmente, esta arquitectura ha sido concebida para clasificar un total de 1000 clases, lo cual, a priori, no la hace pertinente para la problemática que nos ocupa. Por consiguiente, se ha visto en la necesidad de realizar una serie de modificaciones, que ha implicado la eliminación de lo que se denomina “cabeza de red”.

El procedimiento descrito se centra en la eliminación de la capa densa con función de activación *softmax*, cuya descripción detallada puede encontrarse en la sección 2.1.2 Funciones de activación. Esta capa densa ha sido diseñada con el propósito de clasificar las 1000 clases de ImageNet. Al eliminar esta capa, se prescinde de la parte encargada de la clasificación, conservando únicamente la parte convolucional, capaz de extraer las características. Para adaptar este enfoque al caso en cuestión, se ha introducido una nueva capa densa con una única salida y una función de activación sigmoide, ampliamente utilizada en aplicaciones de clasificación binaria. Con el propósito de facilitar la asistencia y la claridad durante el desarrollo del proyecto, se ha asignado la denominación *MegaClassifier_c* a esta arquitectura.

Previamente al procesamiento por parte de la arquitectura EfficientNet-B5, las imágenes deben adaptarse a las condiciones de entrada requeridas por el modelo. Este proceso de adaptación se fundamenta en dos pasos cruciales: en primer lugar, se requiere que todas las imágenes posean un tamaño uniforme de 456x456 píxeles, ya que este es el tamaño de entrada para el que EfficientNet-B5 fue diseñado y preentrenado. En segundo lugar, los valores de los píxeles deben ser normalizados siguiendo el mismo esquema de preprocesamiento que se utilizó durante su entrenamiento con el conjunto de datos ImageNet. Este preprocesamiento asegura la coherencia estadística de los datos de entrada.

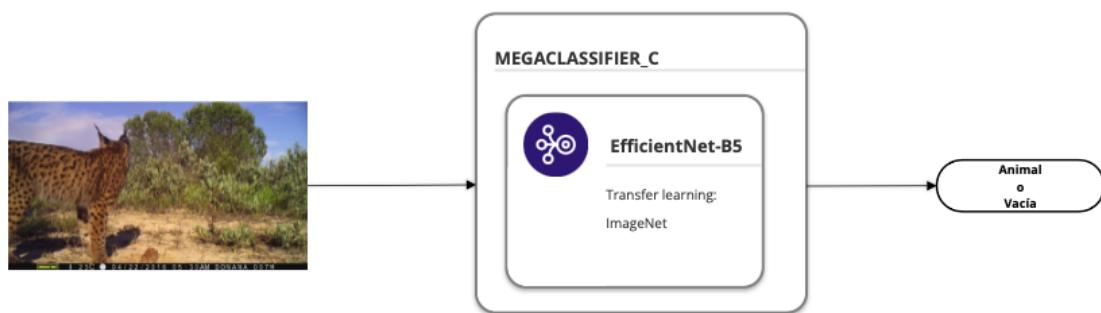
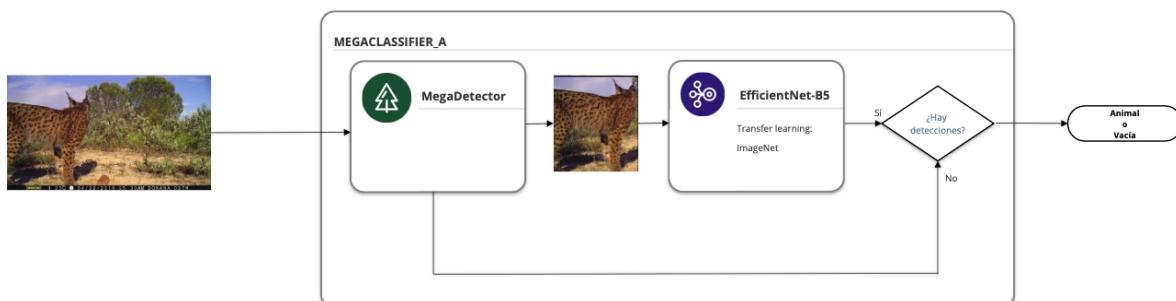


Figura 18: Esquema simplificado modelo *MegaClassifier_c*

Las arquitecturas restantes han sido diseñadas mediante la implementación de *MegaDetector*. Las disparidades entre ellas se manifiestan en la manera en que aprovechan el conocimiento aportado por dicho modelo al flujo de trabajo.

La primera de estas arquitecturas ha sido nombrada como *MegaClassifier_a*. El procesamiento de las imágenes de entrada es realizado por el componente denominado *MegaDetector*, el cual se encarga de detectar la posible zona donde se encuentra el animal en caso de que su presencia sea verificada. Como se detalla en el Apartado 4

MegaDetector, la forma de funcionamiento de este modelo es que, a partir de una imagen dada, en caso de detectar la presencia de un animal, genera como salida las coordenadas del rectángulo de menor área que lo rodea, lo que es conocido en inglés como "bounding box". Mediante un procesamiento de estos datos, que será descrito en los siguientes apartados, es posible generar una imagen reducida que contenga únicamente la zona de interés,



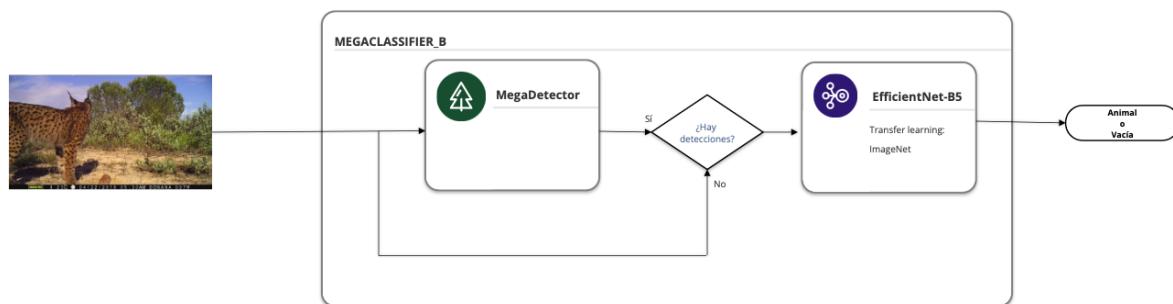
en este caso, la región interna del “bounding box”.

La clave del flujo de esta arquitectura reside en el criterio aplicado según la salida de MegaDetector. En ausencia de detecciones, el modelo clasifica el caso como “clase vacía”, ya que de realizar el procesado de los datos, la imagen resultante sería completamente en negro. Por el contrario, en presencia de detecciones, se implementa el procesamiento de datos para generar la imagen optimizada, la cual constituye la entrada para una red neuronal convolucional implementada igual que hemos expuesto en la arquitectura *MegaClassifier_c*. La salida de esta red constituye la respuesta final para este tipo de casuísticas. De este modo, el conocimiento de MegaDetector es utilizado para realizar una clasificación previa y entrenaremos la red para

*Figura 19: Esquema simplificado modelo *MegaClassifier_a**

clasificar aquellos casos en los que el modelo anterior “no lo tuvo tan claro”.

En tercer lugar, se ha implementado la arquitectura *MegaClassifier_b*. Esta arquitectura es análoga a la precedente, con la salvedad de que, en los supuestos en los que la imagen de entrada del sistema MegaDetector no genera detecciones, se emplea dicha imagen como entrada de manera directa a la CNN. En consecuencia, la función de MegaDetector es íntegramente la de un preprocesador de las imágenes de entrada de la red neuronal convolucional.



*Figura 20: Esquema simplificado modelo *MegaClassifier_b**

4.2 Hiperparámetros

Como se ha explicado en la sección 2.2.1.2 Inicialización de los pesos, a la hora de realizar un entrenamiento, se requiere la inicialización de los pesos. Entre las diversas técnicas presentadas en el apartado anterior, se ha seleccionado la opción de *transfer learning*.

Esta técnica se fundamenta en la reutilización del conocimiento adquirido por un modelo previamente entrenado en una tarea amplia y general, como se evidencia en el caso de la clasificación del conjunto de datos ImageNet. Al aprovechar los pesos preentrenados, el modelo puede iniciar con una representación visual robusta, lo que reduce el tiempo de entrenamiento y el riesgo de sobreajuste, y facilita una convergencia más rápida hacia una solución efectiva en la tarea objetivo.

En el contexto de facilitar la implementación de este proyecto como una aproximación inicial para abordar desafíos relacionados con el uso de CNN o como un referente para la estructuración

de un proyecto de aprendizaje profundo, se determinó preservar una metodología de versionado que permitiera, en cada iteración, la selección u optimización de alguno de los hiperparámetros.

4.2.1 Umbral de MegaDetector

Como se menciona en el Apartado 4

MegaDetector, MegaDetector trabaja con un umbral de confianza para considerar la generación de las detecciones. Por consiguiente, este constituye el primer hiperparámetro que ha sido objeto de ajuste conforme a un criterio establecido.

Se ha implementado el uso de un umbral bajo en MegaDetector, permitiendo así la cobertura de todos los ejemplos positivos del conjunto de entrenamiento. Este umbral generalista prioriza la sensibilidad sobre la especificidad en la primera fase del flujo de trabajo. Al implementar este criterio, tenemos que asumir que el número de falsos positivos aumenta significativamente, pero esto nos resulta ventajoso, dado que en esta etapa el objetivo es garantizar que ningún ejemplo sea omitido durante la detección.

Este enfoque permite que nuestra red neuronal convolucional se especialice en la discriminación de las características distintivas de los verdaderos positivos, lo que resulta en una mejora significativa en la precisión del sistema. Este enfoque integra la alta sensibilidad de MegaDetector con la capacidad de refinamiento que se busca obtener mediante la CNN.

4.2.2 Versión 1: Batch size

En esta primera versión del entreno de nuestra CNN, se implementa un proceso de prueba que abarca la evaluación de diferentes tamaños de lote. Los valores que se han sometido a prueba comprenden de [16, 32, 64, 128], siendo estos valores comunes y ampliamente utilizados.

Como se explica en la sección 2.2.2.4 Batch Size, la selección óptima del tamaño del conjunto de datos es crucial para el rendimiento de una red neuronal convolucional, ya que determina el equilibrio entre la precisión y la eficiencia computacional. Un valor reducido, como 16, puede ofrecer una mejor generalización y escapar de los mínimos locales, mientras que valores mayores, como 64 o 128, aceleran el entrenamiento y estabilizan el gradiente. El objetivo es identificar el punto de equilibrio entre la velocidad de entrenamiento y la calidad del aprendizaje, de modo que se maximice el rendimiento del modelo.

4.2.3 Versión 2: Algoritmo de optimización

La elección del optimizador más apropiado para una CNN constituye una determinación de suma importancia que incide de manera directa en la convergencia, la estabilidad y el

desempeño final del modelo. Los algoritmos probados han sido Adam, RMSprop y SGD, basándose en las diferentes ventajas que ofrecen: Adam se erige como una entidad que fusiona de manera sinérgica los atributos del momento adaptativo y la normalización del gradiente, logrando así un equilibrio dinámico entre la celeridad y la exactitud; RMSprop se distingue por su capacidad para gestionar gradientes cambiantes mediante tasas de aprendizaje adaptativas; por otro lado, a pesar de su aparente simplicidad, SGD puede alcanzar mejores mínimos globales en ciertos escenarios.

4.2.4 Versión 3: Función de pérdida

Con el propósito de optimizar el desempeño del modelo y adaptarlo de manera efectiva al desequilibrio presente en el conjunto de datos, se ha determinado llevar a cabo experimentos con diversas funciones de pérdida durante el proceso de entrenamiento. Además de la función de pérdida binaria estándar *BinaryCrossentropy*, se ha incorporado el uso de *BinaryFocalCrossentropy*, reconocida por su capacidad para enfocarse en ejemplos complejos y mitigar el impacto del desequilibrio entre las clases. Asimismo, se ha evaluado el uso de *BinaryCrossentropy* en conjunto con un ajuste de pesos por clase, parámetro conocido como *class_weight*, lo cual permite penalizar más los errores en la clase minoritaria. Esta estrategia comparativa tiene como objetivo determinar cuál de estas opciones ofrece el mejor equilibrio entre precisión y sensibilidad, maximizando así el rendimiento del modelo en un entorno de clasificación binaria desequilibrada.

4.2.5 Versión 4: Hiperparámetros de optimización y pérdida

Con el fin de maximizar el rendimiento del modelo, resulta fundamental realizar una adecuada selección de los hiperparámetros que controlan el proceso de entrenamiento. En este trabajo, se ha prestado especial atención al valor del *learning rate* del optimizador, ya que este parámetro constituye un elemento determinante en la velocidad de aprendizaje del modelo y puede incidir de manera directa en la estabilidad y la capacidad de convergencia. Además, se ha seleccionado la función de pérdida *BinaryFocalCrossentropy*, que ha demostrado ser la más efectiva en todas las arquitecturas, y se han incluido como hiperparámetros a optimizar sus dos componentes clave: *alpha*, que controla la ponderación entre clases, y *gamma*, que regula el enfoque del modelo hacia los ejemplos más difíciles. La optimización de estos parámetros resulta imperativa para la construcción de modelos más robustos, precisos y adaptados a la naturaleza del problema en cuestión.

4.2.6 Versión 5: Data augmentation

En la versión 5 del modelo se ha implementado el uso de técnicas de augmentation de datos con el propósito de incrementar la capacidad de generalización del modelo y minimizar el riesgo de sobreajuste. Este procedimiento implica la aplicación de transformaciones aleatorias a las imágenes utilizadas para el entrenamiento, tales como giros, variaciones de brillo, recortes o escalados. De esta manera, se generan nuevas versiones sintéticas de las muestras originales. La implementación de un conjunto de datos más heterogéneo durante el entrenamiento fomenta

el aprendizaje de características más robustas e invariantes por parte del modelo. Esta capacidad resulta particularmente beneficiosa en aplicaciones como el fototrampeo, donde las condiciones de iluminación, ángulos y fondos pueden presentar variaciones significativas. En consecuencia, se evidencia un incremento en la capacidad del modelo para adaptarse a nuevas imágenes sin perder precisión.

4.2.7 Versión 6: Early stopping

En la versión 6 del modelo se ha implementado la técnica de early stopping con el propósito de evitar el sobreentrenamiento y seleccionar el punto óptimo del aprendizaje antes de que el modelo comience a degradar su rendimiento en el conjunto de validación. Para esta versión, se han explorado diversos criterios de parada temprana, incluyendo val_loss, val_auc, val_precision y val_recall. El uso de val_loss permite identificar el momento en que el modelo deja de minimizar el error global, mientras que val_auc ofrece una visión más completa del comportamiento del modelo en términos de separabilidad entre clases, lo cual resulta especialmente útil en problemas con cierto desequilibrio. Además, se han considerado las métricas val_precision y val_recall, dada su importancia en tareas de clasificación binaria como la presente, donde es esencial reducir tanto los falsos positivos como los falsos negativos. La comparación del comportamiento del early stopping en función de estas métricas permite la elección del enfoque más alineado con los objetivos del problema.

4.2.8 Versión 7: Fine tuning

En la versión 7 del modelo se implementó la técnica de *fine-tuning*, un paso fundamental en el contexto del *transfer learning*, que consiste en ajustar los pesos de las capas previamente entrenadas para adaptarlas mejor a la tarea específica del proyecto. Para ello, se ha evaluado el rendimiento del modelo al descongelar distintos bloques de capas de la arquitectura EfficientNet-B5: 20, 40, 80 capas y, finalmente, la totalidad del modelo. Esta progresión escalonada permite observar el impacto que tiene la liberación de distintos niveles de abstracción en la red, desde características más generales (como bordes y texturas) hasta otras más específicas. El propósito es hallar un equilibrio entre la capacidad de aprendizaje y el riesgo de sobreajuste, ya que al desbloquear más capas se incrementa el número de parámetros entrenables y, en consecuencia, la complejidad del modelo. Esta estrategia permite maximizar la especialización del modelo en el dominio concreto del conjunto de datos, partiendo de una base robusta ya preentrenada con ImageNet.

A partir de esta versión, los modelos alcanzan un nivel de rendimiento considerablemente alto, lo que motiva a refinar no solo su arquitectura o el proceso de entrenamiento, sino también el criterio de decisión final. Tradicionalmente, en tareas de clasificación binaria se ha empleado un umbral del 0,5 para convertir las probabilidades predichas en clases, asumiendo una distribución equilibrada y costes simétricos para los errores. No obstante, cuando el propósito es optimizar el rendimiento del modelo sobre un conjunto específico de métricas o priorizar ciertos tipos de aciertos o errores, este valor fijo puede no ser el más apropiado. En consecuencia, se propone una metodología más adaptativa, fundamentada en la geometría de la curva ROC, que permite la optimización del umbral mediante la distancia al punto (0,1). Este enfoque permite la

adecuación de la decisión del modelo a sus capacidades reales y al contexto del problema específico.

El cálculo de un umbral óptimo mediante la distancia al punto ideal (0,1) en la curva ROC constituye una técnica común para hallar el equilibrio entre la tasa de verdaderos positivos (True Positive Rate, TPR o recall) y la tasa de falsos positivos (False Positive Rate, FPR). El punto (0,1) representa el clasificador perfecto: 100% de verdaderos positivos y 0% de falsos positivos. La cercanía a este punto en la curva ROC indica un modelo con un mejor comportamiento. En consecuencia, se procede a la evaluación de diversos umbrales, seleccionando aquel que minimiza la distancia euclídea al punto (0,1). Este procedimiento permite alcanzar un equilibrio óptimo entre sensibilidad y especificidad. La fórmula empleada para calcular esta distancia es:

$$\text{distancia} = \sqrt{FPR^2 + (1 - TPR)^2}$$

Para cada punto de la curva ROC se calcula esta distancia, el umbral óptimo será aquel que minimice esta distancia.

4.3 Fase de entrenamiento

En este apartado se exponen las diversas fases de entrenamiento implementadas sobre las tres variantes propuestas del modelo, designadas como a, b y c. Cada una de estas variantes ha implementado una estrategia particular en lo que respecta al preprocesamiento de los datos, la configuración de la arquitectura base y las técnicas de entrenamiento aplicadas. Se explican las decisiones tomadas en cada versión (v1 a v7), desde la configuración inicial hasta la aplicación de técnicas avanzadas como el aumento de datos, detención temprana u optimización fina. En las variantes a y b, se justifica la elección del umbral aplicado a MegaDetector, una herramienta utilizada para filtrar o modificar imágenes según la presencia de animales detectada, lo cual ha influido directamente en la construcción del conjunto de entrenamiento. Finalmente, se presentan los resultados obtenidos durante el entrenamiento de cada versión, lo que permite analizar la evolución del rendimiento del modelo y el impacto de las diferentes decisiones tomadas.

Es preciso considerar que, en las primeras versiones del modelo, el enfoque se ha centrado en aislar y analizar de forma individual cada uno de los componentes clave del sistema, tales como el tamaño de lote, la función de pérdida o el ajuste de hiperparámetros, entre otros. El propósito de estas etapas iniciales no radicaba en optimizar directamente el rendimiento final, sino más bien en comprender el impacto de cada elemento en el comportamiento del modelo y establecer una base sólida para versiones posteriores. No es hasta la versión 5 cuando se integran las decisiones más prometedoras tomadas en versiones anteriores, combinando técnicas como el data augmentation o el uso de funciones de pérdida robustas, para construir un modelo más completo. A partir de este punto, el enfoque se centra en refinar el rendimiento global mediante mejoras acumulativas, tales como el uso de early stopping o el fine-tuning, lo que permite una evolución más coherente y sostenida en las métricas de evaluación.

4.3.1 Búsqueda de umbral de MegaDetector

La primera fase del entrenamiento se centró en la optimización del umbral de confianza de MegaDetector, el modelo de detección de objetos empleado como etapa preliminar de preprocesamiento para las variantes a y b. El propósito de este procedimiento fue determinar un valor de umbral lo suficientemente bajo como para asegurar que la totalidad de las imágenes pertenecientes a la clase positiva (aquellas que contienen animales, humanos y vehículos) en el conjunto de entrenamiento fueran debidamente detectadas, es decir, que se generara al menos una predicción sobre dichas imágenes. Esta determinación se fundamenta en la necesidad imperativa de asegurar que ningún ejemplo positivo relevante sea excluido del proceso de entrenamiento como resultado de un error en la detección previa. En consecuencia, si MegaDetector no detectara la presencia de un animal en una imagen positiva, esta podría ser excluida o mal procesada, lo que repercutiría negativamente en el aprendizaje del clasificador final. En consecuencia, se ha adoptado un umbral generalista que prioriza la cobertura sobre la precisión en esta etapa, asumiendo que el posible ruido puede ser corregido o compensado posteriormente durante el entrenamiento del modelo principal.

Como se ha mencionado en el Apartado 4

MegaDetector, en la documentación de Megadetector se pueden encontrar dos umbrales recomendados en función de las necesidades de cada caso: uno recomendado (*threshold* = 0.2) y otro más restrictivo (*threshold* = 0.05). A estos valores se añaden los umbrales [0.1, 0.003, 0.0015] a las pruebas, donde analizando el código del repositorio del modelo, descubrimos que el umbral con el que trabaja por defecto es 0.1.

Al examinar la Tabla 2, se evidencia que el umbral que ha logrado cumplir con el objetivo establecido es 0.0015.

Tabla 2: Cobertura de positivos de cada umbral de MegaDetector

Umbral	Positivos en el subconjunto de entrenamiento	Positivos detectados	Porcentaje de cobertura
0.2		13.052	98.65%
0.1		13.098	99.00%
0.05	13.230	13.136	99.29%
0.003		13.221	99.23%
0.0015		13.230	100%

Con base en los datos generados por la implementación de MegaDetector sobre el conjunto completo de datos con el umbral establecido, es factible generar métricas de referencia. En el caso de que el detector genere detecciones de una imagen con presencia animal, se puede contabilizar como un verdadero positivo; cuando no genera detecciones, se puede contabilizar

como un verdadero negativo; siguiendo con este criterio, cuando genera detecciones de una imagen vacía, se encuentra en el caso de un falso positivo y, por último, cuando no genera detecciones de una imagen con presencia animal, se puede contabilizar como un falso negativo.

En lo que respecta a este criterio, es factible la estimación de ciertas métricas presentadas en la sección 2.2.2.7 Métricas, del anexo teórico. A modo ilustrativo, se ha determinado la matriz de confusión ideal sobre el conjunto de entrenamiento y la generada por el umbral de cobertura 100%.

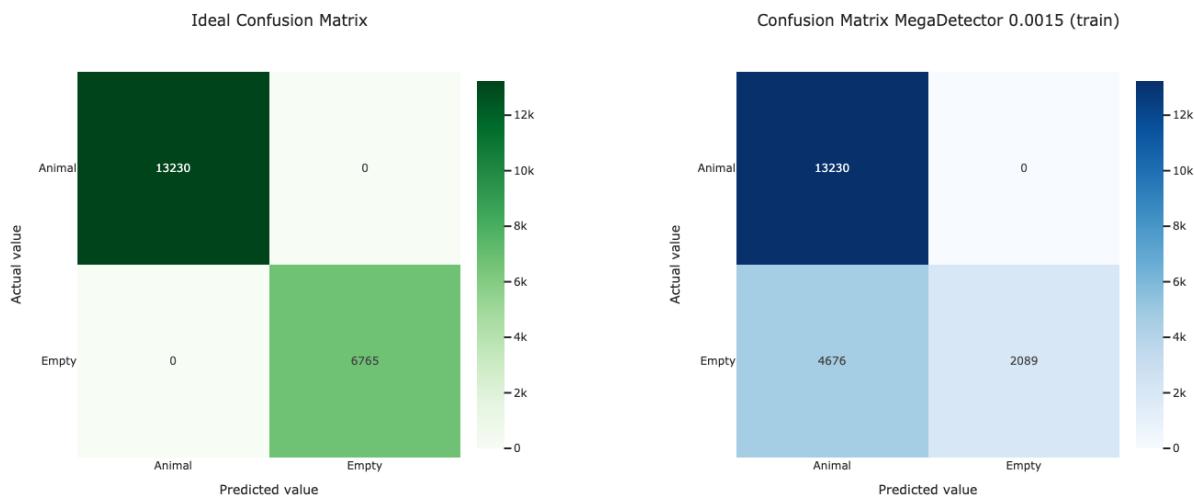


Figura 21: Matriz de confusión ideal VS matriz de confusión umbral=0.0015 sobre el conjunto de entreno

Como se ha mencionado anteriormente en las secciones precedentes, la implementación de un umbral tan bajo resultaría en una cantidad significativa de resultados positivos falsos, como se evidencia en la matriz de confusión correspondiente al conjunto de entrenamiento. Este número se procurará reducir con la CNN mediante los modelos a y b, sin alterar o perjudicar lo mínimo posible el número de falsos negativos.

4.3.2 Preprocesamiento de imágenes con MegaDetector

En el presente apartado se abordará la metodología empleada para la generación de una imagen optimizada a partir de los datos generados por MegaDetector sobre una imagen específica. Este procedimiento implica la eliminación de aquellas regiones de escaso interés, lo que resulta en la obtención de una imagen donde únicamente se muestran las detecciones o detecciones y los bordes ajustados a ellas.

Como se ha mencionado anteriormente, y se desarrolla con mayor profundidad en el Apartado 4

MegaDetector, MegaDetector generará una o varias detecciones sobre una imagen en forma de coordenadas del bounding box. A través de estas coordenadas, es posible construir una máscara binaria, entendida como una imagen en la que los píxeles dentro de la región de interés

se asignan el valor 1, mientras que los píxeles fuera de esta región tienen el valor 0. En el contexto de la codificación RGB, estos valores se representan como 255 y 0, respectivamente. La aplicación de esta máscara se lleva a cabo mediante la multiplicación elemento a elemento, denominada también como el producto de Hadamard:

$$C_{ij} = A_{ij} \cdot B_{ij} \rightarrow C = A \circ B$$

El resultado de esta operación será una imagen en la que los píxeles que se encuentren fuera de la región de interés tendrán un valor de 0, mientras que aquellos que se ubiquen dentro de dicha región mantendrán el valor de la imagen original.

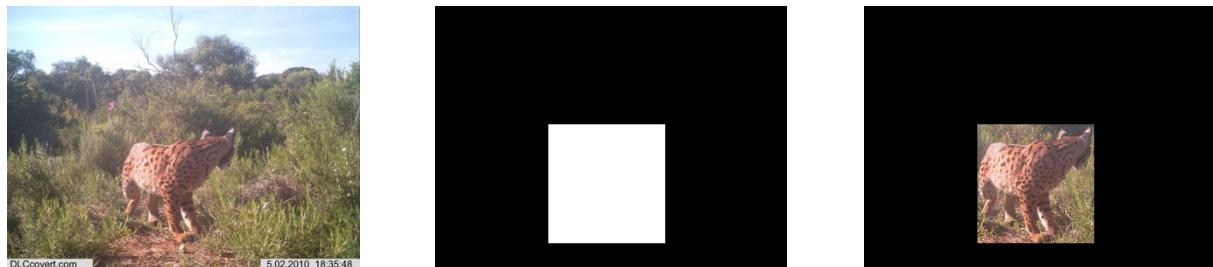


Figura 22: Ejemplo de generación de máscara binaria y su aplicación sobre una imagen con presencia animal

Como se evidencia en la Figura 22, la aplicación de una máscara binaria sobre la imagen original resulta en un resultado representativo. Como se puede observar, una gran parte del área restante queda ocupada por píxeles completamente negros, los cuales no aportan información relevante desde el punto de vista visual o semántico. Esta circunstancia manifiesta una deficiencia en la optimización del espacio de representación, dado que la región útil, correspondiente al objeto de interés, abarca únicamente una porción diminuta del área total de la imagen.

Para optimizar espacialmente la imagen enmascarada y centrarla únicamente sobre la región relevante, se ha diseñado un algoritmo simple pero eficaz que explota las propiedades estructurales de la máscara binaria. Dado que tanto la imagen enmascarada como su correspondiente máscara tienen las mismas dimensiones, es posible determinar el área mínima que contiene todas las detecciones analizando únicamente la máscara.

La máscara binaria está compuesta por valores 0 y 1, donde los unos indican presencia de objeto. A partir de esta propiedad, el algoritmo suma los valores a lo largo de cada fila y cada columna:

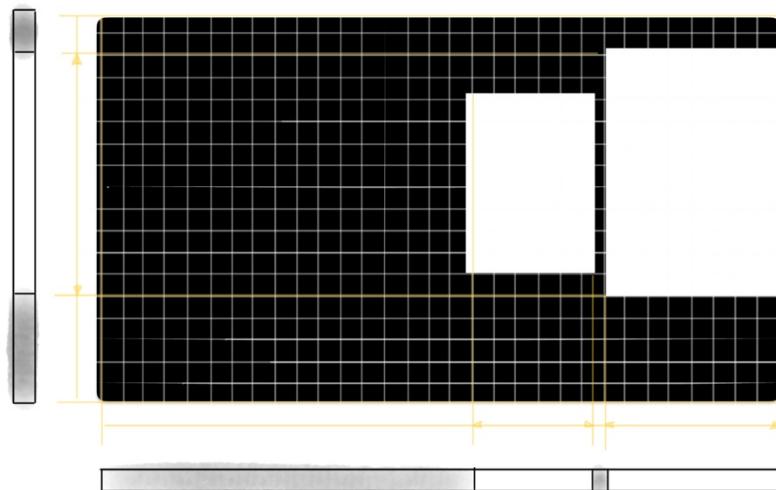


Figura 23: Explicación visual de la operación aplicada para el recorte ajustado

Si la suma horizontal de una fila es mayor que cero, significa que existe al menos un píxel activado en esa fila, por tanto, hay detección. De igual forma, si la suma vertical de una columna es mayor que cero, esa columna forma parte de una región activa.

Mediante este análisis, es posible determinar de forma precisa las filas y columnas donde comienzan y terminan las detecciones. A partir de esos índices, se genera un recorte ajustado que contiene únicamente la región útil de la imagen, minimizando así el espacio negro entre las detecciones y respecto a los bordes. Este proceso no solo mejora la eficiencia visual y computacional, sino que también permite al modelo centrarse en la información relevante, reduciendo el ruido y la redundancia espacial.

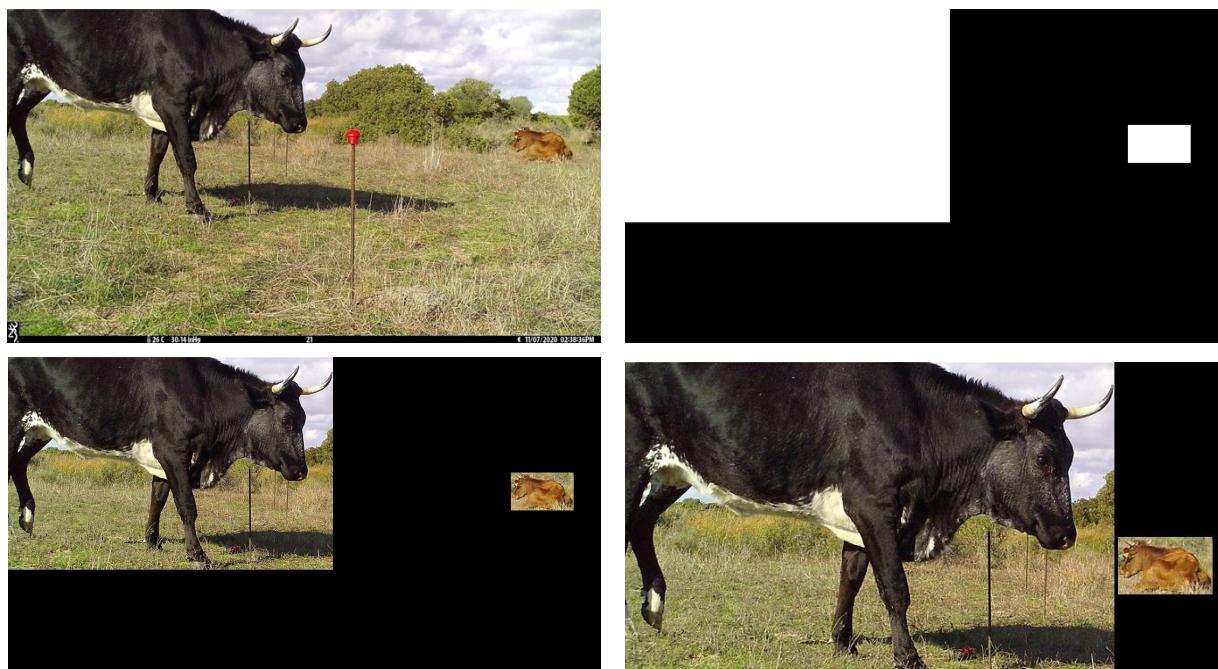


Figura 24: Ejemplo de preprocessado completo con MegaDetector sobre una imagen con presencia animal

4.3.3 Entrenamiento de MegaClassifier_a

A continuación, describimos el proceso de entrenamiento del modelo denominado MegaClassifier_a, que se ha implementado con la arquitectura descrita en la Figura 19. En este caso, las imágenes utilizadas para el entrenamiento y la validación son únicamente aquellas en las que MegaDetector generó detecciones, ya que, por el comportamiento de este modelo, los casos en los que MegaDetector no genera detecciones se clasifican como clase vacía.

En la versión 1, se evaluaron diferentes tamaños de lote [16, 32, 64 y 128] con el objetivo de determinar aquel que ofrecía un mejor equilibrio entre estabilidad en el entrenamiento y rendimiento en validación.

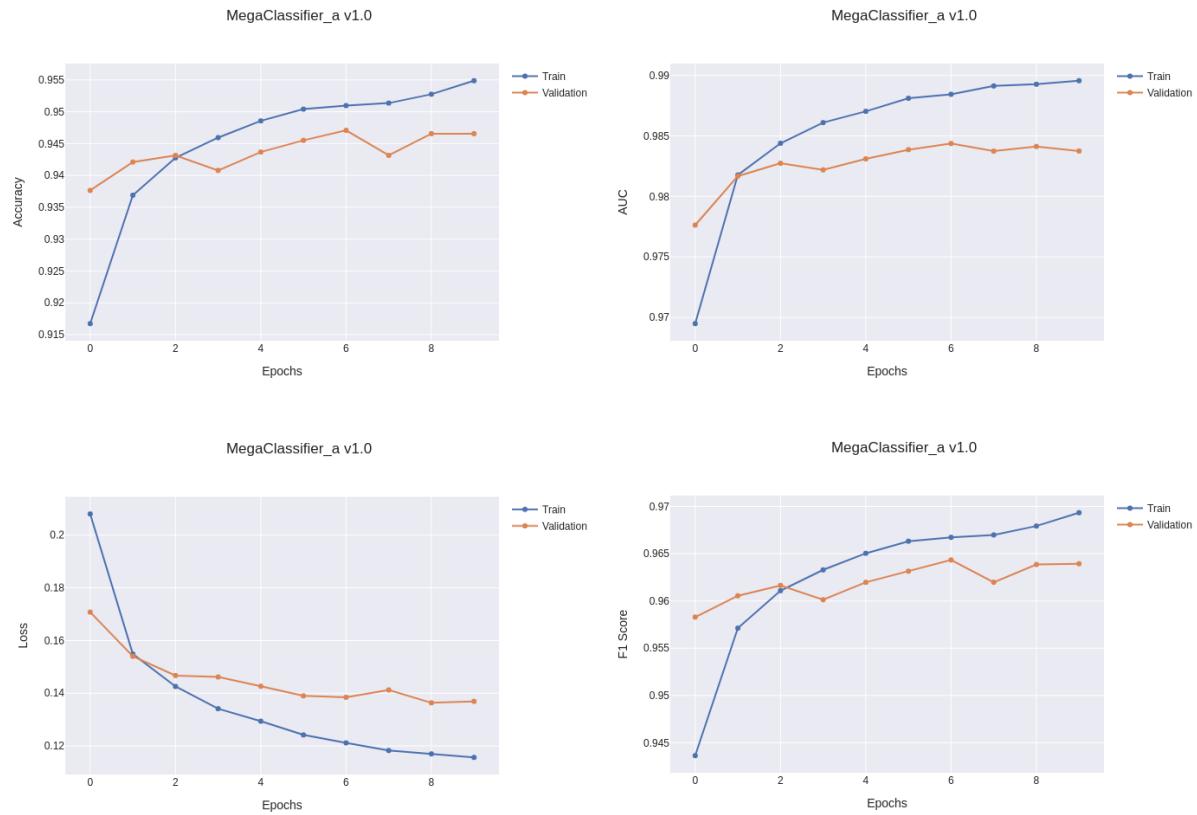


Figura 25: Gráficas de entrenamiento de accuracy, auc, los y f1-score de *MegaClassifier_a_v1.0*

En cuanto a las métricas de rendimiento, se aprecia que tanto la accuracy como la loss mantienen una evolución positiva a lo largo de las épocas: la training accuracy crece de forma sostenida hasta valores razonablemente altos rondando el 80%-90%, mientras que la validation accuracy sigue una tendencia similar, aunque un poco más moderada, lo que sugiere que el modelo generaliza bien aunque dejando un claro margen de mejora que abordaremos en siguientes versiones. Por su parte, el AUC se aproxima a valores cercanos a 0.9, reforzando la idea de que el modelo es capaz de separar adecuadamente las clases en el espacio de decisión; en la misma línea, el F1-score mantiene valores consistentemente elevados, lo que indica un equilibrio correcto entre la capacidad del modelo para identificar positivos y descartar negativos. En conjunto, estas métricas convergen y se mantienen estables, destacando la ventaja de trabajar con un batch_size de 16 frente a opciones mayores, sobre todo en la fase inicial de entrenamiento en la que aún no se han ajustado otros hiperparámetros del modelo.

En la versión 2 del modelo se exploró el impacto del optimizador en el rendimiento de la red, manteniendo constante la arquitectura y el tamaño de la agrupación seleccionado en la versión anterior (16). Para ello, se compararon tres optimizadores ampliamente utilizados en el entrenamiento de redes neuronales: Adam, RMSprop y SGD, todos con sus parámetros por defecto. El propósito de esta fase fue identificar qué estrategia de optimización favorece una convergencia más estable y métricas de evaluación más sólidas en este contexto de clasificación binaria.

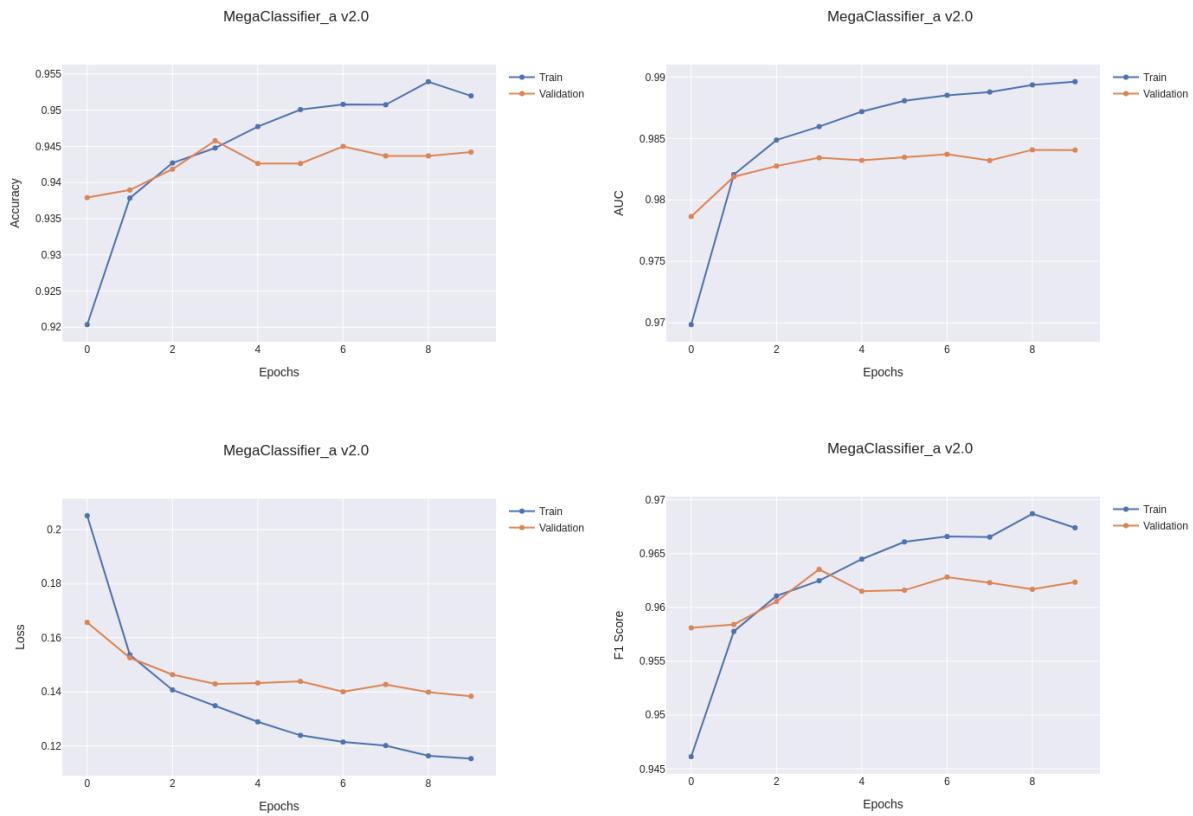


Figura 26: Gráficas de entrenamiento de accuracy, auc, los y f1-score de *MegaClassifier_a_v2.0*

En esta fase, al comparar Adam, RMSprop y SGD, se ha confirmado que Adam sigue ofreciendo los mejores resultados, manteniendo curvas de entrenamiento y validación muy similares a las de la versión anterior, en la que ya se utilizaba este optimizador. La ligera ventaja frente a los otros algoritmos queda reflejada en la estabilidad de las métricas y en una convergencia consistente, lo que justifica seguir empleando Adam como punto de partida para afinar el resto de los hiperparámetros en fases posteriores.

En la tercera versión del modelo, se ha abordado el problema de desequilibrio de clases presente en el conjunto de datos, un factor que puede afectar negativamente al rendimiento del modelo si no se aborda adecuadamente. Para abordar esta problemática, se ha mantenido la configuración base de la arquitectura, el optimizador (Adam) y el tamaño de la muestra (16), pero se han probado distintas funciones de pérdida más adecuadas para este tipo de escenarios. Además de la función estándar BinaryCrossentropy, se ha evaluado el uso de BinaryFocalCrossentropy, que penaliza con mayor intensidad los errores en clases minoritarias, y una versión ponderada de BinaryCrossentropy, aplicando pesos a las clases en función de su

frecuencia. El propósito de esta etapa ha sido mejorar la sensibilidad del modelo frente a la clase menos representada, manteniendo un equilibrio general adecuado entre precisión y recall.

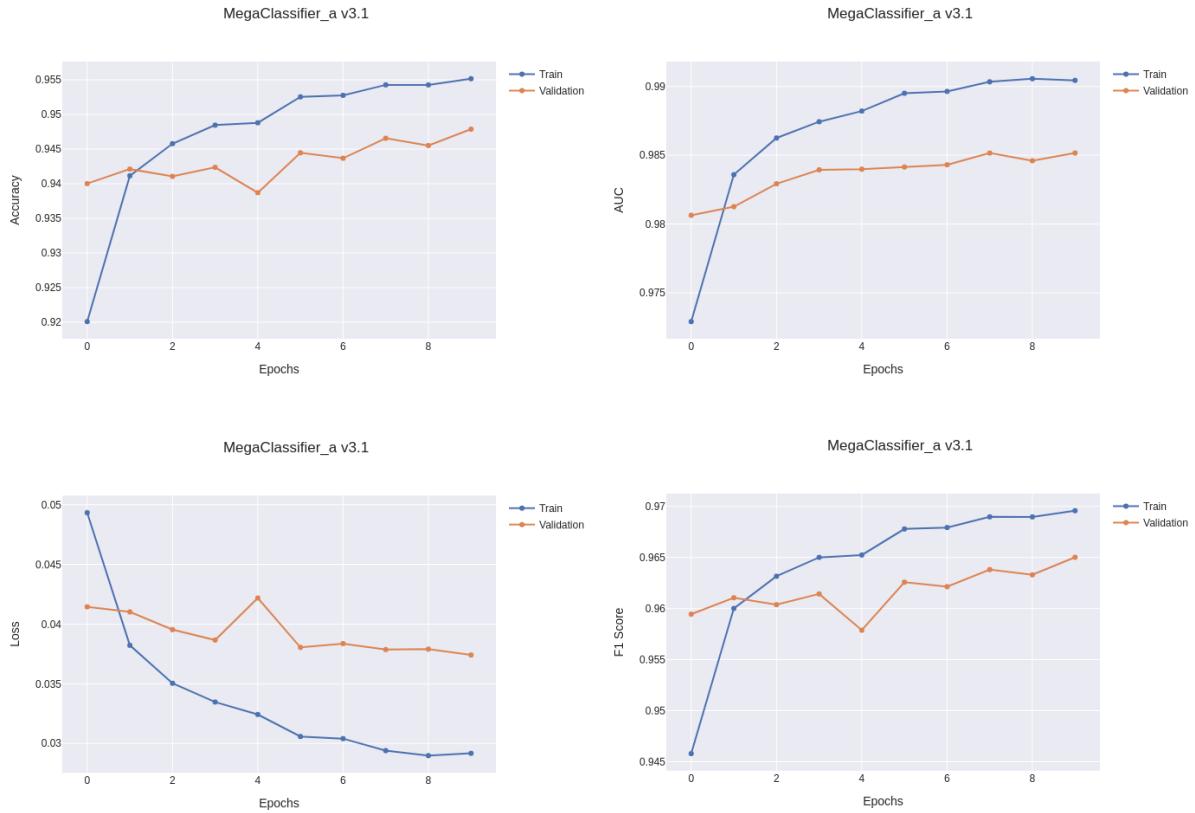


Figura 27: Gráficas de entrenamiento de accuracy, auc, los y f1-score de *MegaClassifier_a_v3.1*.

En la tercera versión se ha comprobado que BinaryFocalCrossentropy (v3.1) ofrece un mejor rendimiento en escenarios con desequilibrio de clases, superando tanto a la función de pérdida estándar (BinaryCrossentropy) como a la variante ponderada. Las gráficas de precisión, pérdida, AUC y puntuación F1 muestran que el modelo mantiene una progresión estable y, sobre todo, una mayor capacidad para reconocer la clase minoritaria sin perjudicar la precisión global. Esto se debe a que la focal loss penaliza con más intensidad los errores en ejemplos difíciles o menos representados, corrigiendo el sesgo natural que surge cuando la distribución de clases está desequilibrada.

En la versión 4 del modelo, se inicia el proceso de ajuste fino de hiperparámetros con el objetivo de maximizar el rendimiento de la configuración seleccionada como base en versiones anteriores. Teniendo en cuenta que tanto el optimizador Adam como la función de pérdida BinaryFocalCrossentropy han demostrado ser las opciones más efectivas, esta fase se centra en optimizar sus parámetros internos. En particular, se busca encontrar el valor óptimo del learning rate para Adam, así como los valores de alpha y gamma para BinaryFocalCrossentropy, que controlan, respectivamente, el peso asignado a la clase minoritaria y el grado de penalización a las predicciones incorrectas. Este procedimiento resulta fundamental para potenciar la capacidad de generalización del modelo y su sensibilidad en contextos con datos desequilibrados.

Para determinar un valor óptimo del learning rate sin recurrir a una búsqueda manual o por rejilla, se ha empleado la técnica conocida como Learning Rate Finder (LRFinder). Esta metodología se centra en el entrenamiento del modelo a lo largo de un número limitado de iteraciones, incrementando progresivamente el learning rate en cada iteración, iniciando desde un valor mínimo. Durante el proceso, se registra la función de pérdida a medida que varía el learning rate. Posteriormente, esta información se representa en una gráfica logarítmica (loss vs. learning rate) (ver Figura 28)¹⁸, en la cual se puede observar el punto a partir del cual la pérdida comienza a disminuir rápidamente y también aquel desde el cual empieza a divergir. El propósito de este análisis es identificar un rango en el que la pérdida disminuye de manera estable, seleccionando un valor dentro de este rango como learning rate óptimo. Esta técnica permite acelerar el entrenamiento y mejorar la convergencia del modelo, evitando tanto valores demasiado bajos que ralenticen el aprendizaje como valores altos que causen inestabilidad.

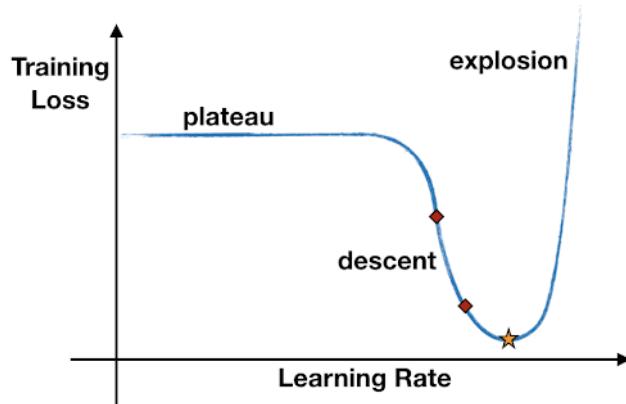


Figura 28: Gráfica teórica de LRFinder

Tal y como se aprecia en la gráfica, la pérdida comienza a descender de manera sostenida a partir de valores en torno a 4×10^{-6} , manteniendo una tendencia estable hasta aproximadamente 2×10^{-3} , momento en el que empieza a aumentar de forma abrupta. Dentro de ese rango descendente se seleccionó como valor óptimo el 10^{-4} , señalado en la gráfica con una estrella verde, ya que representa un punto equilibrado que evita tanto el aprendizaje lento de tasas muy bajas como la inestabilidad asociada a valores elevados. Esta elección se ha utilizado como learning rate fijo para el optimizador Adam en las siguientes fases de entrenamiento. (Figura 29).

En lo que respecta al parámetro α de la función de pérdida BinaryFocalCrossentropy, se determinó asignarle un valor constante fundamentado en una fórmula ampliamente empleada en contextos de clasificación desbalanceada. Esta fórmula toma en consideración la proporción entre el número de instancias pertenecientes a la clase mayoritaria y la clase minoritaria, y se expresa como:

$$\alpha = \frac{1}{1 + \frac{N_{\text{mayoritaria}}}{N_{\text{minoritaria}}}}$$

¹⁸ Fuente: <https://blog.dataiku.com/hubfs/training loss.png>

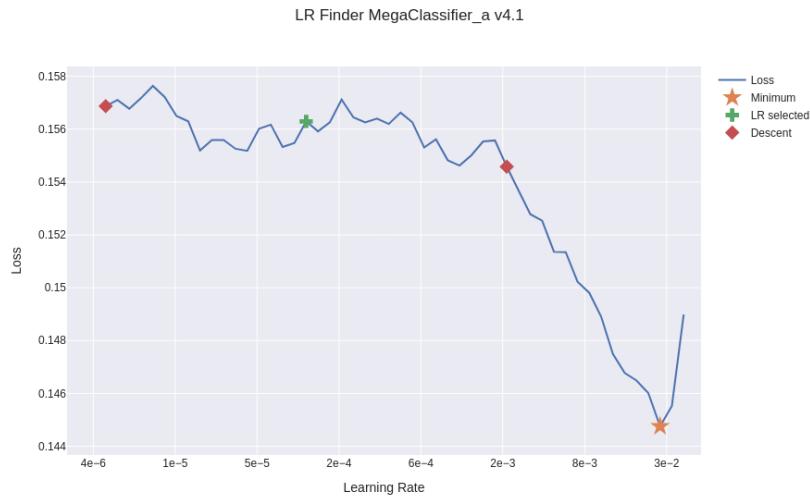


Figura 29: Gráfica método LRFinder en MegaClassifier_a

Esta expresión permite ajustar el peso que se le da a la clase minoritaria de forma automática en función del grado de desbalanceo del dataset. En este caso concreto, al aplicar la fórmula con las proporciones reales del conjunto de datos, se obtuvo un valor de $\alpha = 0.2611$. Este valor fue fijado durante el entrenamiento para reforzar la contribución de la clase menos representada en el cálculo de la pérdida, ayudando así a mejorar la sensibilidad del modelo.

En cuanto al parámetro γ , que controla la intensidad con la que la función Focal Loss penaliza las predicciones incorrectas con alta confianza, se evaluaron tres valores distintos: 1.0, 2.0 y 3.0. Este hiperparámetro permite enfocar el entrenamiento del modelo en los ejemplos más difíciles, reduciendo la influencia de las predicciones que el modelo ya clasifica correctamente con alta probabilidad.

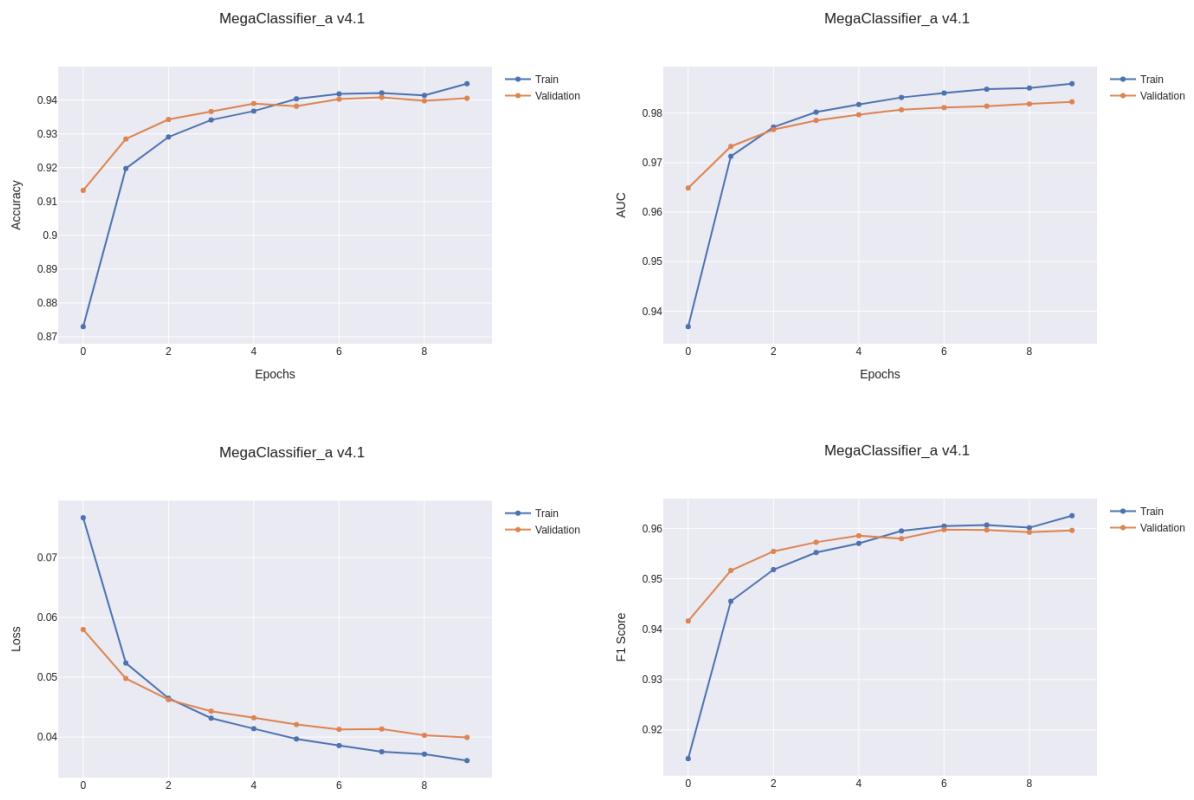


Figura 30: Gráficas de entrenamiento de accuracy, auc, los y f1-score de MegaClassifier_a_v4.1

La versión seleccionada como mejor ha sido la v4.1 con un valor de gamma de 2.0, ya que contribuye a una convergencia más estable, tal y como evidencian las curvas de entrenamiento y validación para métricas en especial accuracy y f1-score, que ahora se muestran más próximas. Sin embargo, se observa que en la loss aún persiste cierto margen de mejora, ya que la diferencia entre la pérdida de entrenamiento y la de validación sigue estando presente, aunque se ha reducido. Este hecho indica que el modelo puede seguir beneficiándose de técnicas que refuerzan su capacidad de generalización.

En la versión 5 del modelo se implementaron técnicas de augmentation de datos con el propósito de incrementar la variabilidad del conjunto de entrenamiento y minimizar el riesgo de sobreajuste en etapas posteriores de optimización fina. Esta estrategia busca simular distintas condiciones del dato real, favoreciendo así una mayor capacidad de generalización del modelo sin necesidad de añadir nuevos datos manualmente. De esta manera, se establecen las bases para realizar un ajuste fino más efectivo y seguro, manteniendo la robustez alcanzada en versiones anteriores.

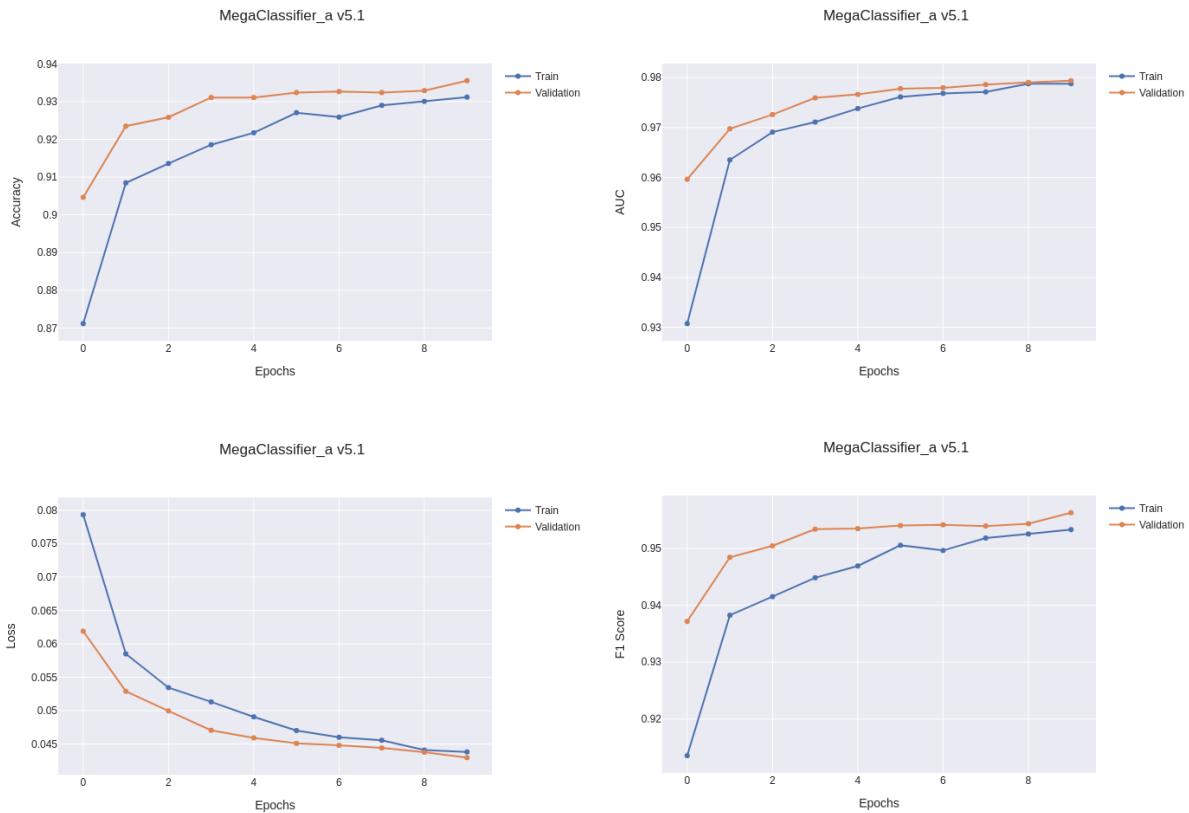


Figura 31: Gráficas de entrenamiento de accuracy, auc, los y f1-score de *MegaClassifier_a_v5.1*

En la versión 5.1 del modelo se han implementado cuatro técnicas de data augmentation que han demostrado su eficacia: el volteo horizontal, el ajuste de brillo entre 80% y 100%, rotaciones aleatorias de 15° y zoom de hasta un 20%. Los resultados obtenidos no solo mantienen los buenos resultados de versiones anteriores, sino que también se observa una mayor estabilidad en las curvas de validación, lo cual resulta especialmente útil como base sólida para el proceso de fine-tuning.

En la versión 6 del modelo se introdujo la técnica de Early Stopping como mecanismo de regularización adicional para evitar el sobreentrenamiento y mejorar la eficiencia del proceso de entrenamiento. El propósito de esta implementación fue inducir la finalización automática del

entrenamiento cuando el modelo evidenciara una disminución en la mejora de ciertas métricas críticas, evitando de este modo una pérdida en la generalización. Se evaluaron diversos criterios de monitoreo, tales como la pérdida de validación, con el propósito de minimizarla, así como métricas orientadas a la calidad del modelo, como val_auc, val_precision y val_recall, en estos casos buscando su maximización. Esta estrategia permite ajustar dinámicamente el número óptimo de épocas, evitando entrenamientos innecesarios y asegurando que el modelo conserve el mejor rendimiento alcanzado durante el proceso.

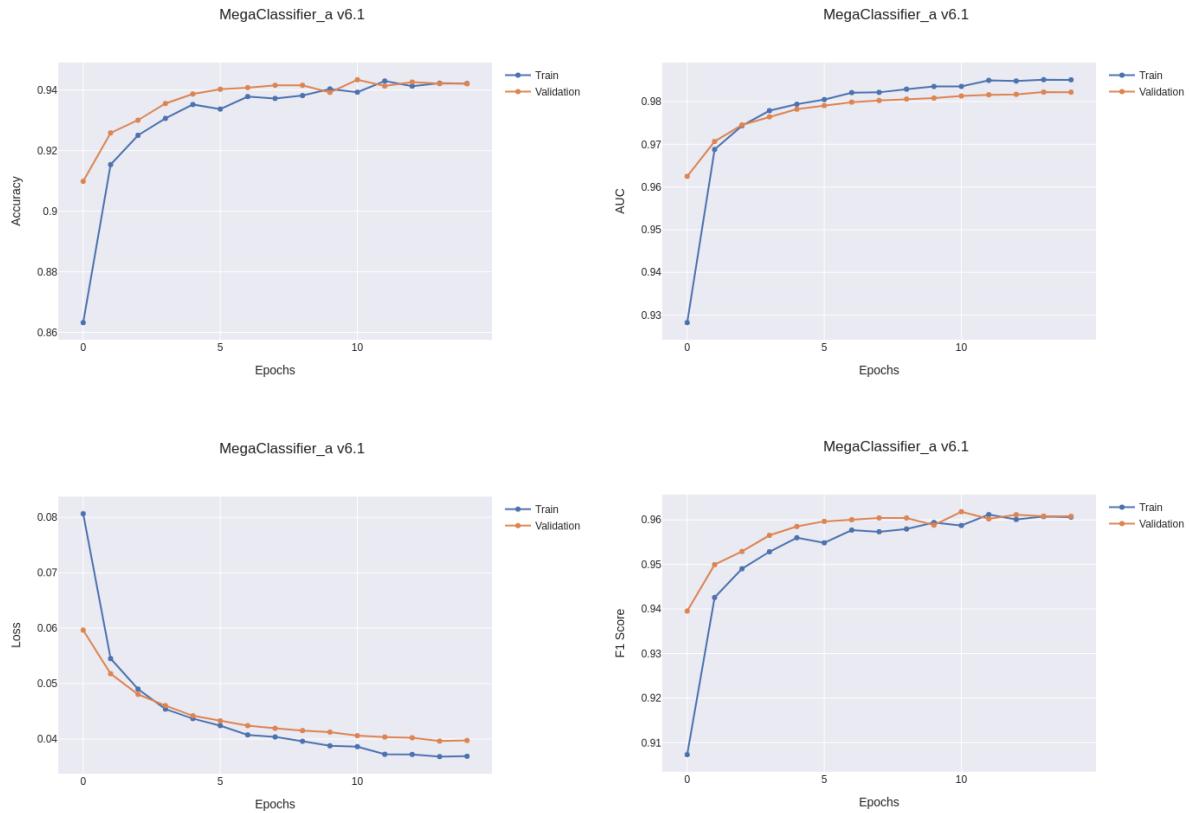


Figura 32: Gráficas de entrenamiento de accuracy, auc, los y f1-score de *MegaClassifier_a_v6.1*.

En esta sexta versión, la adopción de Early Stopping como criterio de regularización permitió interrumpir automáticamente el entrenamiento en el punto en el que la métrica de validación val_auc (v6.1) dejó de mejorar de forma significativa. Al centrarse en maximizar la habilidad discriminativa del modelo con la métrica AUC, se consigue un entrenamiento más eficiente y menos propenso al sobreajuste, pues se evita alargar el proceso una vez alcanzado el mejor rendimiento en validación. En las gráficas se aprecia cómo el modelo conserva una convergencia estable entre entrenamiento y validación, mostrando mejoras graduales en F1-score y AUC sin que la loss de validación se eleve de nuevo, lo que confirma que el entrenamiento se detiene en un momento óptimo para la generalización.

En la versión 7 del modelo se implementó el proceso de fine-tuning, con el propósito de optimizar el aprendizaje del modelo base previamente entrenado mediante transfer learning. Para ello, se desbloquearon progresivamente distintos bloques de capas del modelo preentrenado, evaluando el impacto en el rendimiento al permitir que se ajusten sus pesos. Se implementaron cuatro configuraciones: 20 capas, 40 capas, 80 capas y el modelo completo. De manera simultánea, se implementó una disminución en la tasa de aprendizaje en comparación con la fase de entrenamiento inicial, con el propósito de impedir una alteración abrupta de los

pesos que ya habían sido optimizados. Esta estrategia permitió analizar el equilibrio entre flexibilidad de ajuste y riesgo de sobreajuste, clave en etapas avanzadas de entrenamiento fino.

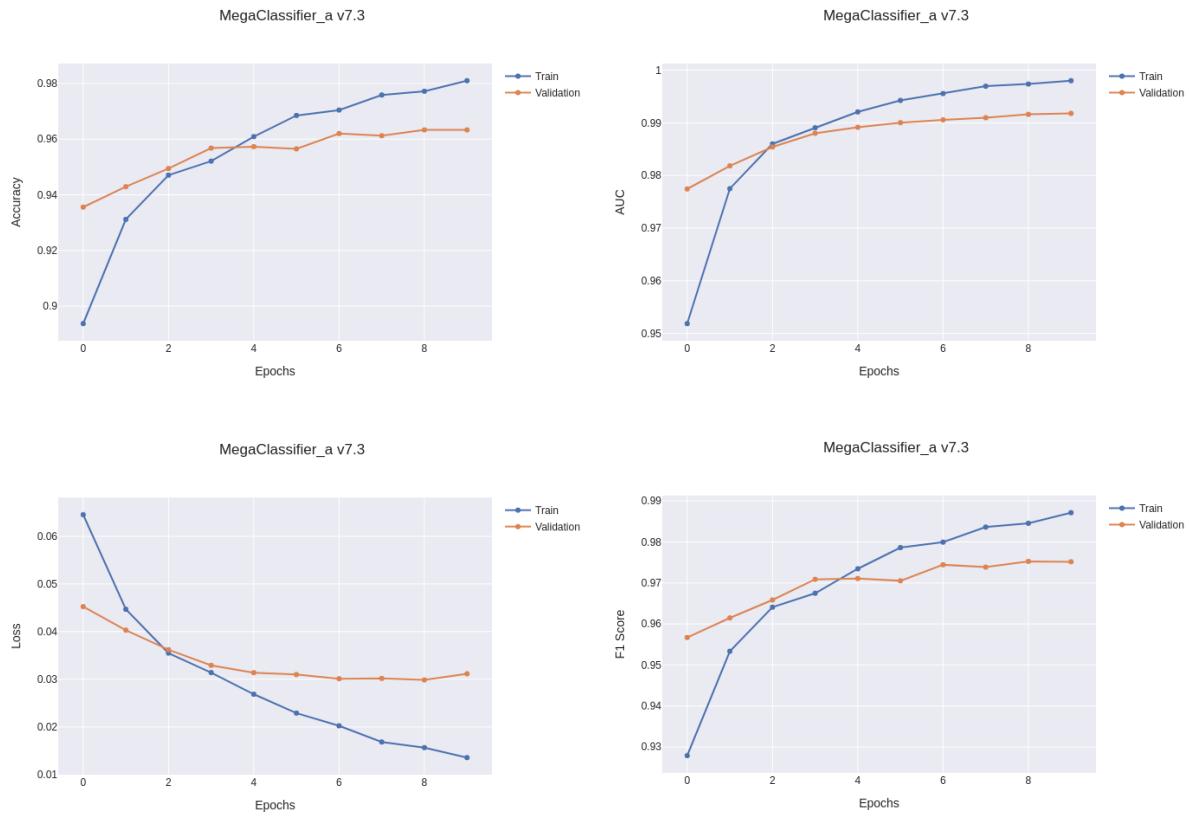


Figura 33: Gráficas de entrenamiento de accuracy, auc, los y f1-score de *MegaClassifier_c_v7.3*

La versión 7.3 evidencia el máximo rendimiento cuantitativo del modelo hasta el momento, tras la implementación de un ajuste fino exhaustivo. No obstante, el análisis visual de las gráficas sugiere que el modelo comienza a sobreajustarse, la pérdida se estabiliza o incluso sube levemente, mientras las curvas de rendimiento en entrenamiento siguen creciendo. En consecuencia, se ha optado por avanzar con la versión 8, la cual incorpora la estrategia Early Stopping con el propósito de prevenir que un fine-tuning excesivo comprometa la capacidad de generalización del modelo además de volver a reducir en 10^{-1} el learning rate.

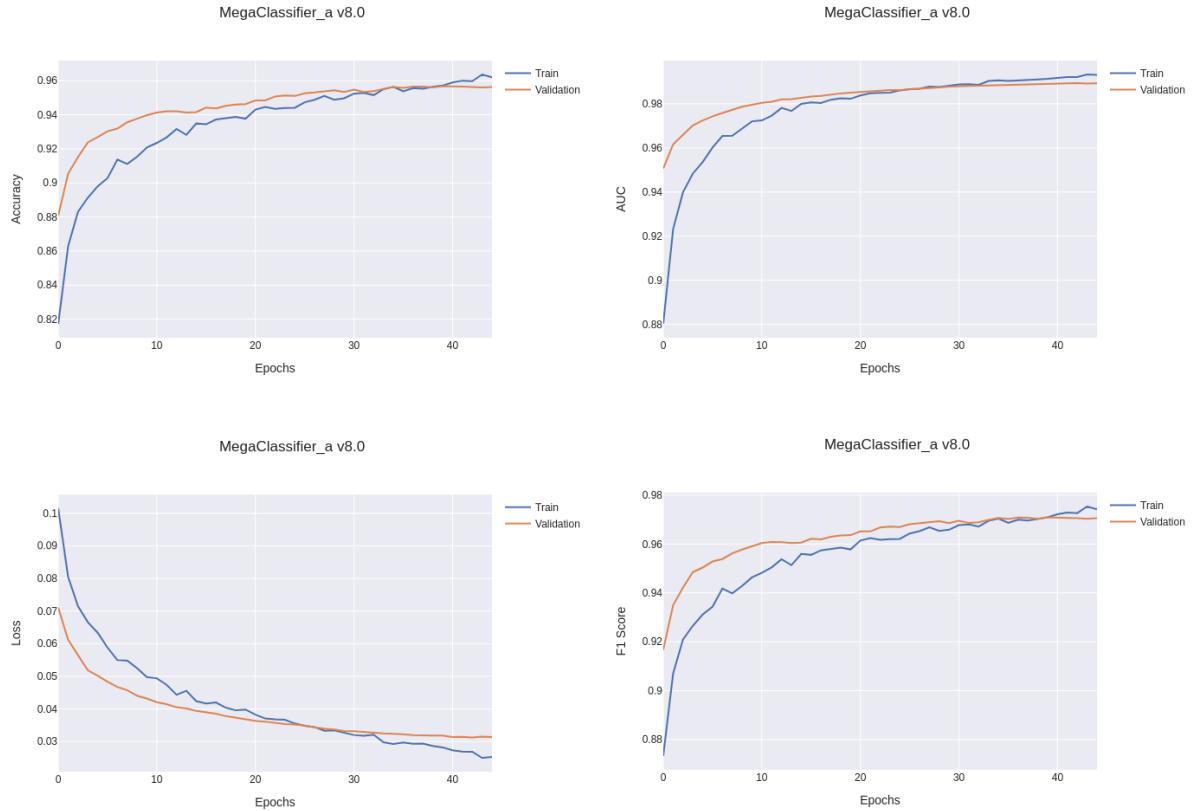


Figura 34: Gráficas de entrenamiento de accuracy, auc, los y f1-score de *MegaClassifier_a_v8.0*

En esta versión final v8.0, se han consolidado todas las mejoras introducidas en etapas previas incluyendo la selección del learning rate óptimo, el ajuste de los hiperparámetros de la función de perdida, las estrategias de data augmentation y el criterio de Early Stopping, para lograr un modelo con un rendimiento más robusto en la tarea de clasificación binaria. Las gráficas muestran una convergencia estable entre el entrenamiento y la validación en métricas clave (accuracy, AUC, F1-score), con una loss que desciende de forma consistente y se mantiene baja en validación. Ello pone de manifiesto un mayor equilibrio entre la capacidad de aprendizaje y la prevención del sobreajuste, evidenciando un salto cualitativo respecto a versiones anteriores y reflejando que el modelo aprovecha de manera eficaz el enriquecimiento de datos y las técnicas de regularización implementadas.

4.3.4 Entrenamiento de MegaClassifier_b

En este apartado, vamos a abordar los resultados del entrenamiento del modelo denominado MegaClassifier_b. Es el segundo y último modelo que hace uso de las imágenes preprocesadas por MegaDetector. En este caso, el conjunto de datos no se reduce como en el modelo anterior, sino que las imágenes en las que MegaDetector genera detecciones se preprocesan, como se describió en el apartado 4.3.2 Preprocesamiento de imágenes con MegaDetector, y las imágenes en las que MegaDetector no genera detecciones se aportan como entrada a la CNN.

Como el versionado sigue el mismo criterio para todos los modelos, en este apartado se detallará de una forma más escueta.

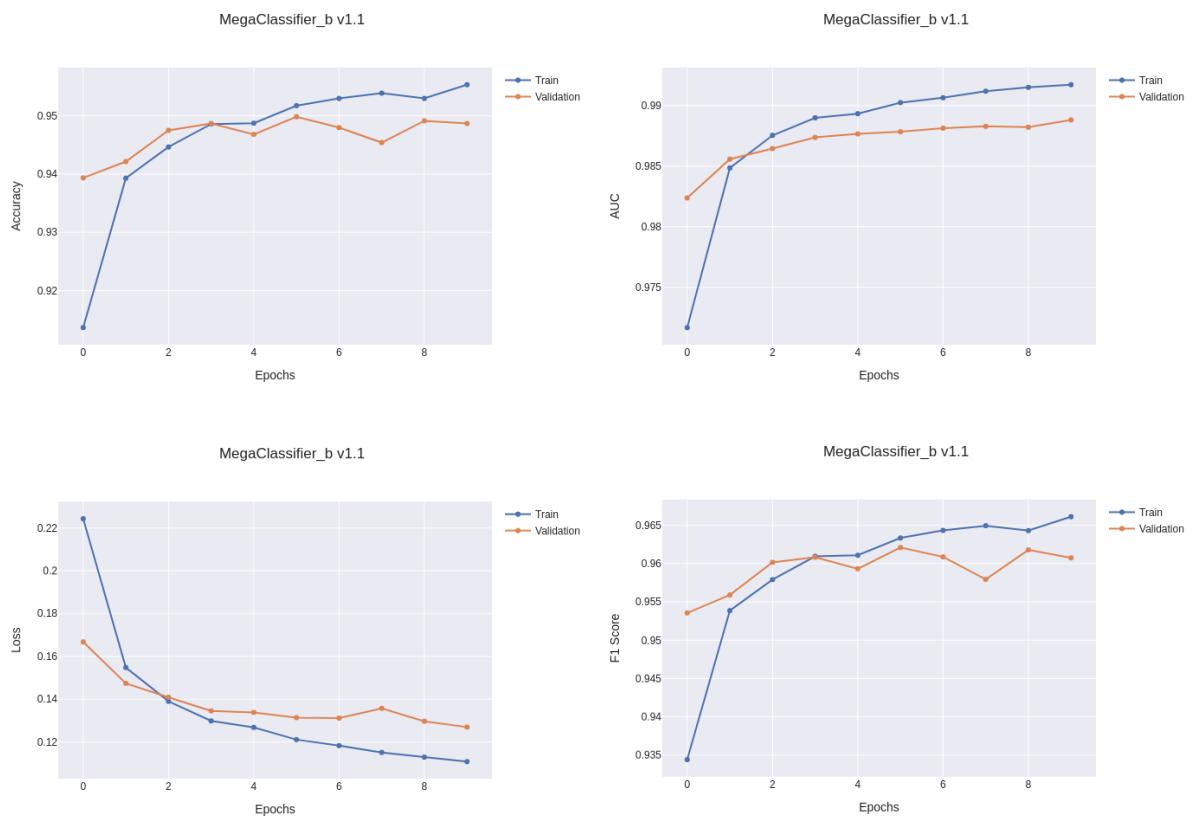


Figura 35: Gráficas de entrenamiento de accuracy, auc, los y f1-score de *MegaClassifier_b_v1.1*.

En esta primera versión del modelo megaclassifier_b, la configuración que logra un mejor equilibrio entre el entrenamiento y la validación es la v1.1, con un tamaño de lote de 32.

Al analizar las gráficas de precisión, pérdida, AUC y F1-score, se observa que los valores de validación progresan de manera más estable y se mantienen cercanos a los del entrenamiento, lo que indica una mejor capacidad de generalización que otras opciones de tamaño de lote. Concretamente, se aprecia que la pérdida de validación desciende de manera más sostenida, reduciendo la brecha frente a la pérdida de entrenamiento, y que tanto el AUC como el F1-score en validación se elevan sin grandes oscilaciones.

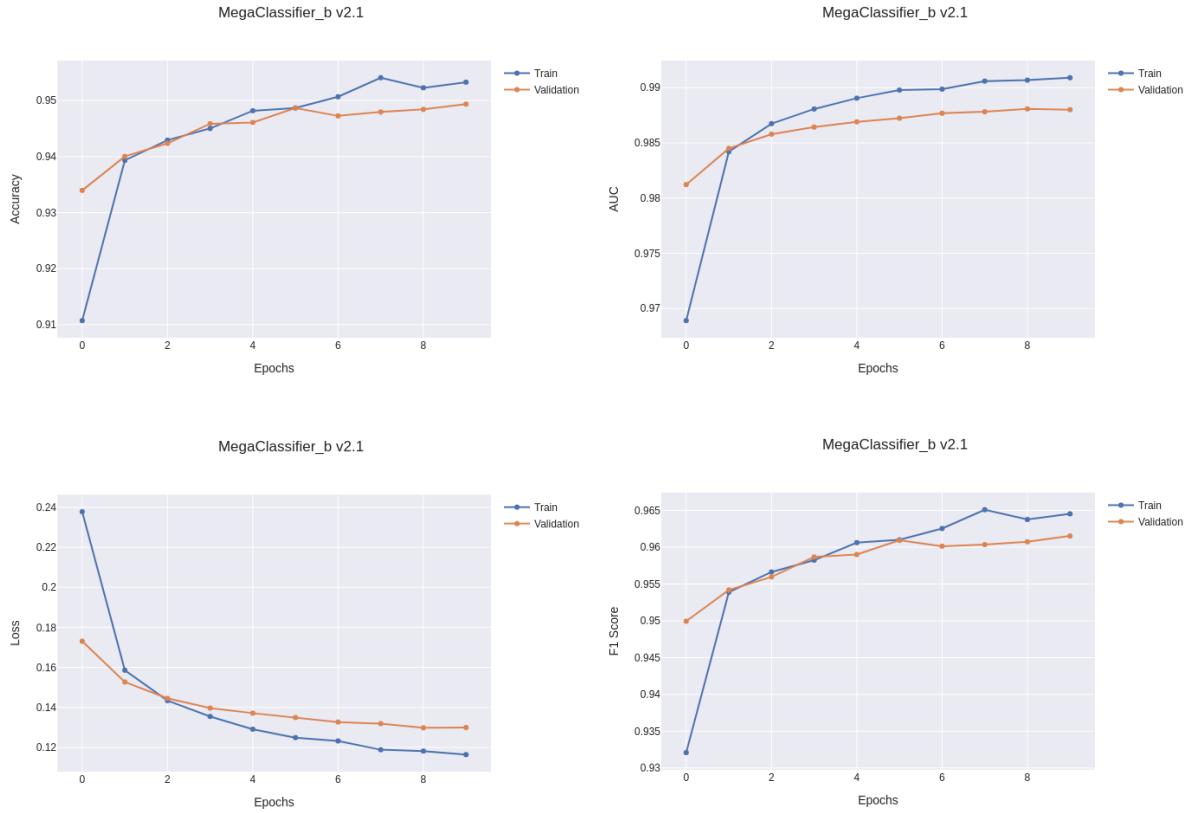


Figura 36: Gráficas de entrenamiento de accuracy, auc, los y f1-score de *MegaClassifier_b_v2.1*

En esta segunda versión, el uso de RMSprop (v2.1) ofreció mejores resultados que los otros optimizadores evaluados, logrando una convergencia más estable entre las curvas de entrenamiento y validación. Se observa que la loss de validación desciende de forma más sostenida, manteniéndose cerca de la loss de entrenamiento sin mostrar oscilaciones drásticas, mientras que tanto el AUC como el F1-score presentan valores altos y relativamente constantes. Este comportamiento sugiere que la naturaleza adaptativa de RMSprop ayuda a manejar mejor los gradientes, equilibrando la velocidad de aprendizaje.

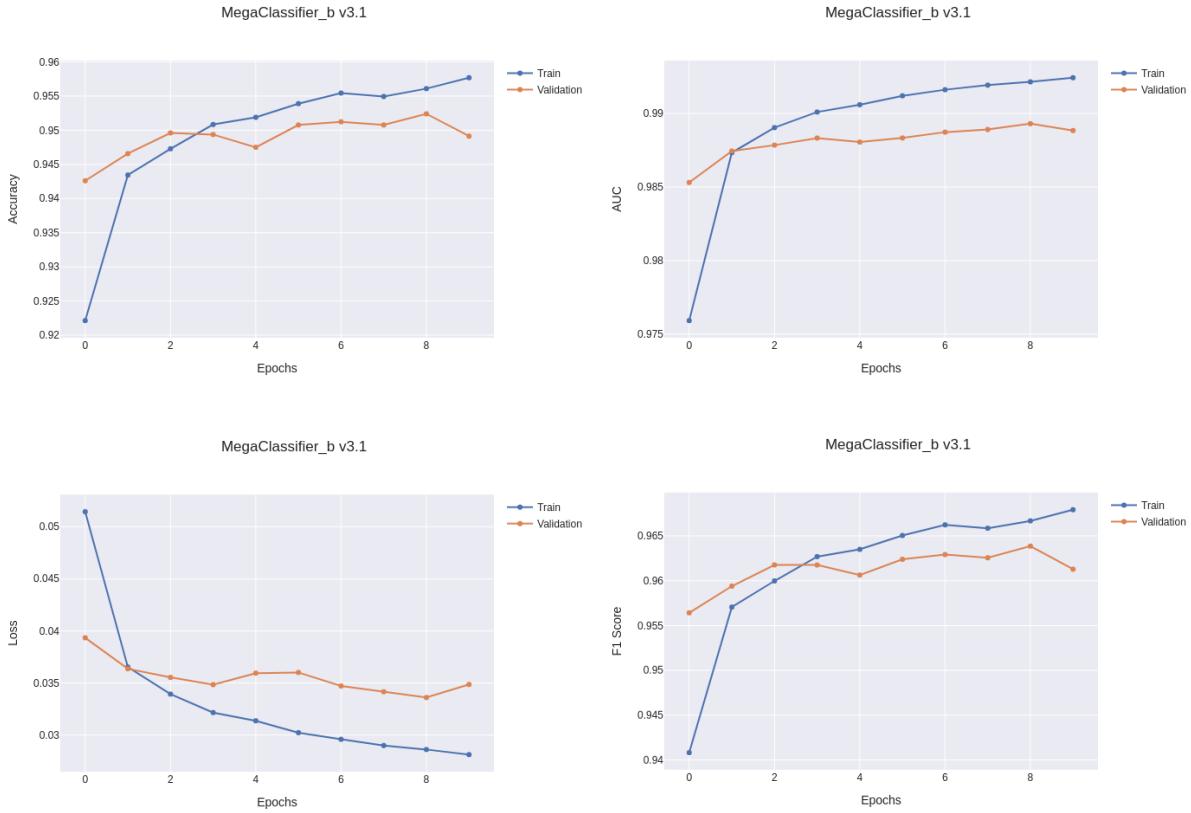


Figura 37: Gráficas de entrenamiento de accuracy, auc, los y f1-score de *MegaClassifier_b_v3.1*

En esta tercera versión, el uso de BinaryFocalCrossentropy (v3.1) permitió gestionar de manera más efectiva el desequilibrio de clases, al penalizar con mayor intensidad los errores en ejemplos menos representados. Las curvas de precisión y pérdida en el entrenamiento y la validación se mantienen más próximas entre sí, y tanto el área bajo la curva como el F1-Score muestran incrementos notables, lo que indica que el modelo logra capturar mejor los casos minoritarios sin perder precisión en la clase mayoritaria. Este comportamiento sugiere que la Focal Loss contribuye a un mayor equilibrio entre sensibilidad y precisión, lo que mejora el rendimiento general en comparación con la función de pérdida estándar.

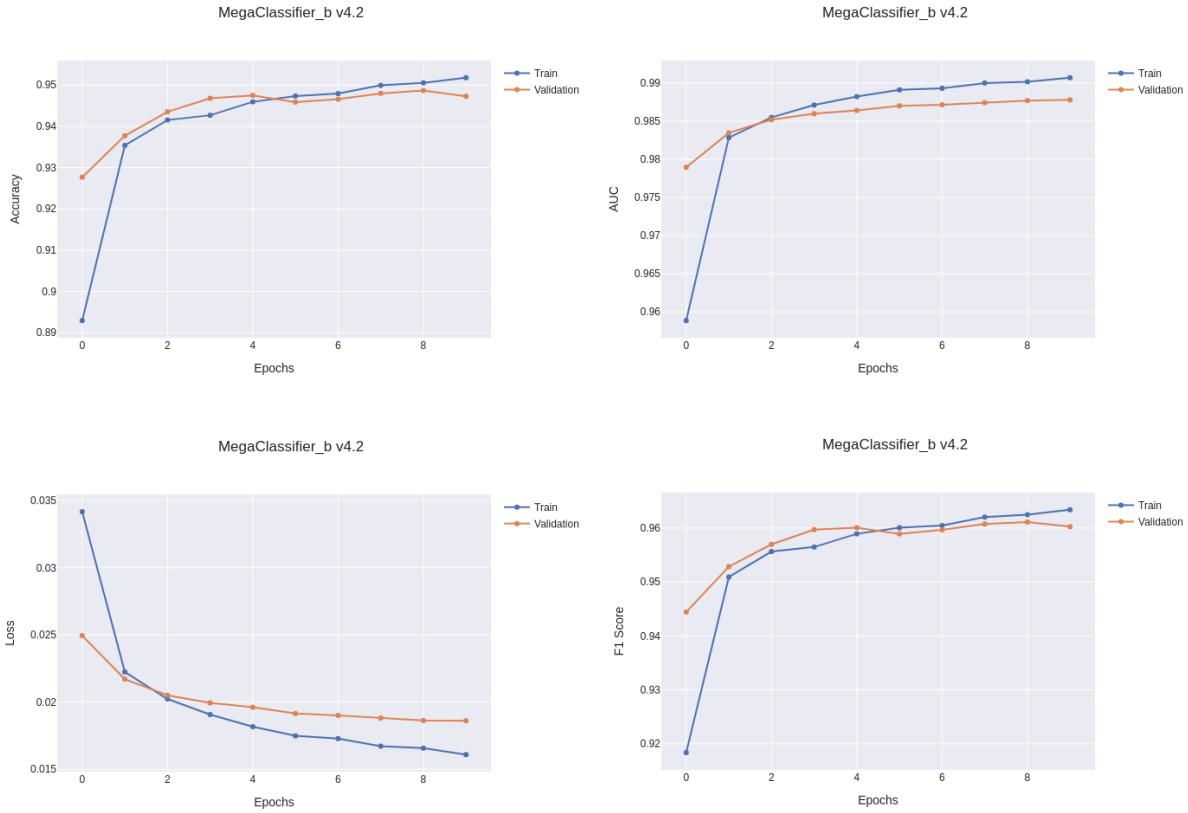


Figura 38: Gráficas de entrenamiento de accuracy, auc, los y f1-score de *MegaClassifier_b_v4.2*

En esta cuarta versión, con un learning rate de 2×10^{-4} , $\alpha = 0.3383$ y $\gamma = 3.0$, el modelo (v4.2) logra un ajuste más afinado en la Focal Loss, potenciando la capacidad de discriminar correctamente los ejemplos minoritarios y reduciendo, a la vez, el sobreajuste. Al examinar las gráficas de accuracy, loss, AUC y F1-score, se observa una disminución consistente de la loss y un incremento sostenido de las métricas de validación, lo que indica una mejor generalización comparada con configuraciones anteriores. Especialmente relevante resulta el F1-score, que refleja un adecuado equilibrio entre precisión y exhaustividad, beneficiándose de la penalización más fuerte que impone $\gamma = 3.0$ en los errores difíciles.

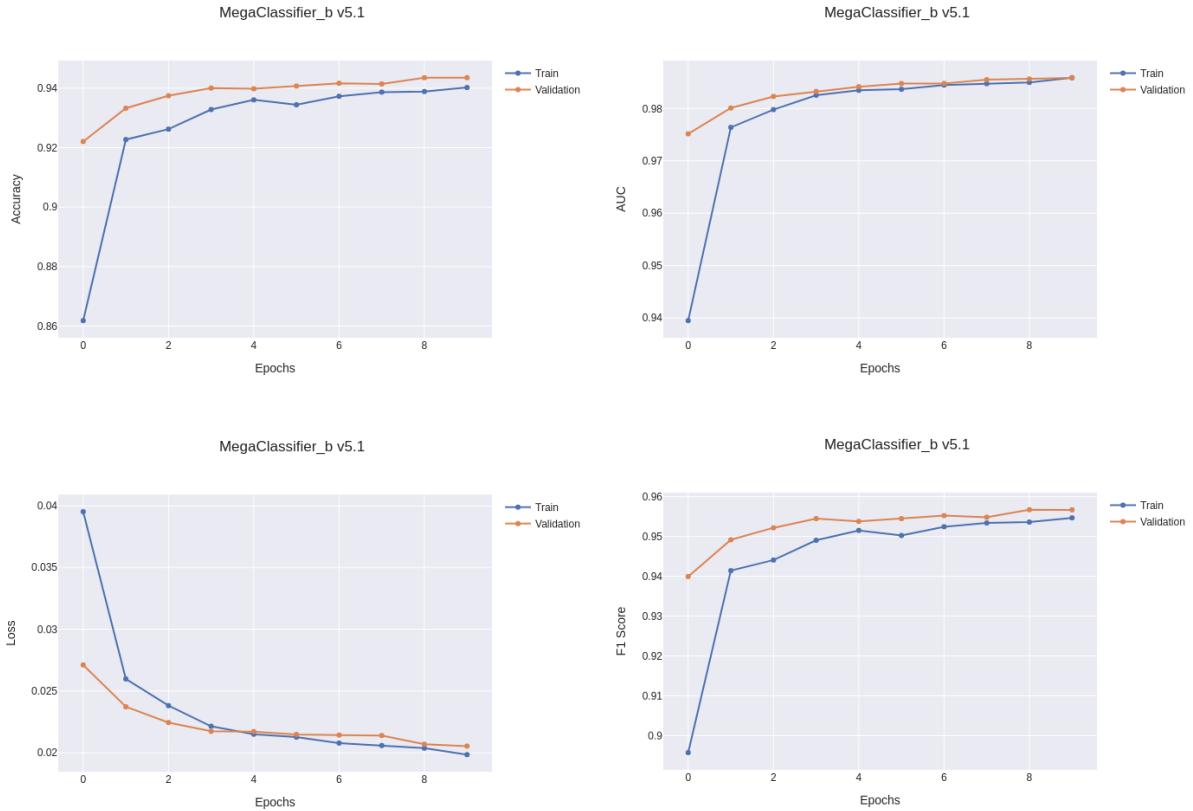


Figura 39: Gráficas de entrenamiento de accuracy, auc, los y f1-score de *MegaClassifier_b_v5.1*

En esta quinta versión, la introducción de data augmentation provee una variedad adicional de muestras al conjunto de entrenamiento, lo que favorece la generalización del modelo y reduce la brecha entre los resultados en entrenamiento y validación. Según se observa en las gráficas, la loss de validación desciende de forma más estable y las métricas de evaluación (accuracy, AUC, F1-score) muestran una tendencia al alza, evidenciando que el modelo se ha vuelto más robusto ante la variabilidad de los datos y menos propenso al sobreajuste.

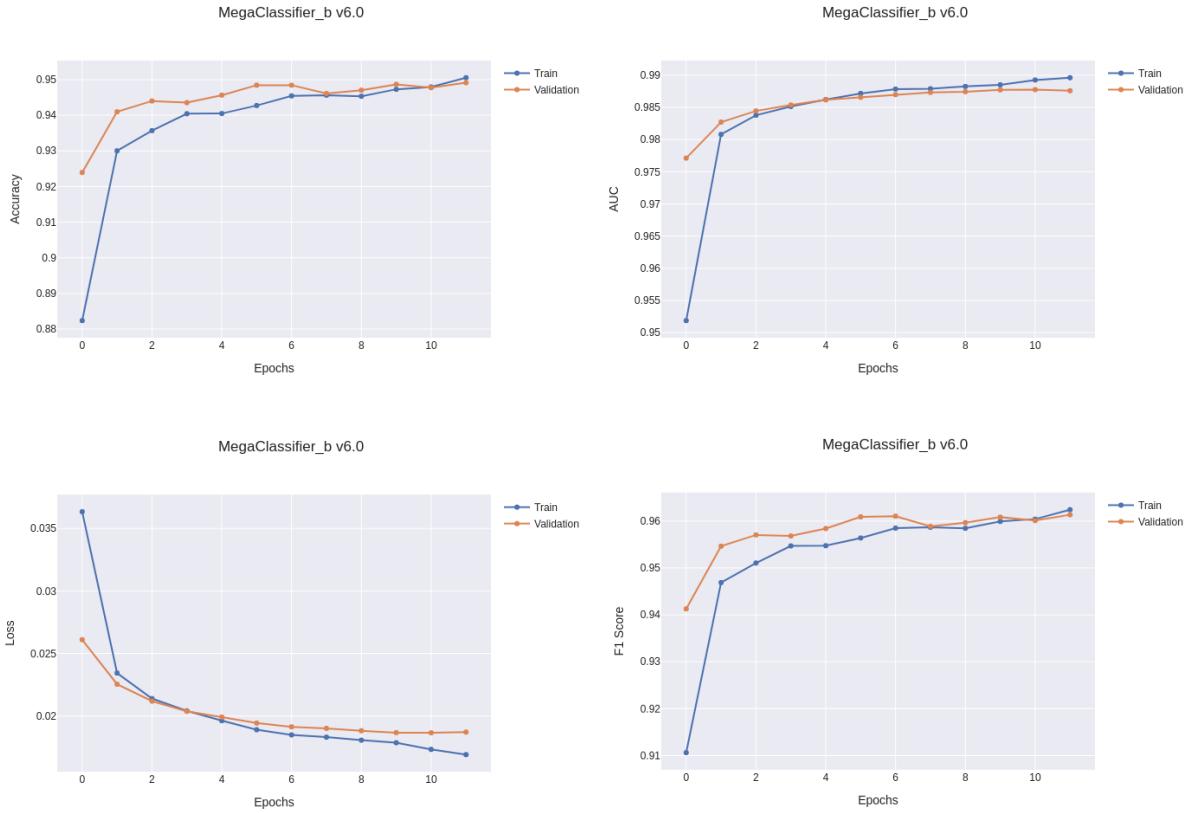


Figura 40: Gráficas de entrenamiento de accuracy, auc, los y f1-score de *MegaClassifier_b_v6.0*

En esta sexta versión del modelo *MegaClassifier_b*, se introdujo el uso de la función Early Stopping, que utiliza como criterio de parada la minimización de *val_loss*. Esta estrategia demostró ser efectiva para evitar el sobreentrenamiento y garantizar que el modelo conserve su mejor capacidad de generalización. Las gráficas muestran una clara estabilización entre las curvas de entrenamiento y validación: tanto la loss como las métricas de rendimiento (accuracy, AUC y F1-score) progresan de forma paralela, sin divergencias significativas. En particular, en *val_loss*, se observa una clara meseta que justifica el punto de parada automática en la época 11, restaurando los pesos de la época anterior. Este comportamiento indica que se ha alcanzado un rendimiento óptimo sin incurrir en sobreajuste y optimizando la eficiencia del proceso de entrenamiento.

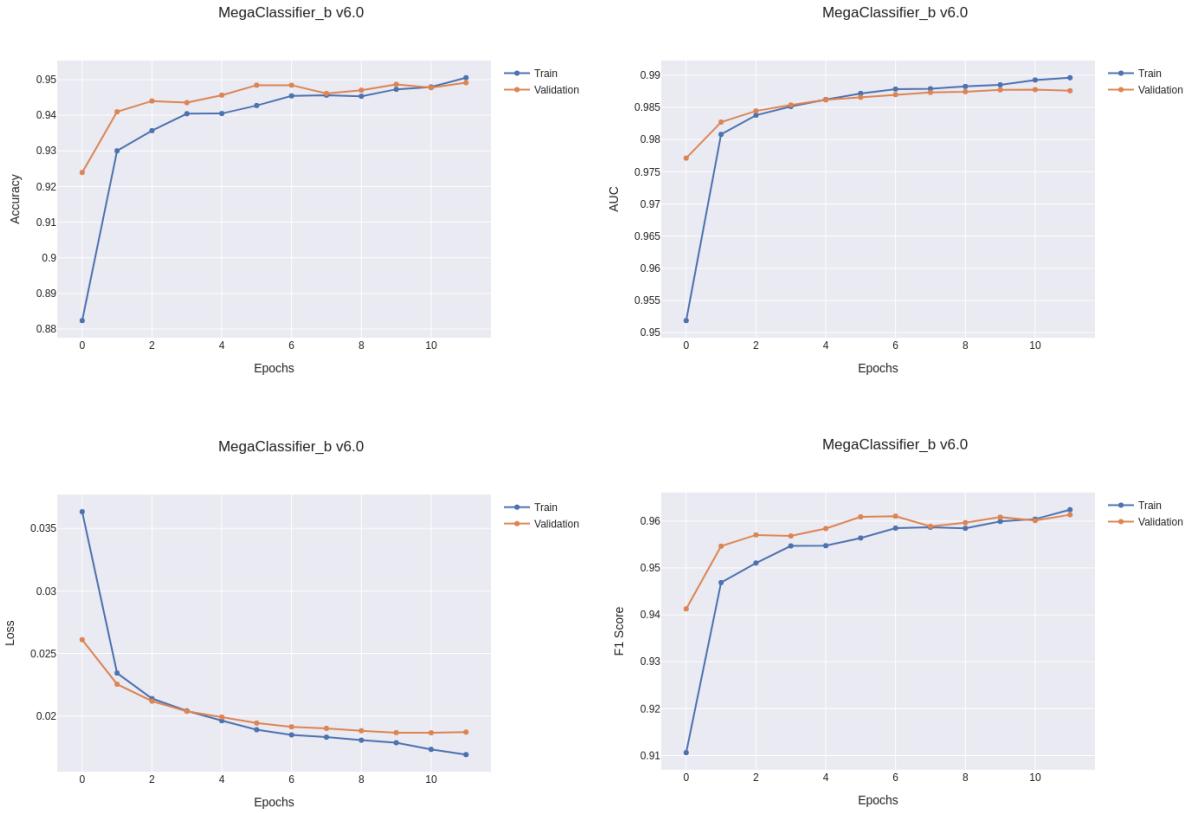


Figura 41: Gráficas de entrenamiento de accuracy, auc, los y f1-score de MegaClassifier_b_v7.3

En esta versión 7.3 del modelo MegaClassifier_b, se llevó a cabo el proceso de fine-tuning desbloqueando completamente las capas del modelo base, permitiendo así un ajuste más profundo de sus pesos en función del conjunto de datos específico. Esta estrategia se traduce en una mejora generalizada de todas las métricas: las curvas de accuracy, AUC, F1-score y loss reflejan una progresión continua y estable, con validación muy próxima al entrenamiento, lo que indica una excelente capacidad de generalización sin indicios claros de sobreajuste. Destaca especialmente el rendimiento en AUC y F1-score, que alcanzan valores notablemente altos, evidenciando una mejora en la discriminación de clases. No obstante, a pesar del buen desempeño de esta versión, no se desarrolla una versión 8 de la rama b, ya que en el capítulo siguiente se demuestra que los modelos basados en MegaClassifier_a, ofrecen un rendimiento superior, consolidando así la dirección más prometedora del pipeline.

4.3.5 Entrenamiento de MegaClassifier_c

Como se ha mencionado anteriormente en apartados previos, la entrada proporcionada a esta CNN son las imágenes originales del dataset, a las cuales se ha aplicado únicamente el preprocesamiento necesario para satisfacer los requerimientos de la arquitectura utilizada. En el presente apartado se procederá a la exposición de los datos y métricas de entrenamiento correspondientes a las diversas versiones del modelo denominado MegaClassifier_c, el cual es el único de los implementados que no hace uso de MegaDetector.

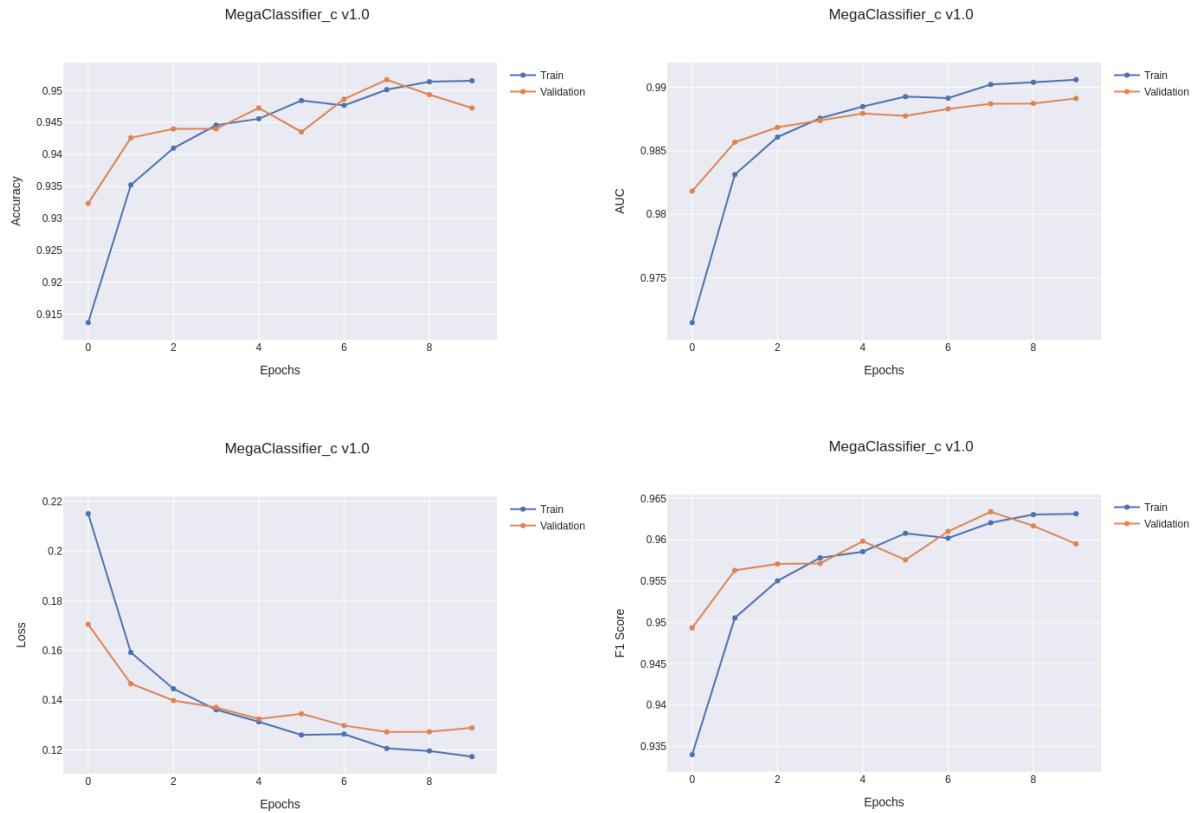


Figura 42: Gráficas de entrenamiento de accuracy, auc, los y f1-score de *MegaClassifier_c_v1.0*

En esta primera versión del modelo *MegaClassifier_c*, se evaluaron distintos valores de batch size, siendo la configuración de 16 (v1.0) la que ofreció un mejor equilibrio entre aprendizaje y generalización. Las curvas de accuracy, AUC, F1-score y loss reflejan una evolución positiva y consistente, con una separación controlada entre entrenamiento y validación, lo que sugiere un entrenamiento efectivo y sin sobreajuste aparente. No obstante, también se aprecian ciertos aspectos mejorables: por un lado, la loss de validación se mantiene de forma sostenida por encima de la de entrenamiento, lo que podría indicar un margen de mejora en la regularización del modelo; por otro, tanto la accuracy como el F1-score en validación presentan ligeras oscilaciones hacia las últimas épocas, lo que sugiere que el modelo aún no ha alcanzado una convergencia completamente estable. Además, el hecho de que el AUC de validación alcance valores tan altos desde fases tempranas podría reflejar un comportamiento algo optimista, lo que invita a revisar la representatividad del conjunto de validación. En conjunto, la versión v1.0 representa un buen punto de partida, aunque con indicios que justifican seguir afinando el modelo en futuras iteraciones.

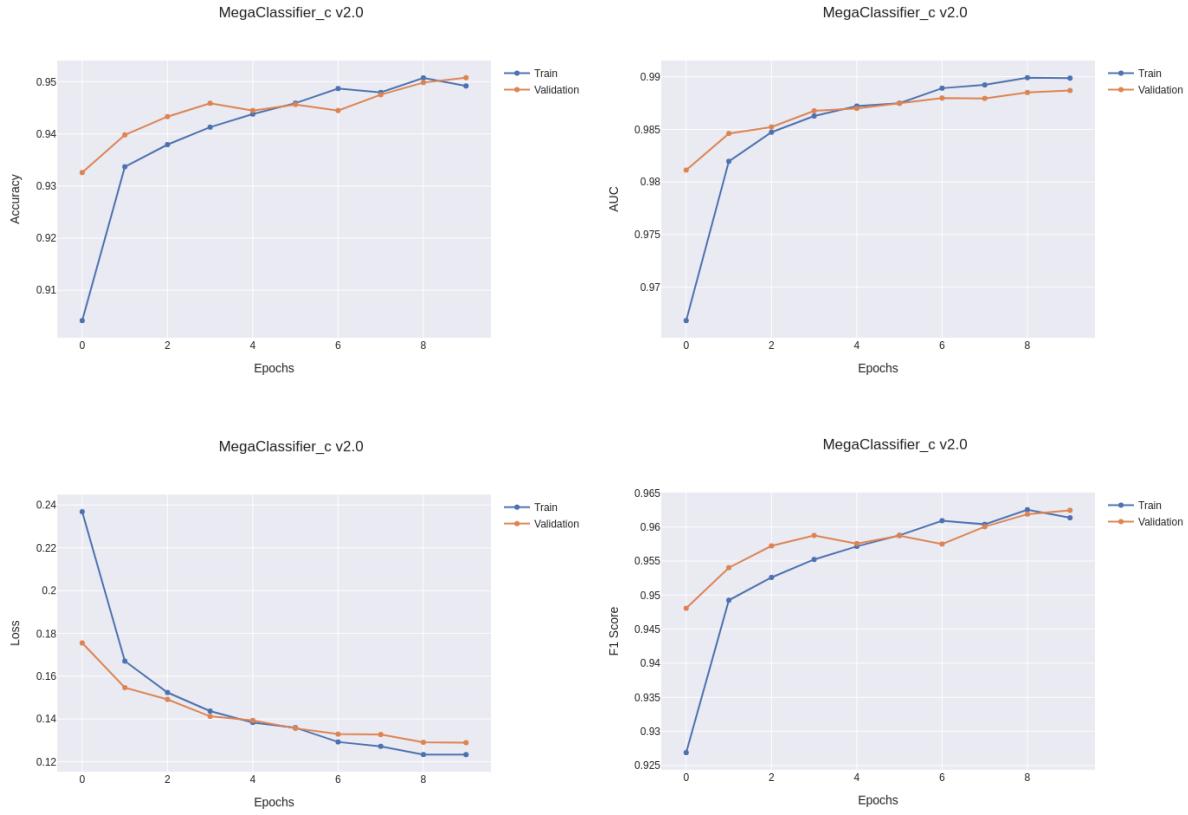


Figura 43: Gráficas de entrenamiento de accuracy, auc, los y f1-score de *MegaClassifier_c_v2.0*

En la versión 2 del modelo *MegaClassifier_c* se compararon distintos optimizadores, resultando nuevamente ganadora la configuración con Adam, al igual que en la versión anterior. Las gráficas muestran un comportamiento muy similar a v1.0, con métricas como accuracy, F1-score y AUC creciendo de forma estable y con curvas de validación cercanas a las de entrenamiento, lo que indica buena generalización. No obstante, persisten ligeras oscilaciones en las métricas de validación y una loss algo más alta en validación que en entrenamiento, lo que sugiere un leve margen de mejora.

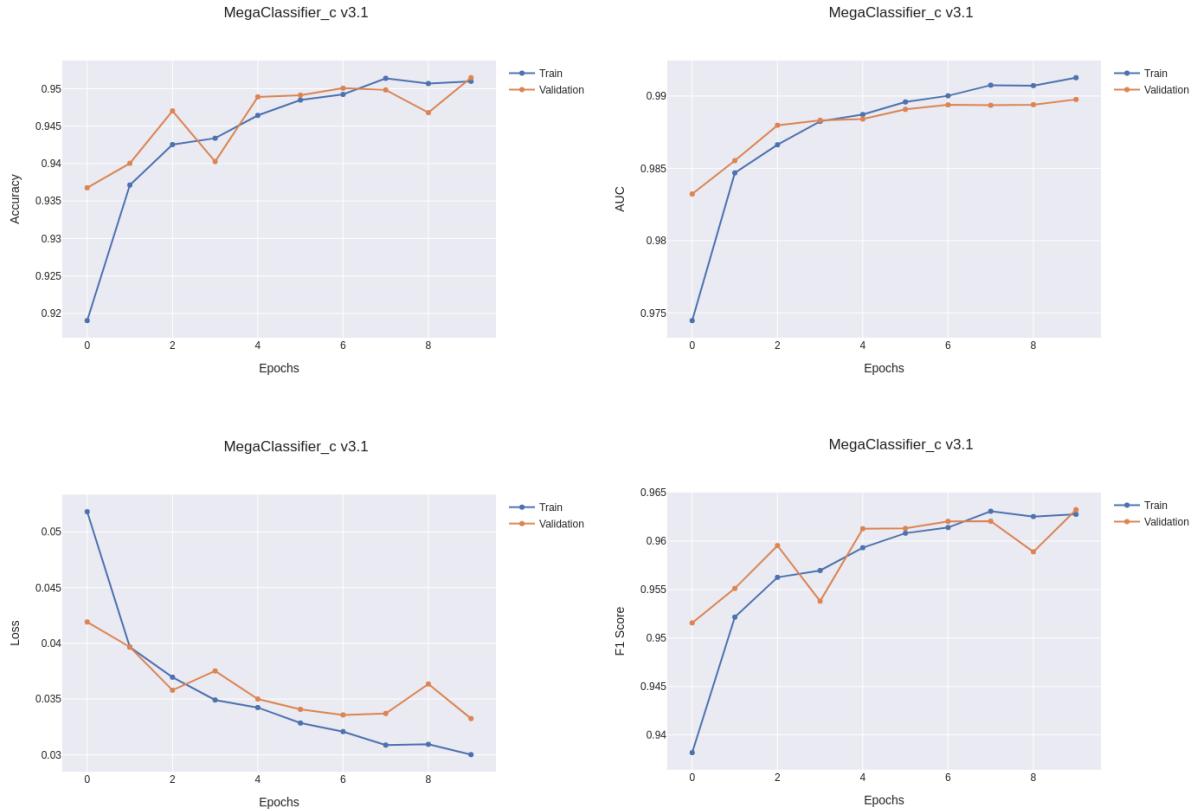


Figura 44: Gráficas de entrenamiento de accuracy, auc, los y f1-score de *MegaClassifier_c_v3.1*

En la versión 3 del *MegaClassifier_c* se evaluaron distintas funciones de pérdida para mejorar el rendimiento frente a posibles desequilibrios de clase. La versión ganadora fue la v3.1, que hace uso de *BinaryFocalCrossentropy*. Las métricas muestran un rendimiento sólido, tanto accuracy como f1-score y AUC presentan valores altos, con curvas de entrenamiento y validación muy cercanas entre sí, lo que sugiere una buena generalización del modelo. No obstante, en la métrica de pérdida se observan ciertas oscilaciones en la curva de validación, lo cual indica que podría haber margen de mejora en cuanto a la estabilidad del aprendizaje. Aun así, los resultados en conjunto consolidan esta versión como la más adecuada de esta fase.

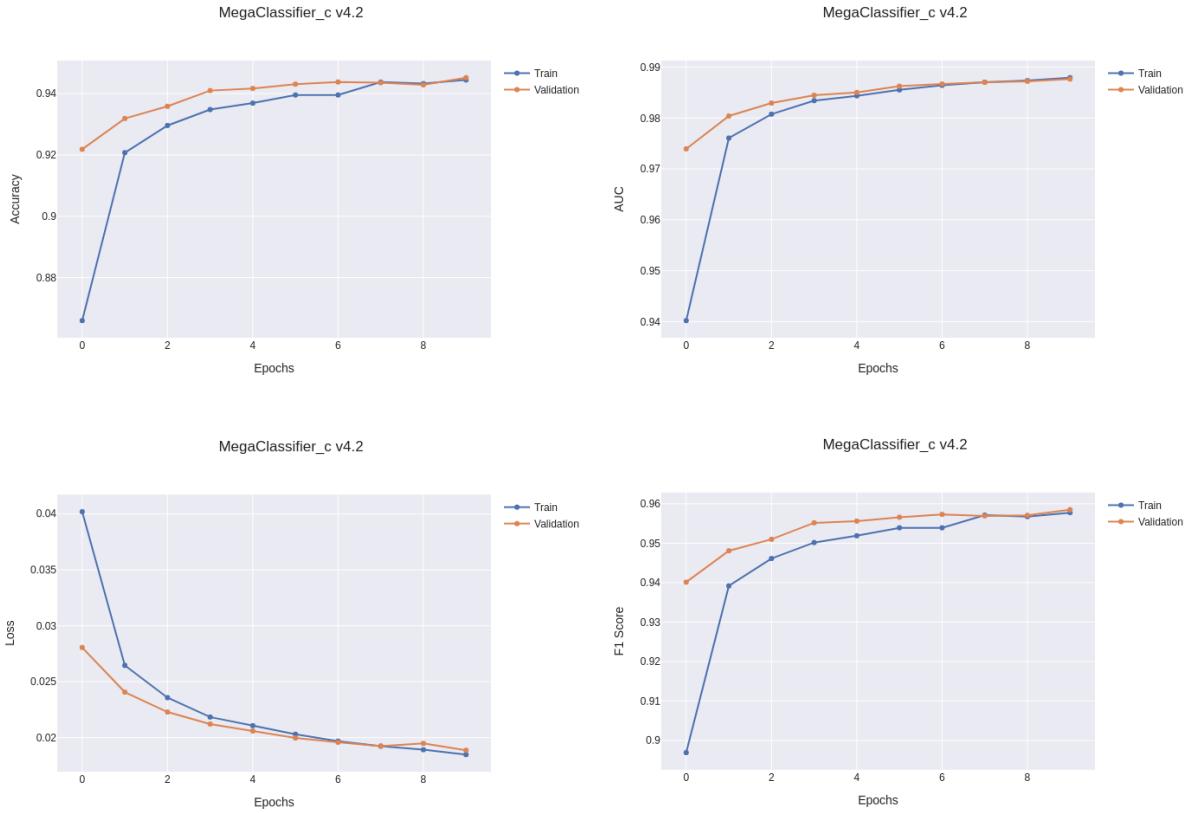


Figura 45: Gráficas de entrenamiento de accuracy, auc, los y f1-score de *MegaClassifier_c_v4.2*

En esta cuarta versión del modelo *MegaClassifier_c* se han ajustado específicamente los hiperparámetros del optimizador Adam y de la función de pérdida *BinaryFocalCrossentropy*, con un learning rate de 1e-4, un alpha de 0.3383 y un gamma de 3.0. Esta configuración, correspondiente a la versión v4.2, ha mostrado un comportamiento notablemente más estable en todas las métricas, con curvas de entrenamiento y validación que permanecen estrechamente alineadas a lo largo de las épocas. Especialmente en la métrica de pérdida, se observa una convergencia progresiva y sincronizada, sin indicios de sobreajuste. Aunque las diferencias de rendimiento respecto a versiones anteriores no son drásticas, sí se aprecia una mejora en la estabilidad y en el ajuste fino del modelo, lo que refuerza la elección de estos valores como óptimos dentro del contexto de esta arquitectura.

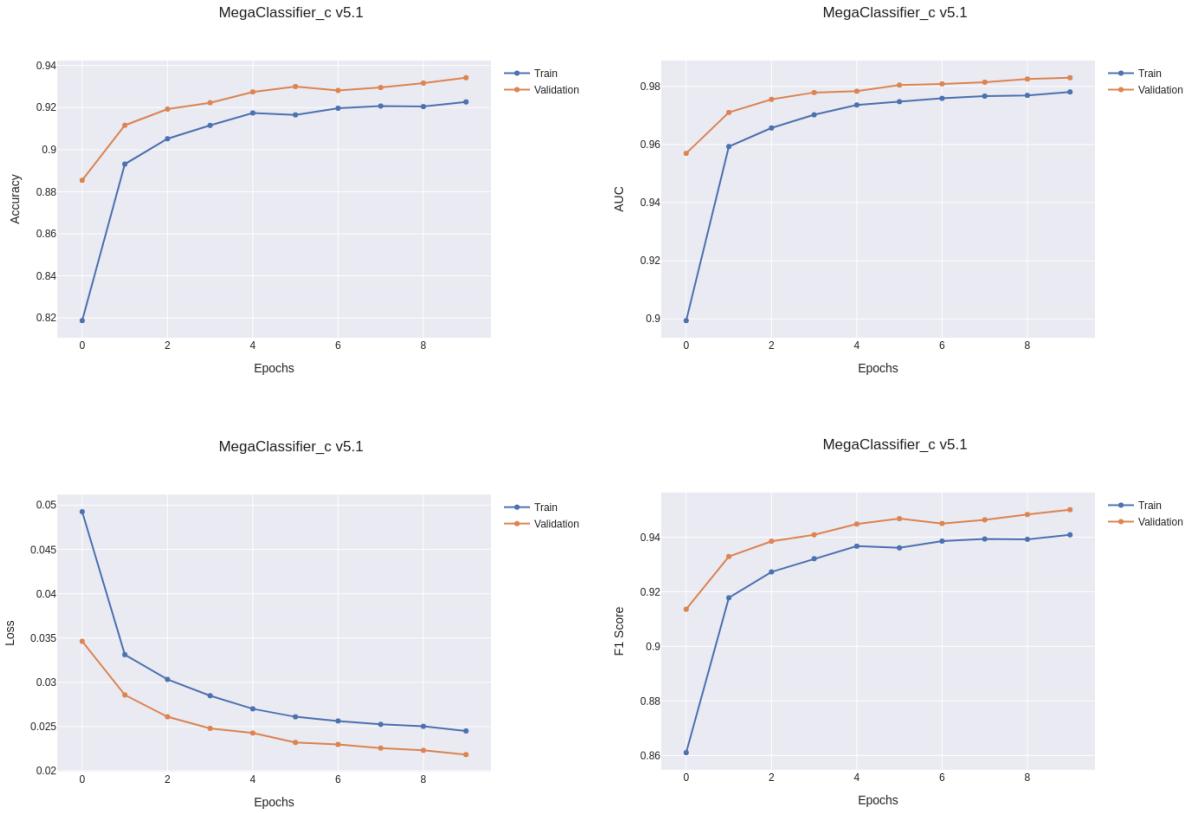


Figura 46: Gráficas de entrenamiento de accuracy, auc, los y f1-score de *MegaClassifier_c_v5.1*

En la versión 5 del modelo se introdujeron técnicas de data augmentation con el objetivo de mejorar la generalización. Como se observa en las métricas de la versión v5.1, esta estrategia ha contribuido a reducir el sobreajuste evidente en versiones anteriores: las curvas de entrenamiento y validación se mantienen más próximas a lo largo de las épocas, y la pérdida evoluciona de manera más paralela. Sin embargo, también se aprecia que el modelo aún no consigue cerrar del todo la brecha entre entrenamiento y validación, especialmente en loss, lo que sugiere que, si bien el efecto ha sido positivo, todavía queda margen de mejora para seguir fortaleciendo la capacidad de generalización del modelo.

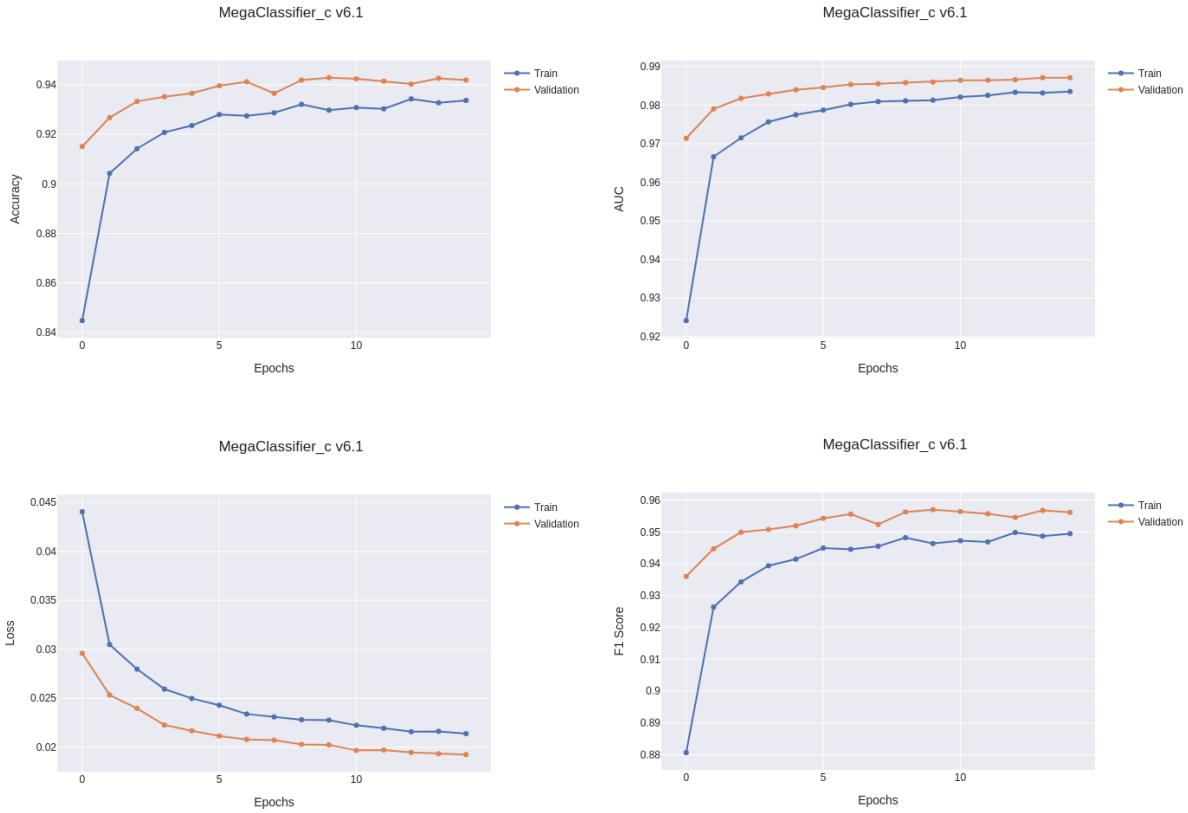


Figura 47: Gráficas de entrenamiento de accuracy, auc, los y f1-score de *MegaClassifier_c_v6.1*

En la versión 6.1 del modelo *MegaClassifier_c*, se introdujo la estrategia de early stopping utilizando como criterio la maximización del AUC de validación. Esta medida permitió detener el entrenamiento justo en el momento más favorable, evitando el sobreentrenamiento y asegurando que el modelo retuviera el mejor rendimiento alcanzado. A nivel de métricas, se observa una clara separación entre los conjuntos de entrenamiento y validación, especialmente visible en accuracy y f1-score, donde el modelo valida sistemáticamente con mejores valores que en train, un comportamiento inusual que podría estar indicando cierto regularization effect derivado del early stopping. No obstante, el descenso constante y estable de la loss en ambos conjuntos, junto con el excelente comportamiento del AUC (que alcanza valores superiores a 0.98 en validación), apuntan a una evolución sólida del modelo. A pesar de estos buenos resultados, el hecho de que train quede por debajo de val sugiere que podría explorarse una configuración más potente o un mayor número de épocas sin perder la generalización, especialmente ahora que el modelo parece bien regularizado. Aunque este aspecto queda justificado ya que esta versión sienta las bases para aplicar fine-tuning.

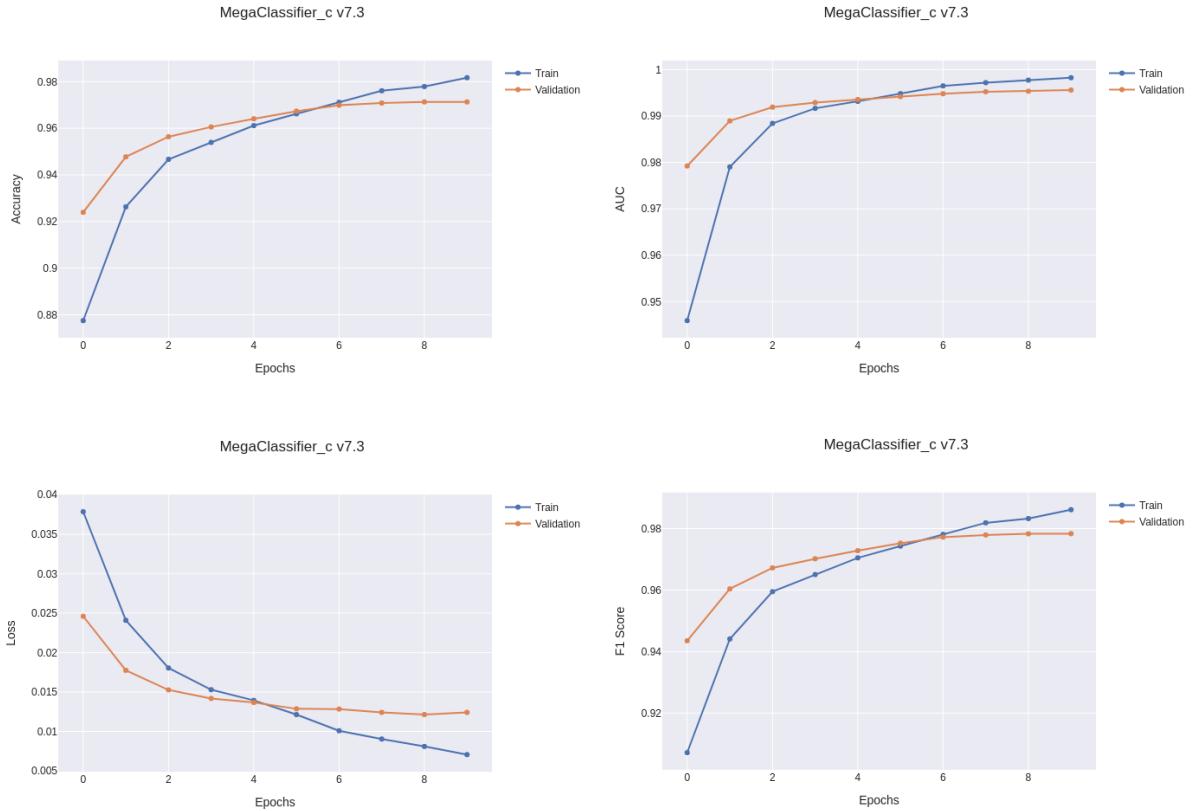


Figura 48: Gráficas de entrenamiento de accuracy, auc, los y f1-score de *MegaClassifier_c_v7.3*

A pesar del notable rendimiento de la versión 7.3, con mejoras consistentes y métricas muy elevadas en todas las dimensiones, se aprecian ciertos indicios que abren margen para optimización. Por un lado, la pérdida en validación tiende a estancarse hacia las últimas épocas, sin mostrar una mejora significativa, lo que sugiere que el modelo podría estar acercándose a una meseta sin beneficios adicionales por continuar entrenando. Además, aunque las curvas de entrenamiento y validación están bastante alineadas, la ligera separación en las últimas épocas junto a un gap constante en loss invita a explorar ajustes más finos. Por este motivo, en la versión 8 se optará por implementar early stopping para prevenir entrenamientos innecesarios y aplicar una reducción en la tasa de aprendizaje con el fin de favorecer una convergencia más suave y controlada en las fases finales del ajuste fino.

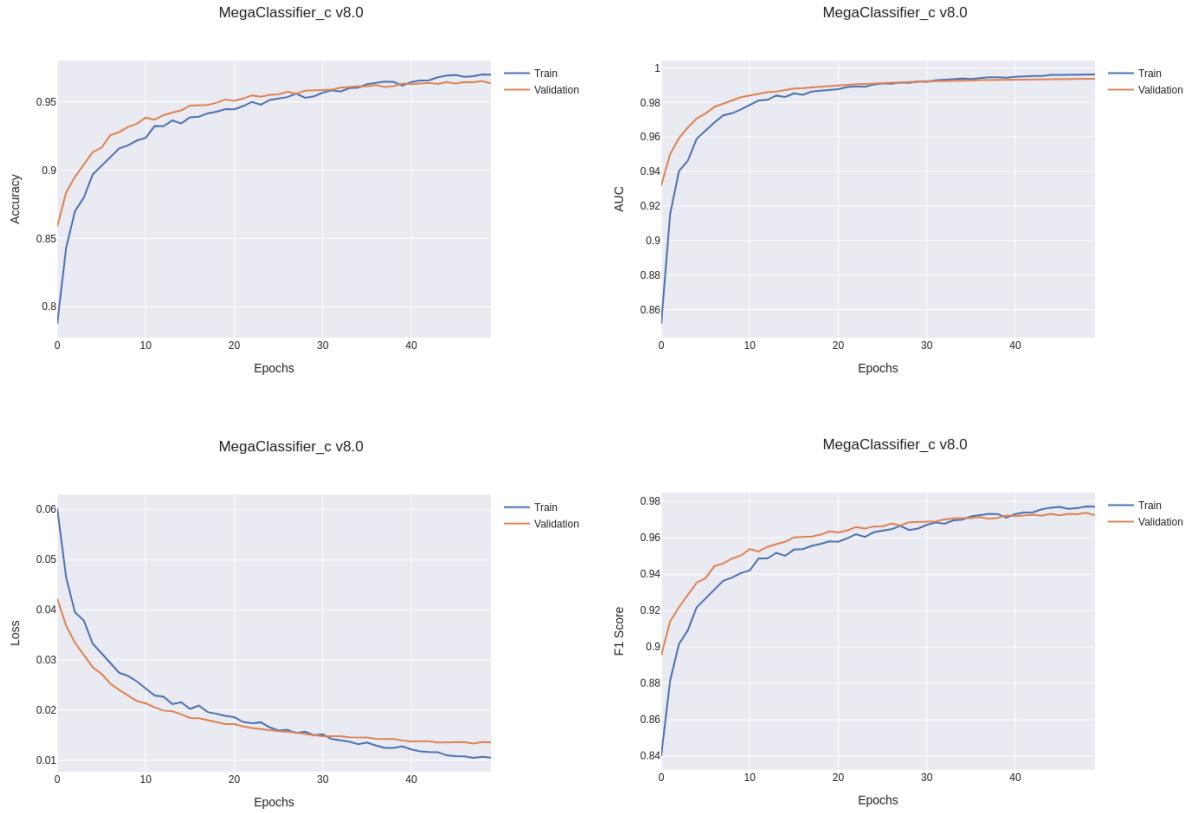


Figura 49: Gráficas de entrenamiento de accuracy, auc, los y f1-score de *MegaClassifier_c_v8.0*

La versión 8.0 de *MegaClassifier_c* representa el cierre óptimo del proceso de ajuste fino, consolidando mejoras tanto en estabilidad como en rendimiento. La combinación de una reducción en la tasa de aprendizaje junto con la aplicación de early stopping permitió un entrenamiento más controlado y eficiente, manteniendo una evolución constante y suave en las métricas clave. Se aprecia una convergencia sostenida a lo largo de las épocas, alcanzando valores sobresalientes en Accuracy, AUC y F1-score, con una pérdida mínima. Las curvas reflejan un modelo bien ajustado, con gran capacidad de generalización, lo que consolida esta versión como la mejor iteración del modelo *MegaClassifier_c*.

Capítulo 5

Resultados y Discusión

En este capítulo se presentan y analizan los resultados obtenidos tras la evaluación de cada una de las versiones desarrolladas de los modelos MegaClassifier_a, MegaClassifier_b y MegaClassifier_c utilizando el conjunto de test. El objetivo principal es contrastar el rendimiento de los modelos en un entorno no visto durante el entrenamiento, evaluando su capacidad de generalización. A lo largo del desarrollo, se han implementado múltiples estrategias de optimización y ajuste fino, abarcando desde la selección de hiperparámetros y funciones de pérdida hasta la inclusión de técnicas como data augmentation, fine-tuning y early stopping. Este capítulo permite visualizar y comparar el impacto de cada una de estas decisiones.

Finalmente, se identifican los modelos más efectivos dentro de cada arquitectura, destacando sus fortalezas y considerando sus limitaciones. Esto no solo aporta una visión cuantitativa del rendimiento, sino que también permite una reflexión crítica sobre el proceso de mejora progresiva aplicado durante la experimentación.

Como vamos a utilizar MegaDetector como preprocesador para las imágenes de dos de los tres modelos, resulta interesante medir la capacidad de rendimiento que tendría sobre el conjunto de pruebas. Como ya explicamos en el Capítulo 4, donde se calcularon los *true positives*, *false positives*, *true negatives* y *false negatives*, podemos crear la matriz de confusión de los test, además de algunas de las métricas indicadas en el apartado 2.2.1 Etapa de entrenamiento. Consideraremos estas métricas el objetivo a batir, puesto que para los modelos MegaClassifier_a y MeglaClassifier_b, obtener peores resultados sería sinónimo de no aprovechar el rendimiento que nos ofrece MegaDetector; en el caso de MegaClassifier_c, dado que la situación es mejorar estos objetivos, estaríamos argumentando que quizás no es del todo ventajoso el uso de MegaDetector en nuestro flujo de trabajo.

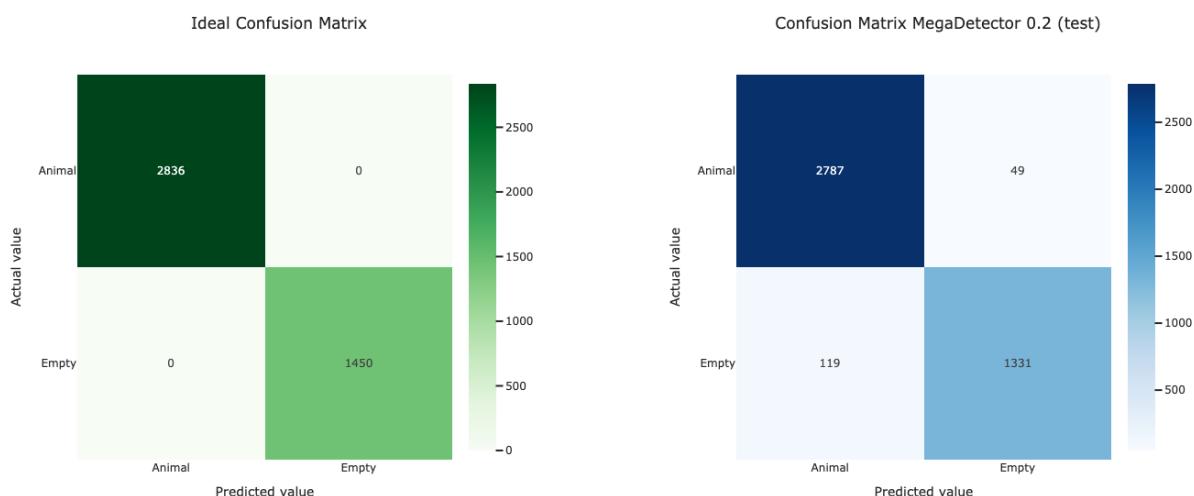


Figura 50: Matriz de confusión ideal sobre el conjunto de prueba y Matriz de confusión de MegaDetector con umbral por defecto

Tabla 3: Métricas objetivo

Modelo	Accuracy	Precision	Recall	Specificity	F1-Score
MD_0.2	0,960803	0,959050	0,982722	0,917931	0,970742

Estas métricas, junto con la matriz de confusión, serán las que deberemos superar o, como mínimo, igualar.

5.1 Resultados en el conjunto de prueba

Para aquellas arquitecturas que utilizan MegaDetector, se han calculado las métricas resultantes de ejecutar el conjunto de entrenamiento con el valor del umbral escogido como óptimo (umbral = 0,0015) para que sirvan de referencia del punto de partida y del que se intenta optimizar para mejorar los valores objetivo.

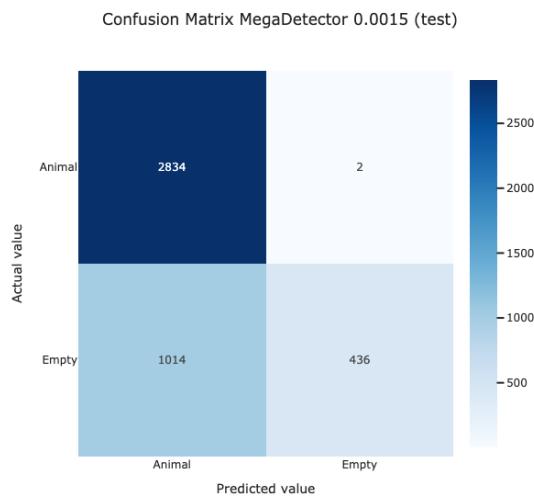


Figura 51: Matriz de confusión de MegaDetector con umbral=0.0015

Tabla 4: Métricas de MegaDetector 0.0015

Modelo	Accuracy	Precision	Recall	Specificity	F1-Score
MD_0.0015	0,762949	0,736486	0,999295	0,300690	0,847995

5.1.1 Resultados MegaClassifier_a

Aunque todas las versiones de MegaClassifier_a en la versión 1 presentan un rendimiento sólido y cercano, la versión v1.0 (batch size = 16) destaca por obtener el AUC más alto (0.9913) y el menor loss (0.117), lo que indica una mayor estabilidad y capacidad para generalizar. Además, logra el F1-Score más alto de todas las variantes, lo que evidencia un buen equilibrio entre precision (0.9598) y recall (0.9701). Frente a v1.1, que tiene un recall ligeramente superior, v1.0 mejora en loss, specificity y AUC, métricas clave para un comportamiento más fiable en entornos con clases desbalanceadas. A pesar de que v1.3 muestra un recall marginalmente mejor, su precision cae significativamente a 0.9503 y su loss se incrementa a 0.1307, lo que podría reflejar cierta inestabilidad en su aprendizaje. En definitiva, v1.0 se posiciona como la más equilibrada y robusta, aunque una posible pega es que su specificity (0.9230) no es la más alta, lo que podría implicar un pequeño margen de mejora en la detección de imágenes negativas.

Tabla 5: Resultados versión 1 de MegaClassifier_a

Modelo	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v1.0	0,953803	0,117191	0,959803	0,970064	0,922973	0,964906	0,991280
v1.1	0,952403	0,120207	0,959803	0,967994	0,922659	0,963881	0,990790
v1.2	0,944937	0,127551	0,958392	0,958392	0,918621	0,958392	0,989803
v1.3	0,949603	0,130708	0,950282	0,972924	0,906992	0,961470	0,989472

Además del sólido desempeño cuantitativo en métricas como loss, F1-score y AUC, la curva ROC de la versión v1.0 muestra una separación clara respecto a la diagonal aleatoria, con una AUC de 0.9913, lo que indica una excelente capacidad discriminativa del modelo incluso bajo umbrales variados. La curva se mantiene pegada al borde superior izquierdo, con un False Positive Rate bajo incluso para True Positive Rates altos, lo que confirma que el modelo minimiza eficazmente los falsos positivos sin sacrificar sensibilidad. Este comportamiento refuerza la elección de v1.0 como el punto de partida más robusto para experimentar en la versión 2, ya que no solo supera en rendimiento general a las variantes v1.1, v1.2 y v1.3, sino que además presenta una relación muy favorable entre precision y recall, siendo más equilibrada y menos volátil frente a cambios en el umbral de decisión. Aunque su specificity no sea la más alta, el comportamiento general en esta curva sugiere que el modelo se comporta de forma más consistente y predecible ante nuevas muestras.

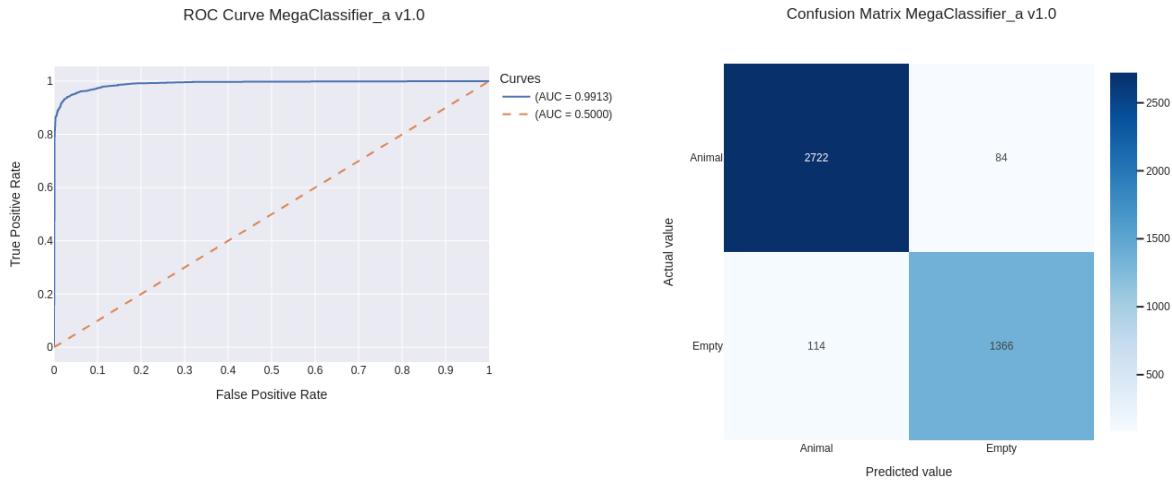


Figura 52: Gráfica Curva ROC y Matriz de confusión de *MegaClassifier_a_v1.0*

En la versión 2 de *MegaClassifier_a*, se evaluó el impacto del optimizador sobre el rendimiento final del modelo, comparando Adam (v2.0), RMSprop (v2.1) y SGD (v2.2). A pesar de que todas las versiones ofrecieron resultados sólidos, v2.0 —entrenada con Adam— destacó ligeramente por encima del resto, alcanzando un AUC de 0.9914, el más alto entre las tres, y manteniendo un excelente equilibrio entre precisión (0.9598), recall (0.9697) y f1-score (0.9647). Además, fue la única configuración que logró mantener el AUC como el loss respecto a la versión anterior v1.0, aunque esto tiene todo el sentido del mundo ya que poseen la misma configuración. El rendimiento del modelo también se respalda visualmente en la curva ROC, donde se observa una alta capacidad discriminativa. Por estos motivos, v2.0 se establece como la mejor opción para continuar con los experimentos posteriores.

Tabla 6: Resultados versión 2 de *MegaClassifier_a*

Modelo	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v2.0	0,953570	0,116801	0,959803	0,969719	0,922921	0,964735	0,991352
v2.1	0,953336	0,120190	0,957334	0,971725	0,918901	0,964476	0,990880
v2.2	0,943304	0,142839	0,956629	0,957642	0,915348	0,957135	0,987679

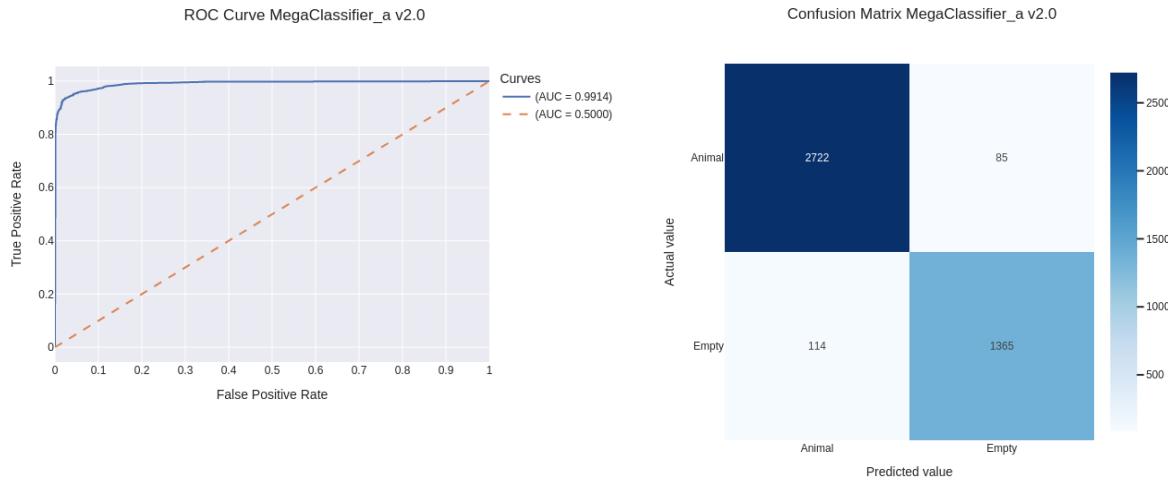


Figura 53: Gráfica Curva ROC y Matriz de confusión de *MegaClassifier_a_v2.0*

En la versión 3 del modelo *MegaClassifier_a*, orientada a evaluar el impacto de distintas funciones de pérdida, se testearon: *BinaryCrossentropy*, *BinaryFocalCrossentropy*, y *BinaryCrossentropy* con pesos. Los resultados evidencian que la variante v3.1, basada en *BinaryFocalCrossentropy*, ofrece el mejor equilibrio global entre métricas: obtiene el valor más bajo de loss (0.036113), el AUC más alto (0.991685), y una F1-Score notablemente sólida (0.963877). Frente al resto de versiones, mejora considerablemente en Specificity (0.930200), sin perjudicar la capacidad de detección de positivos (Recall de 0.963367) ni la precisión.

Tabla 7: Resultados versión 3 de *MegaClassifier_a*

Modelo	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v3.0	0,952170	0,119644	0,954866	0,972352	0,914724	0,963530	0,990970
v3.1	0,952170	0,036113	0,964386	0,963367	0,930200	0,963877	0,991685
v3.2	0,950070	0,131302	0,938999	0,984837	0,890645	0,961372	0,991166

Aunque v3.2 presenta un Recall superior (0.984837), lo hace a costa de una notable caída en Specificity (0.890645) y un incremento del loss (0.131302), lo que sugiere un modelo menos equilibrado y con mayor tasa de falsos positivos. La versión v3.0, por su parte, muestra un comportamiento intermedio, pero sin destacar sobre v3.1. En resumen, v3.1 representa la mejor alternativa, al mantener una excelente capacidad discriminativa y estabilidad, posicionándose como la opción más robusta de cara a fases posteriores del pipeline.

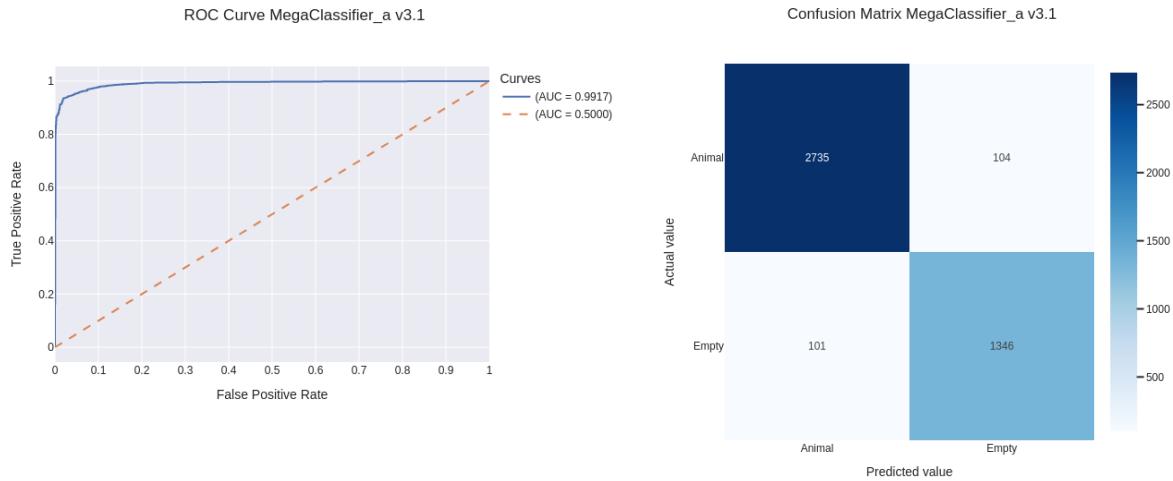


Figura 54: Gráfica Curva ROC y Matriz de confusión de *MegaClassifier_a_v3.1*

La versión v4.1 ($\gamma=2.0$) fue la que logró un mejor equilibrio global en las métricas evaluadas: presentó una precisión del 94.99%, un recall del 97.11%, y un F1-score del 96.04%, junto con una de las loss más bajas (0.0405) de todas las versiones analizadas. Además, su AUC de 0.9891 refuerza su capacidad discriminativa. A pesar de que su specificity (90.61%) es algo inferior a la referencia de MD_0.2, la mejora en recall permite reducir el número de falsos negativos, algo clave en este problema. La matriz de confusión muestra un buen equilibrio entre clases, validando visualmente estos resultados. Por tanto, v4.1 se posiciona como la versión más sólida para esta etapa.

Tabla 8: Resultados versión 4 de *MegaClasifier_a*

Modelo	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v4.0	0,946804	0,072863	0,953808	0,965382	0,911725	0,959560	0,988474
v4.1	0,948203	0,040527	0,949929	0,971161	0,906085	0,960428	0,989086
v4.2	0,947737	0,024058	0,950635	0,969784	0,907039	0,960114	0,989482

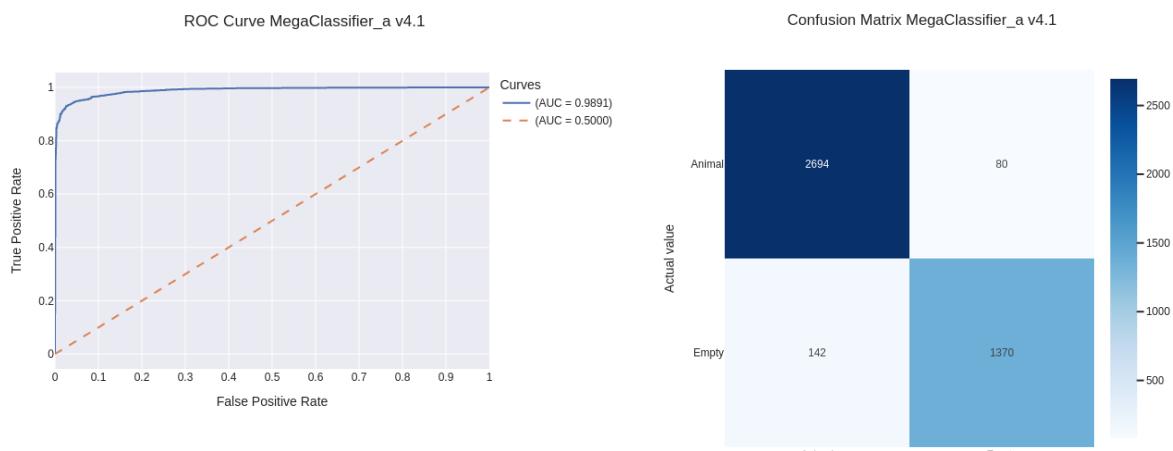


Figura 55: Gráfica Curva ROC y Matriz de confusión de *MegaClassifier_a_4.1*

De entre las variantes, la versión v5.0 resultó ser la más equilibrada, alcanzando un excelente AUC de 0.9894 y un F1-Score de 0.9597. Además, su matriz de confusión muestra un desempeño sólido con una buena capacidad para clasificar tanto imágenes con animales como vacías, con una tasa de verdaderos positivos alta (2702) y relativamente pocos falsos negativos (93). La curva ROC reafirma esta robustez, mostrando una alta sensibilidad en todos los umbrales. Aunque presenta una ligera caída en Specificity respecto a versiones anteriores, este sacrificio se ve compensado por una mayor Recall, lo que es deseable en contextos donde minimizar la tasa de falsos negativos es prioritario.

Tabla 9: Resultados de versión 5 de *MegaClassifier_a*

Modelo	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v5.0	0,947037	0,040047	0,952750	0,966726	0,910127	0,959687	0,989364
v5.1	0,945870	0,042837	0,947814	0,969697	0,902246	0,958631	0,987455

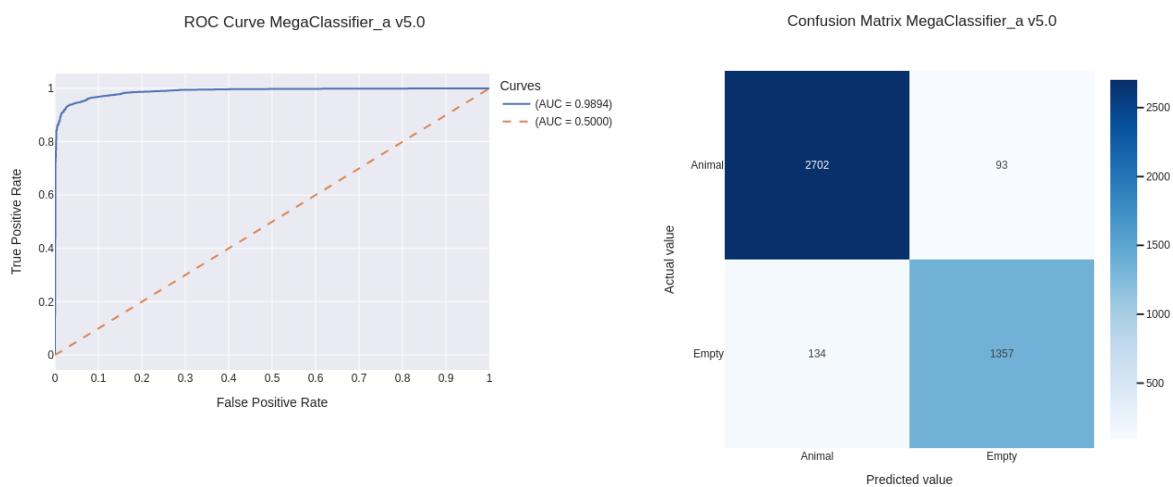


Figura 56: Gráfica Curva ROC y Matriz de confusión de *MegaClassifier_a_5.0*

De todas las variantes, la versión v6.1 se consolidó como la mejor opción gracias a un balance óptimo entre rendimiento y estabilidad. Logró un AUC de 0.9896 y un F1-Score de 0.9619, valores que la posicionan por encima del resto en cuanto a discriminación y equilibrio entre precisión y sensibilidad. A pesar de una Specificity ligeramente inferior a v6.0, la mejora en Recall y la baja pérdida (loss) sugieren una capacidad más eficaz para identificar correctamente las imágenes con animales. En este contexto, v6.1 se perfila como una base robusta para las siguientes etapas de fine-tuning.

Tabla 10: Resultados de versión 6 de MegaClassifier_a

Modelo	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v6.0	0,948203	0,039309	0,956629	0,964794	0,916554	0,960694	0,989751
v6.1	0,950070	0,039531	0,954866	0,969220	0,914209	0,961989	0,989641
v6.2	0,932105	0,051857	0,938999	0,957569	0,885050	0,948193	0,981188
v6.3	0,939104	0,047640	0,956982	0,951279	0,914864	0,954122	0,984788

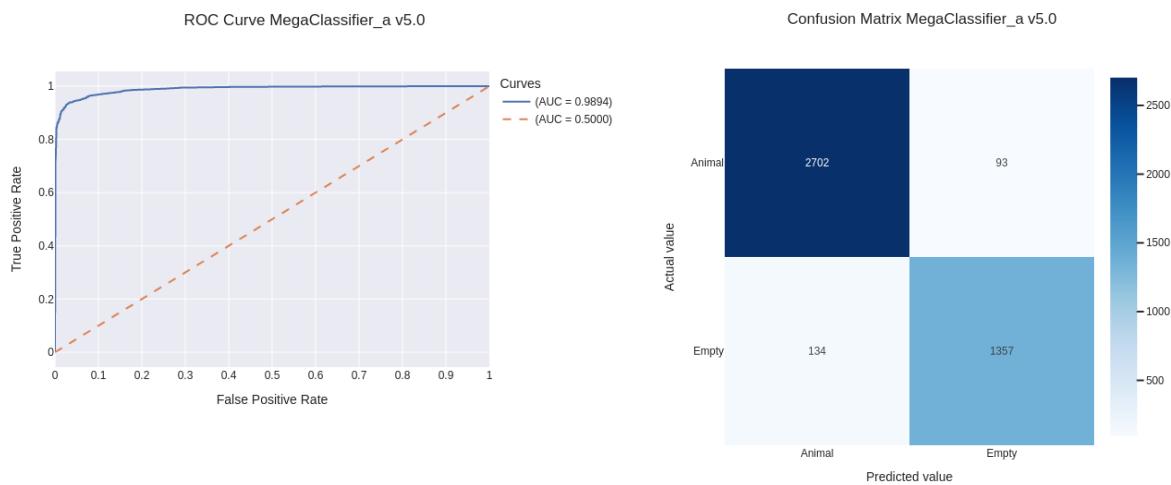


Figura 57: Gráfica Curva ROC y Matriz de confusión de MegaClassifier_a_6.1

La versión 7.3 del modelo MegaClassifier_a se consolida como la mejor de toda la serie evaluada, alcanzando un rendimiento sobresaliente tras aplicar un proceso de fine-tuning completo y ajustar el umbral de decisión a 0.542. Este ajuste permite maximizar el F1-Score hasta un 0.9754, con una precisión de 96.36 % y un recall de 98.56 %, mostrando un equilibrio excepcional entre falsos positivos y negativos. La matriz de confusión refleja tan solo 40 falsos negativos frente a 103 falsos positivos, mejorando sustancialmente la sensibilidad del sistema sin comprometer la especificidad (93.19 %). Además, se consigue un AUC de 0.9954, evidenciando una discriminación casi perfecta entre clases. Todo esto respalda a la versión 7.3 como la más robusta y eficaz dentro del marco de MegaClassifier_a.

Tabla 11: Resultados versión 7 de MegaClassifier_a

Modelo	Umbral	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	-	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v7.0	0,5000	0,961503	0,031393	0,963681	0,977818	0,930919	0,970698	0,031393
v7.0	0,5283	0,960103	0,031393	0,956629	0,982615	0,919344	0,969448	0,031393
v7.1	0,5000	0,961736	0,030456	0,965092	0,976802	0,933288	0,970912	0,031393
v7.1	0,5410	0,962203	0,030456	0,958745	0,983719	0,923127	0,971071	0,031393
v7.2	0,5000	0,965702	0,031018	0,975317	0,972916	0,951490	0,974115	0,031393
v7.2	0,5863	0,965469	0,031018	0,962623	0,984848	0,929987	0,973609	0,031393
v7.3	0,5000	0,966869	0,029603	0,968618	0,981071	0,940108	0,974805	0,031393
v7.3	0,5422	0,966636	0,029603	0,963681	0,985575	0,931923	0,974505	0,031393

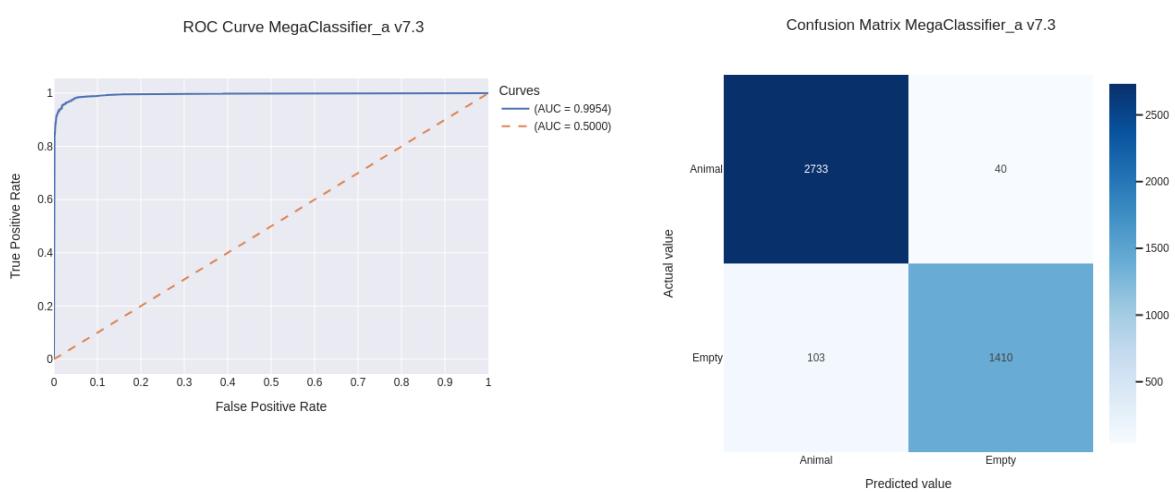


Figura 58: Gráfica Curva ROC y Matriz de confusión de MegaClassifier_a_7.3 con umbral=0.5422

La versión 8.0 de MegaClassifier_a representa la culminación del proceso de optimización del modelo, integrando todos los aprendizajes previos. Gracias a la incorporación de early stopping y una reducción del learning rate, se ha conseguido un entrenamiento más estable y prolongado, lo que permite al modelo refinar su capacidad predictiva sin incurrir en sobreajuste.

En términos de rendimiento, esta versión se posiciona como una de las mejores evaluadas. Supera o iguala al resto de modelos en todas las métricas clave: accuracy (96.22%), precision (hasta 96.5%), recall (98.3%) y F1-score (97.1%), manteniendo además una AUC excelente de 0.9939. La curva ROC muestra un comportamiento casi ideal, y la matriz de confusión indica una reducción clara de falsos negativos respecto a versiones anteriores, cumpliendo el objetivo de minimizar la tasa de error en la detección de animales.

Por todo ello, consideramos a la versión v8.0 con umbral 0.5351 como una buena candidata para la versión final de MegaClassifier_a

Tabla 12: Resultados versión 8 de MegaClassifier_a

Modelo	Umbral	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	-	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v8.0	0,5000	0,962203	0,031476	0,965444	0,977159	0,933962	0,971266	0,993940
v8.0	0,5351	0,962203	0,031476	0,959450	0,983020	0,924242	0,971092	0,993940

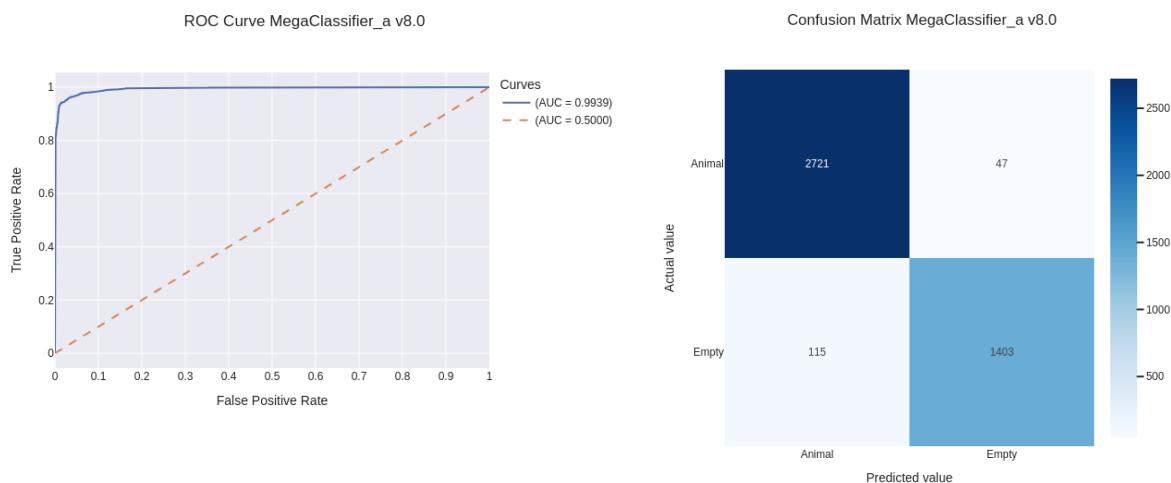


Figura 59: Gráfica Curva ROC y Matriz de confusión de MegaClassifier_a_8.0 con umbral=0.5351

5.1.2 Resultados MegaClassifier_b

La versión seleccionada para representar a MegaClassifier_b es la v1.1, ya que presenta un equilibrio sólido entre todas las métricas clave. Aunque la v1.0 ofrece una ligera ventaja en términos de accuracy (0.9496 vs. 0.9491), la v1.1 mejora ligeramente el recall (0.9746 vs. 0.9736) y mantiene una precision prácticamente idéntica (0.9478 vs. 0.9496), lo cual es especialmente relevante en contextos donde minimizar los falsos negativos es prioritario.

Tabla 13: Resultados versión 1 de MegaClassifier_b

Modelo	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v1.0	0,949137	0,118498	0,947814	0,974619	0,903141	0,961030	0,990769
v1.1	0,949603	0,118768	0,949577	0,973608	0,905921	0,961442	0,990850
v1.2	0,948437	0,121157	0,959450	0,962504	0,921179	0,960975	0,990044
v1.3	0,947037	0,127469	0,955571	0,964070	0,914576	0,959802	0,989081

Además, la curva ROC de v1.1 muestra un excelente comportamiento con un AUC de 0.9909, lo que indica una alta capacidad discriminativa. Su matriz de confusión refleja una distribución equilibrada de errores, con 70 falsos positivos y 148 falsos negativos, cifras competitivas frente a otras versiones.

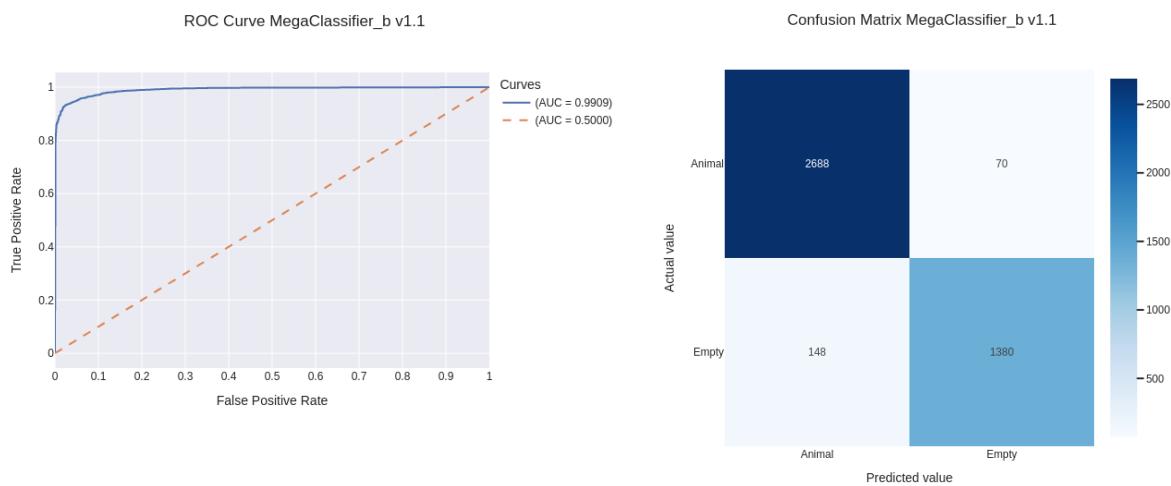


Figura 60: Gráfica Curva ROC y Matriz de confusión de MegaClassifier_b_1.1

Aunque la especificidad (0.9031) es ligeramente inferior a otras variantes, el F1-Score elevado (0.9610) demuestra que la versión v1.1 mantiene una buena armonía entre precisión y sensibilidad. Esta combinación la convierte en la opción más robusta para continuar el desarrollo del modelo.

Basándonos en los resultados mostrados en la tabla y en los gráficos complementarios de la matriz de confusión y la curva ROC, la versión v2.1 de MegaClassifier_b se presenta como la más equilibrada y robusta para representar este modelo.

Aunque la versión v2.0 posee una ligera ventaja en métricas como Precision y Specificity, la v2.1 consigue compensarlo con una mejora en Recall, F1-Score y Accuracy, lo cual sugiere una mejor capacidad general del modelo para identificar correctamente tanto clases positivas como negativas. Además, la AUC apenas se ve afectada entre ambas versiones (0.9898 vs. 0.9899), por lo que no compromete la calidad del discriminador.

La matriz de confusión de la v2.1 confirma un buen equilibrio entre False Positives y False Negatives, sin mostrar sesgos relevantes hacia alguna de las dos clases, y la curva ROC evidencia una alta capacidad de discriminación entre clases. Por tanto, v2.1 se considera la versión óptima de esta serie para continuar el desarrollo o servir como modelo base.

Tabla 14: Resultados versión 2 de MegaClassifier_b

Modelo	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v2.0	0,948437	0,121675	0,957687	0,964146	0,918312	0,960906	0,989952
v2.1	0,949603	0,121691	0,955571	0,967857	0,915209	0,961675	0,989828
v2.2	0,934438	0,178185	0,938999	0,961025	0,885809	0,949884	0,980745

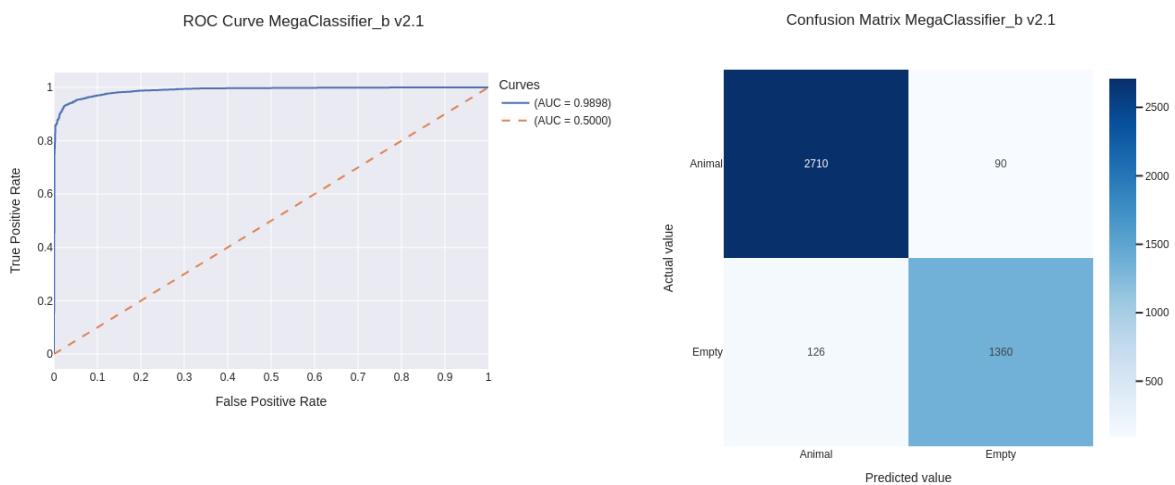


Figura 61: Gráfica Curva ROC y Matriz de confusión de MegaClassifier_b_2.1

Entre las versiones del modelo MegaClassifier_b, la versión v3.1 se posiciona como la más equilibrada y robusta. Aunque la v2.1 también mostró muy buenos resultados, v3.1 destaca por presentar el menor valor de loss (0.0316), lo que indica una mejor optimización del modelo durante el entrenamiento. Además, mantiene un excelente equilibrio entre precisión (0.9531), recall (0.9719) y f1-score (0.9624), lo que la hace especialmente adecuada para evitar tanto falsos positivos como falsos negativos. Su matriz de confusión refuerza esta estabilidad: comete menos falsos negativos que v2.1 (133 vs. 126) pero también reduce los falsos positivos (78 vs. 90), lo que contribuye a una mayor especificidad sin sacrificar recall. Finalmente, su curva ROC y el valor de AUC (0.9906) confirman la alta capacidad discriminativa del modelo.

Por tanto, MegaClassifier_b v3.1 representa una evolución positiva del modelo, ofreciendo métricas sólidas y un comportamiento equilibrado en la clasificación, siendo la mejor candidata para representar esta arquitectura.

Tabla 15: Resultados versión 3 de MegaClassifier_b

Modelo	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v3.0	0,949603	0,120743	0,954161	0,969198	0,912985	0,961620	0,990051
v3.1	0,950770	0,031587	0,953103	0,971953	0,911628	0,962435	0,990582
v3.2	0,948670	0,123565	0,944993	0,976676	0,898833	0,960573	0,990069

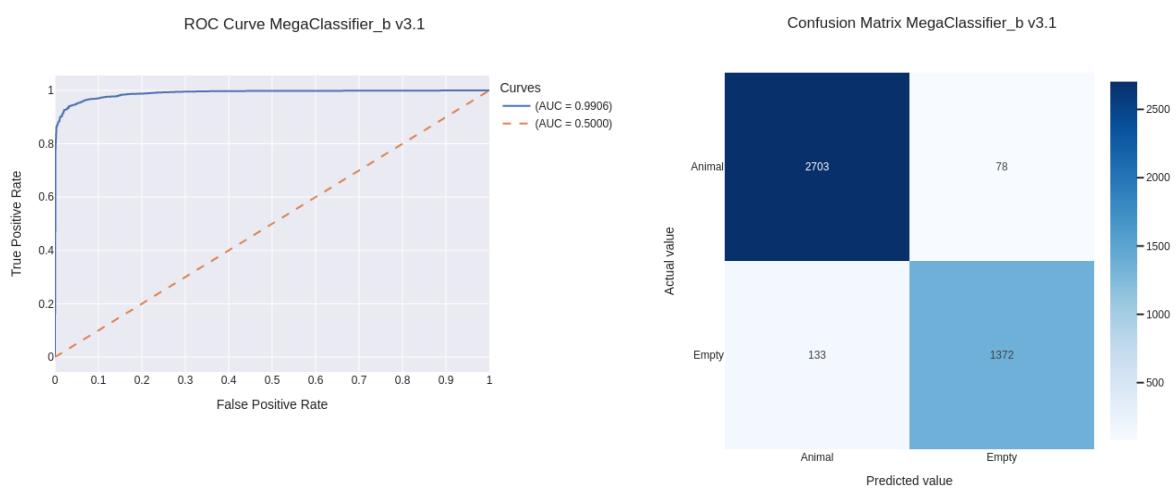


Figura 62: Gráfica Curva ROC y Matriz de confusión de MegaClassifier_b_3.1

Entre todas las versiones de MegaClassifier_b, la versión v4.2 se presenta como la opción más sólida para representar al modelo final. A pesar de que v3.1 ofrece métricas ligeramente superiores en accuracy (0.950770) y f1-score (0.962435), v4.2 compensa con una pérdida mucho menor (loss de 0.017183 frente a 0.031587) y un equilibrio más fino entre precision (0.959097) y recall (0.962151), acompañado de una especificidad ligeramente superior (0.920493). Además, v4.2 muestra un comportamiento más estable, evidenciado por su curva ROC con un AUC de 0.9895, prácticamente igual a la de v3.1 pero lograda con menor pérdida. La matriz de confusión también revela una distribución de errores muy contenida: aunque v3.1 comete menos falsos negativos, v4.2 logra reducir significativamente los falsos positivos, favoreciendo un menor número de imágenes vacías erróneamente etiquetadas como animales, lo cual es deseable en muchos contextos de aplicación real.

En resumen, v4.2 es la versión que consigue el mejor compromiso entre rendimiento general, consistencia y eficiencia del modelo, justificando su selección como representante final de MegaClassifier_b.

Tabla 16: Resultados versión 4 de MegaClassifier_b

Modelo	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v4.0	0,946337	0,065610	0,953808	0,964693	0,911606	0,959220	0,988771
v4.1	0,946570	0,033405	0,956276	0,962726	0,915589	0,959491	0,989250
v4.2	0,947970	0,017183	0,959097	0,962151	0,920493	0,960622	0,989467

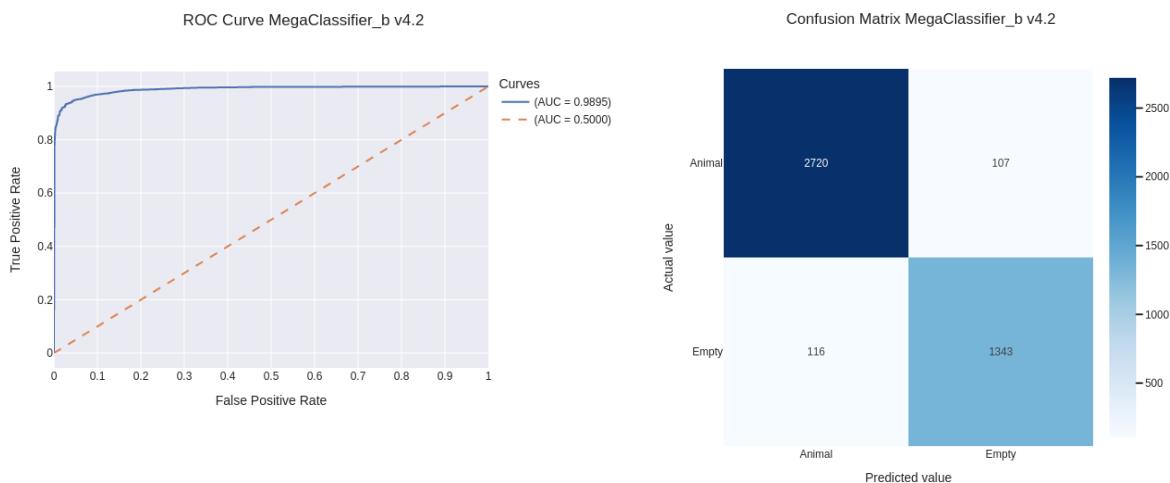


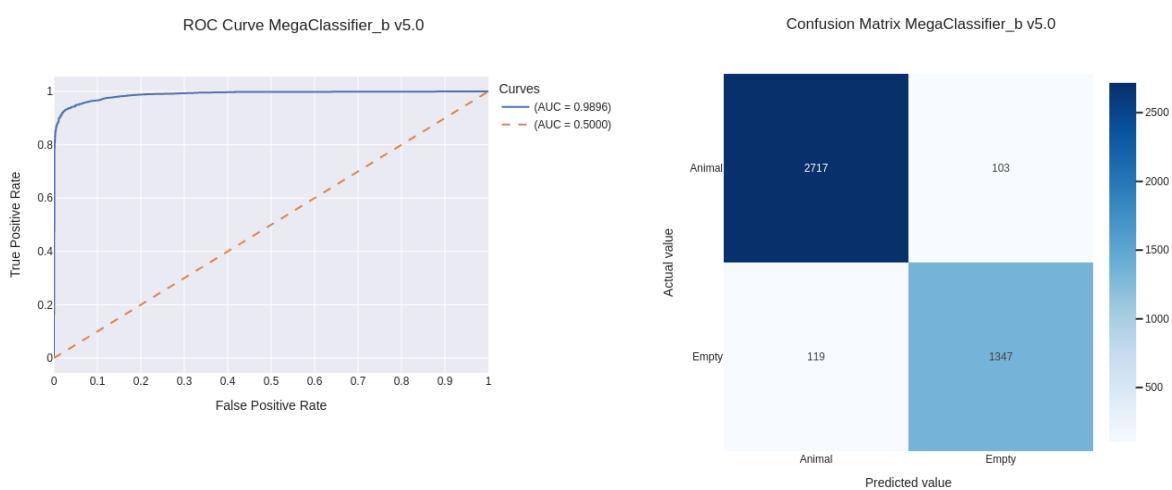
Figura 63: Gráfica Curva ROC y Matriz de confusión de MegaClassifier_b_4.2

La versión v5.0 ofrece un rendimiento muy equilibrado y competitivo en todos los frentes. Presenta un accuracy de 0.9482 y un loss mínimo de 0.0171, el más bajo entre las variantes, lo que indica una alta confianza del modelo en sus predicciones. Además, logra una precision de 0.9580, manteniendo una tasa de falsos positivos baja. Aunque el recall (0.9635) no es el más alto, sí está muy próximo al máximo, logrando una buena cobertura de los casos positivos sin comprometer excesivamente la precisión. Tanto la specificity (0.9635) como el AUC (0.9896) son sobresalientes, reflejando una excelente capacidad para discriminar entre clases. El F1-Score de 0.9608 refuerza esta solidez en el balance entre precision y recall. La matriz de confusión respalda esta consistencia, con solo 103 falsos negativos y 119 falsos positivos sobre el total del conjunto de prueba. Además, la curva ROC muestra una forma ideal con un AUC muy cercano a 1, lo que valida la alta discriminación del modelo.

En conjunto, v5.0 demuestra ser la versión más sólida, con un excelente equilibrio entre sensibilidad y especificidad, lo que la convierte en la mejor opción para representar a *MegaClassifier_b* en esta etapa.

*Tabla 17: Resultados versión 5 de *MegaClassifier_b**

Modelo	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v5.0	0,948203	0,017055	0,958039	0,963475	0,963475	0,960750	0,989552
v5.1	0,944004	0,019107	0,941114	0,973377	0,973377	0,956974	0,987441



*Figura 64: Gráfica Curva ROC y Matriz de confusión de *MegaClassifier_b* 5.0*

La versión seleccionada para representar a MegaClassifier_b es la v6.0, ya que ofrece un equilibrio destacado entre todas las métricas de evaluación. Con una accuracy del 94,75 %, un F1-Score de 0,9599 y un AUC de 0,9892, este modelo demuestra una gran capacidad para clasificar correctamente tanto imágenes con animales como vacías. Además, mantiene una precision alta (0,9509) y un recall robusto (0,9691), lo que refleja una correcta identificación de los casos positivos, minimizando los falsos negativos. Aunque otras versiones exploran configuraciones similares, la v6.0 logra consolidar una specificity competitiva (0,9075) y una pérdida (loss) baja, lo que evidencia un modelo no solo preciso, sino también confiable y bien generalizado.

Tabla 18: Resultados versión 6 de MegaClassifier_b

Modelo	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v6.0	0,947503	0,017308	0,950987	0,969098	0,907518	0,959957	0,989213
v6.1	0,946570	0,017369	0,950635	0,968043	0,906729	0,959260	0,989042
v6.2	0,944404	0,019773	0,939351	0,975110	0,889318	0,956897	0,986473
v6.3	0,942137	0,019940	0,952750	0,959517	0,908844	0,956122	0,985483

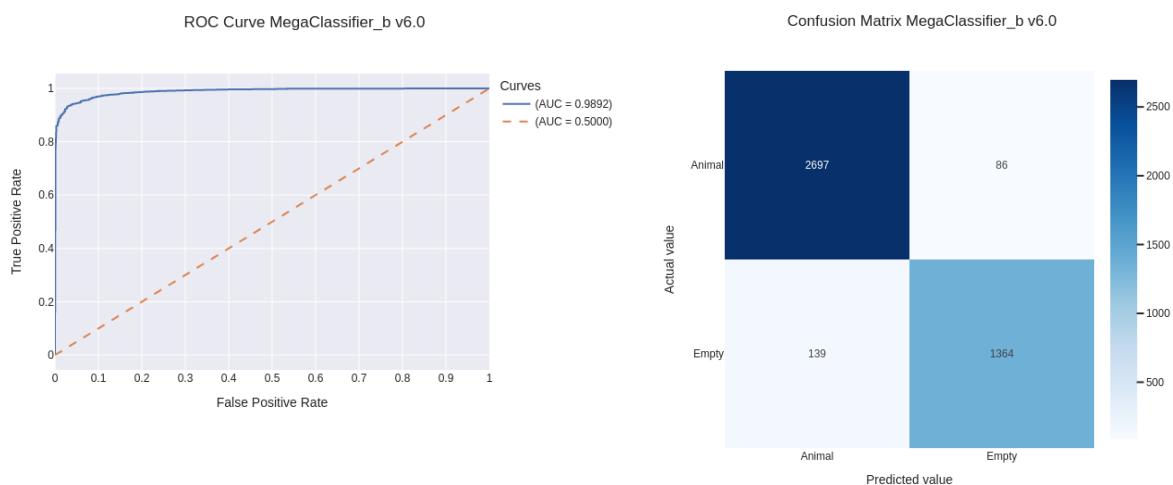


Figura 65: Gráfica Curva ROC y Matriz de confusión de MegaClassifier_b_6.0

Para la versión 7 de MegaClassifier_b, se realizaron distintas pruebas variando el umbral de decisión para optimizar el rendimiento del modelo. La variante v7.3, con un umbral ajustado a 0.5553, fue la que ofreció los mejores resultados globales, alcanzando una accuracy de 0.960803, una precision de 0.955571, un recall de 0.984738 y un AUC de 0.994050. La specificity alcanzada fue de 0.917862 y el F1-score fue de 0.969936, consolidando a esta versión como la más equilibrada. Además, la matriz de confusión muestra una reducción significativa en los falsos negativos (solo 42), lo cual es clave para aplicaciones donde la detección de animales es prioritaria. No obstante, aunque este modelo destaca entre las versiones de MegaClassifier_b, se decidió no continuar con una versión 8, ya que los resultados obtenidos con la versión 7.3 de MegaClassifier_a fueron aún superiores en casi todas las métricas. Por ello, se priorizó el desarrollo y refinamiento de la variante a, dado su mayor potencial de mejora.

Tabla 19: Resultados versión 7 de MegaClassifier_b

Modelo	Umbral	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	-	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v7.0	0,5000	0,953570	0,014690	0,961213	0,968384	0,925221	0,964785	0,992305
v7.0	0,5310	0,953803	0,014690	0,952398	0,977207	0,911301	0,964643	0,992305
v7.1	0,5000	0,958236	0,013388	0,963681	0,972944	0,930264	0,968291	0,993544
v7.1	0,5240	0,958936	0,013388	0,958039	0,979452	0,921296	0,968627	0,993544
v7.2	0,5000	0,959636	0,013471	0,963329	0,975366	0,929966	0,969310	0,993540
v7.2	0,5089	0,959869	0,013471	0,961213	0,977762	0,926569	0,969417	0,993540
v7.3	0,5000	0,963136	0,013242	0,967913	0,976174	0,938263	0,972025	0,994050
v7.3	0,5553	0,960803	0,013242	0,955571	0,984738	0,917862	0,969936	0,994050

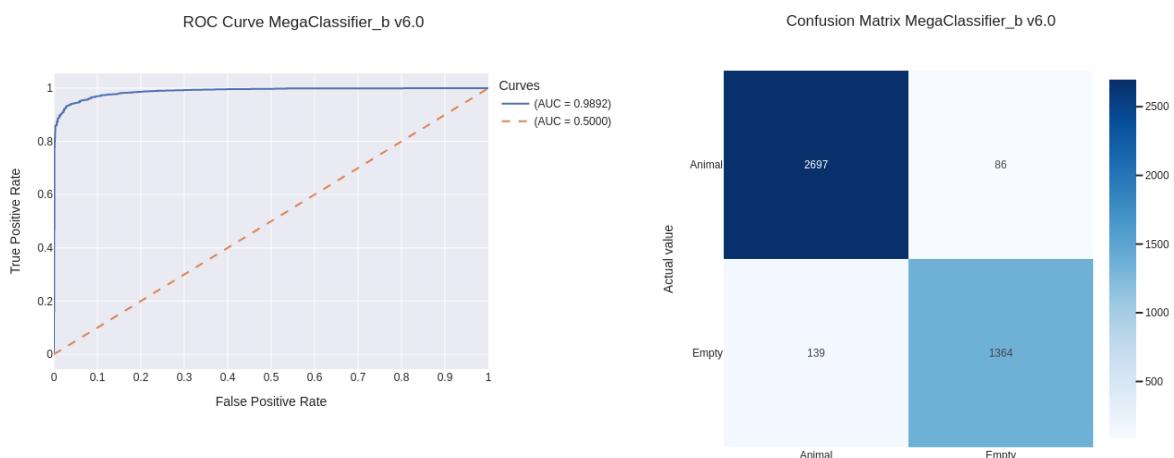


Figura 66: Gráfica Curva ROC y Matriz de confusión de MegaClassifier_b_7.3 con umbral 0.5553

5.1.3 Resultados MegaClassifier_c

Entre las cuatro versiones experimentadas, la v1.0 destaca como la opción más equilibrada, con un AUC de 0.9913, la mayor precisión (0.9468) junto con un recall elevado (0.9746) y una pérdida relativamente baja (0.117). Si bien las versiones v1.1 y v1.2 también presentan métricas consistentes, v1.0 muestra un menor compromiso entre precisión y recall, logrando el F1-score más alto (0.9605) de la serie. La matriz de confusión refuerza esta elección al evidenciar un número moderado de falsos negativos y falsos positivos, mientras que la curva ROC confirma su capacidad discriminativa frente al clasificador aleatorio. En consecuencia, v1.0 se selecciona como el modelo representativo de esta primera etapa en la arquitectura MegaClassifier_c.

Tabla 20: Resultados versión 1 de MegaClassifier_c

Modelo	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v1.0	0,948437	0,116997	0,946756	0,974592	0,901372	0,960472	0,991312
v1.1	0,948437	0,118076	0,954513	0,967131	0,913248	0,960781	0,990797
v1.2	0,946804	0,123052	0,950282	0,968728	0,906250	0,959416	0,990322
v1.3	0,942837	0,130903	0,952398	0,960868	0,908475	0,956614	0,988802

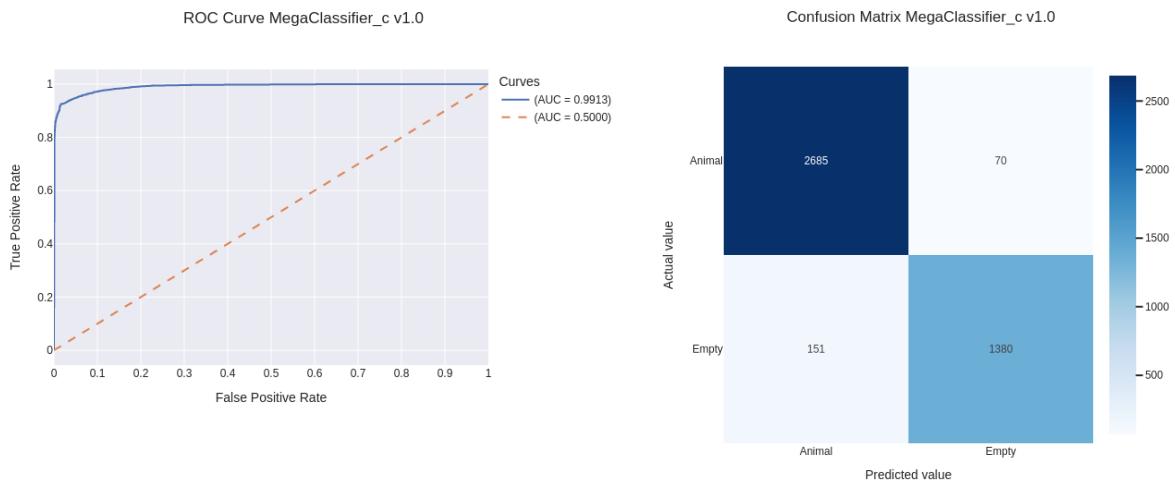


Figura 67: Gráfica Curva ROC y Matriz de confusión de MegaClassifier_c_1.0

Para la versión 2 del modelo MegaClassifier_c, la selección del modelo más adecuado recae en la versión v2.0, la cual logra un balance notable entre todas las métricas evaluadas. Aunque su accuracy (0.9484) es idéntico al de la versión v2.1, v2.0 obtiene un menor valor de loss (0.1178 vs. 0.1208), lo que indica una mayor confianza del modelo en sus predicciones. Además, su curva ROC muestra un excelente AUC de 0.9910, y en cuanto a la matriz de confusión, comete menos falsos negativos que v2.1 (137 vs. 149), algo especialmente relevante en escenarios donde evitar este tipo de error es prioritario. En contrapartida, v2.1 alcanza una mejor especificidad, pero con una leve caída en el recall. Por tanto, v2.0 demuestra un mejor equilibrio global, especialmente en términos de recall y pérdida, consolidándose como la versión más robusta para continuar con la siguiente fase de experimentación.

Tabla 21: Resultados versión 2 de MegaClassifier_c

Modelo	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v2.0	0,948437	0,117769	0,951693	0,969817	0,908849	0,960669	0,991020
v2.1	0,948437	0,120822	0,965444	0,957008	0,931228	0,961208	0,990690
v2.2	0,933971	0,160566	0,945346	0,954432	0,895058	0,949867	0,984198

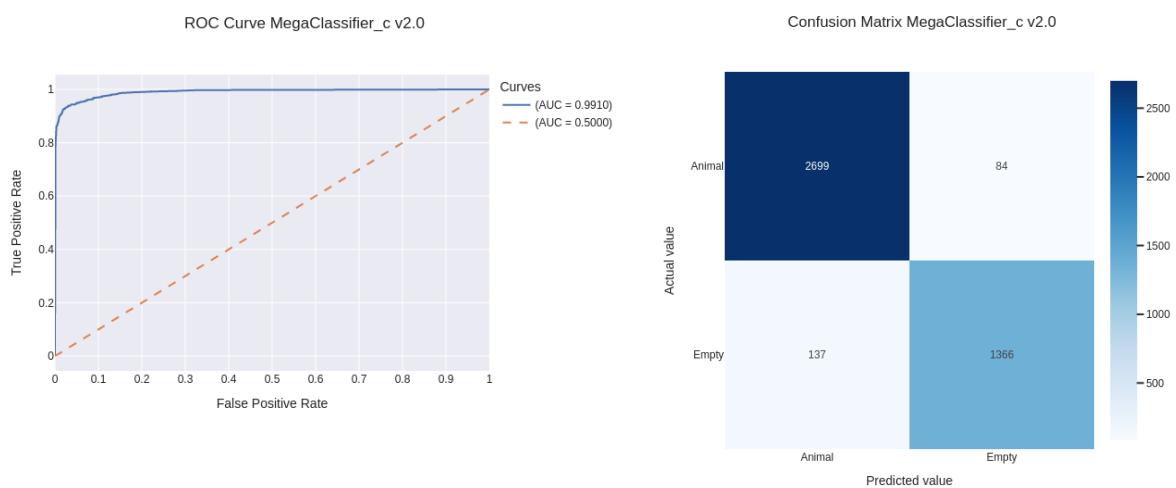


Figura 68: Gráfica Curva ROC y Matriz de confusión de MegaClassifier_c_2.0

En la versión 3 de MegaClassifier_c se exploraron distintas funciones de pérdida con el objetivo de mejorar el rendimiento del modelo. Tras evaluar las métricas de desempeño y analizar tanto la curva ROC como la matriz de confusión, la versión v3.1 se posicionó como la más destacada. Este modelo alcanzó el mayor F1-Score (0.963720) y AUC (0.991477), lo que refleja un excelente equilibrio entre precisión y sensibilidad. Además, su pérdida fue la más baja entre todas las versiones evaluadas (0.029810), indicando una mayor confianza del modelo en sus predicciones. La matriz de confusión muestra un buen compromiso entre verdaderos positivos y verdaderos negativos, con tan solo 100 falsos negativos y 106 falsos positivos, una mejora notable respecto a otras configuraciones. Todo esto posiciona a la versión v3.1 como la mejor candidata para continuar con el desarrollo de MegaClassifier_c.

Tabla 22: Resultados versión 3 de MegaClassifier_c

Modelo	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v3.0	0,950070	0,117052	0,948872	0,975000	0,904980	0,961758	0,991191
v3.1	0,951937	0,029810	0,964739	0,962702	0,930748	0,963720	0,991477
v3.2	0,945404	0,129326	0,931241	0,985448	0,878580	0,957578	0,990757

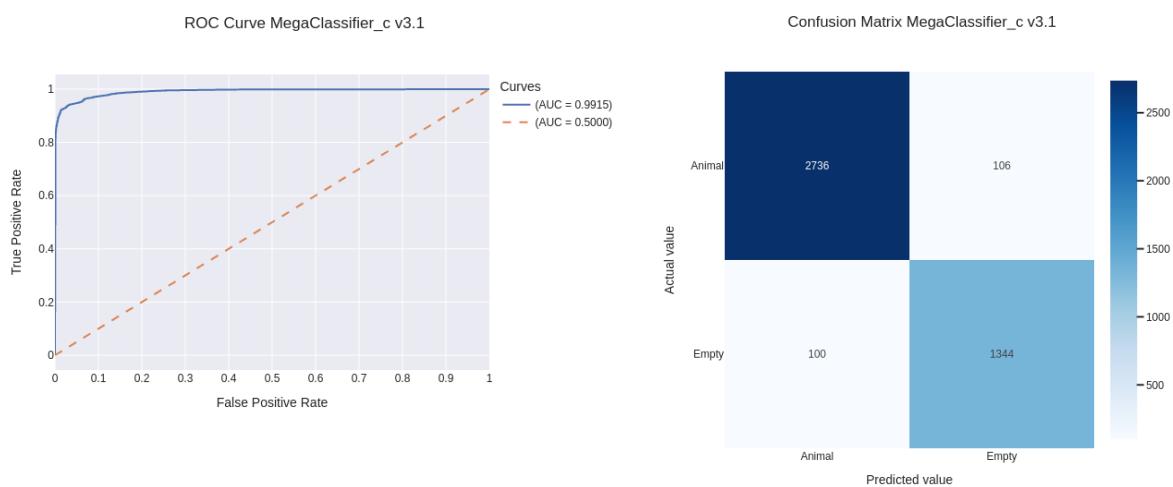


Figura 69: Gráfica Curva ROC y Matriz de confusión de MegaClassifier_c_3.1

El modelo v4.2 se posiciona como el más robusto dentro de esta etapa de experimentación. No solo logra un AUC de 0.9893, cercano a los máximos históricos alcanzados por esta variante, sino que también consigue un excelente equilibrio entre sensibilidad y especificidad (0.9597 y 0.9151, respectivamente). Esto se traduce en una capacidad sólida tanto para detectar correctamente las instancias positivas (presencia de animal), como para evitar falsos positivos en imágenes vacías. La matriz de confusión refuerza este diagnóstico, mostrando una baja tasa de errores en ambos tipos de clases: 114 falsos positivos y 124 falsos negativos, en un contexto de miles de imágenes. Este buen desempeño se refleja también en métricas agregadas como el F1-score de 0.9579 y una precisión de 0.9562, lo que denota que el modelo no solo predice con acierto, sino que mantiene una consistencia notable.

A pesar de que los resultados de v4.2 son muy prometedores, aún no se consideran como versión final, ya que la exploración continuará hasta la versión 8. Sin embargo, esta versión establece una base muy sólida, y será una referencia clave para contrastar los avances en las iteraciones siguientes.

Tabla 23: Resultados versión 4 de MegaClassifier_c

Modelo	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v4.0	0,939337	0,070791	0,947109	0,960658	0,899329	0,953835	0,987464
v4.1	0,942371	0,034622	0,950987	0,961497	0,906144	0,956213	0,988822
v4.2	0,944470	0,017445	0,956276	0,959660	0,915068	0,957965	0,989314

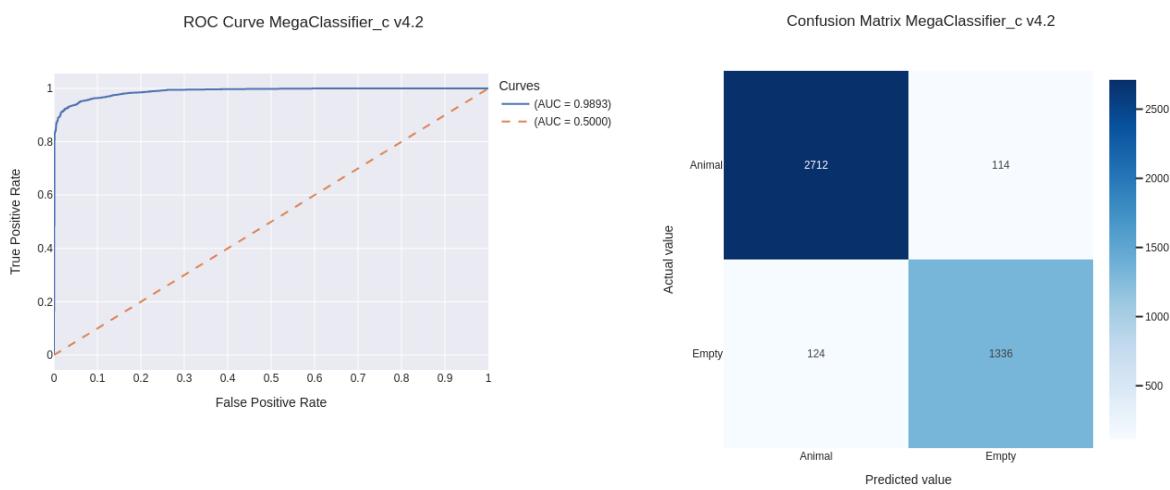


Figura 70: Gráfica Curva ROC y Matriz de confusión de MegaClassifier_c_4.2

Para la versión 5 de MegaClassifier_c, se ha seleccionado la variante v5.1 como la opción más adecuada para representar esta rama del modelo. Esta decisión se basa en su desempeño más equilibrado en métricas clave, destacando una precisión de 0.9559, un F1-Score de 0.9579 y una specificity de 0.9145, superior a la de su contraparte v5.0. La versión v5.1 presenta una mejor capacidad para minimizar falsos positivos sin comprometer la detección de verdaderos positivos. Estos resultados, confirmados también por la matriz de confusión, respaldan a v5.1 como la versión más robusta y fiable de esta etapa del modelo.

Tabla 24: Resultados versión 5 de MegaClassifier_c

Modelo	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v5.0	0,933738	0,020612	0,948519	0,951202	0,899863	0,949859	0,984892
v5.1	0,944470	0,017280	0,955924	0,959986	0,914501	0,957951	0,989616

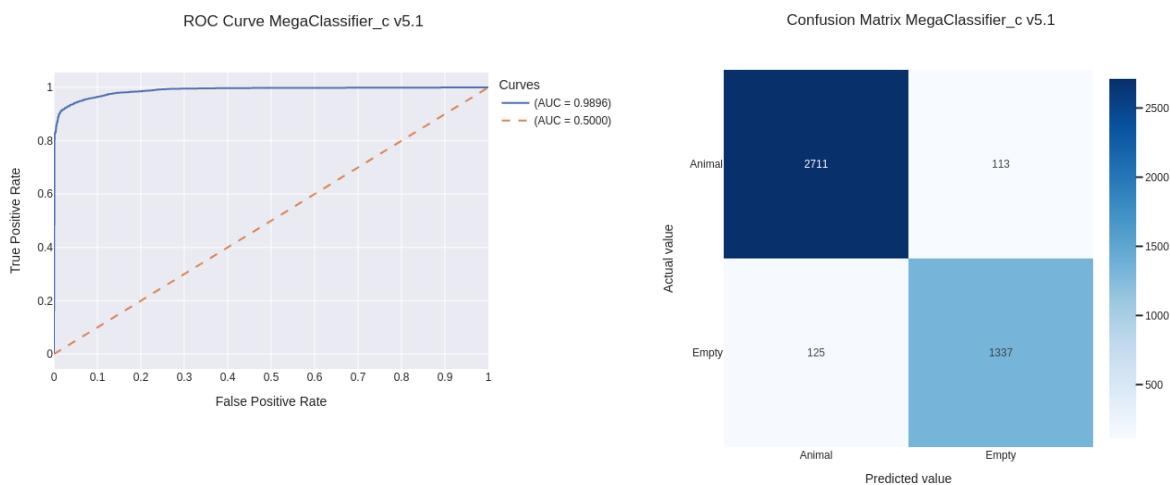


Figura 71: Gráfica Curva ROC y Matriz de confusión de MegaClassifier_c_5.1

En la versión 6.1 de MegaClassifier_c, se observa un excelente equilibrio entre las métricas principales, lo cual la posiciona como una de las variantes más sólidas de toda la línea. Alcanzó una accuracy de 0.9398 y un loss reducido de 0.0181, lo cual indica una mejora en la estabilidad durante el proceso de entrenamiento. Además, su combinación de precision (0.9595), recall (0.9501), specificity (0.9191) y un F1-score elevado (0.9547) sugiere que el modelo no solo es preciso al detectar animales, sino también eficiente en evitar falsos positivos con imágenes vacías. Por último, un AUC de 0.9885 confirma una capacidad predictiva excepcional. Esta versión sienta una buena base para aplicar fine-tuning.

Tabla 25: Resultados versión 6 de MegaClassifier_c

Modelo	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v6.0	0,931638	0,020658	0,952045	0,945047	0,904829	0,948533	0,985447
v6.1	0,939804	0,018169	0,959450	0,950070	0,919128	0,954737	0,988521
v6.2	0,930938	0,020618	0,945346	0,950035	0,894126	0,947685	0,984852
v6.3	0,929305	0,023085	0,952045	0,941751	0,904158	0,946870	0,982541

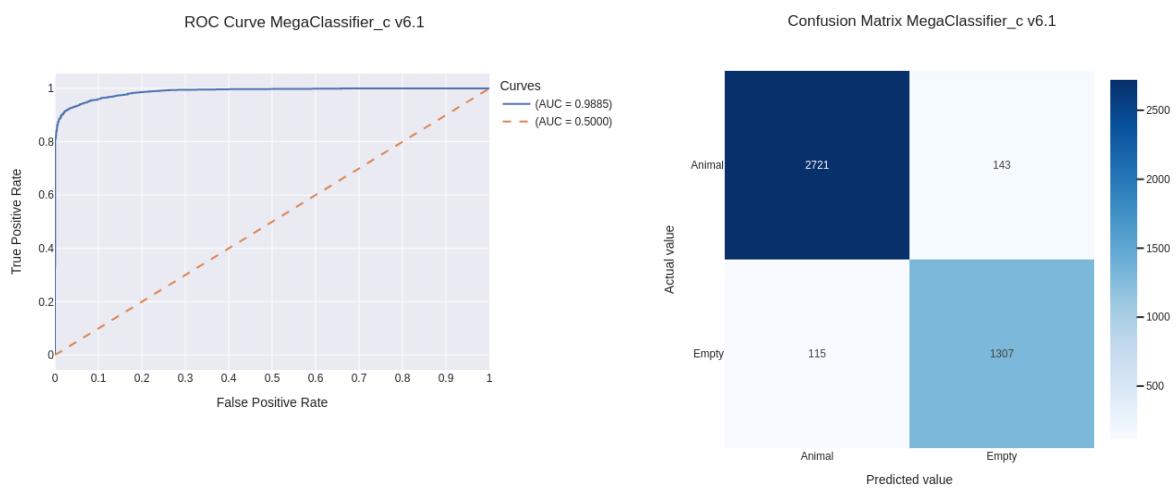


Figura 72: Gráfica Curva ROC y Matriz de confusión de MegaClassifier_c_6.1

La versión v7.3 de MegaClassifier_c, evaluada con un umbral optimizado de 0.5397, representa hasta el momento la versión más precisa y equilibrada del modelo. Con métricas sobresalientes como un F1-Score de 0.9805, recall de 98.64%, specificity de 95.15% y una AUC de 0.9964, este modelo demuestra una excelente capacidad discriminativa y un comportamiento muy estable. Sin embargo, no se considera aún la versión final, ya que el objetivo de la siguiente iteración la versión 8 es reforzar la estabilidad durante el entrenamiento y mejorar la robustez del modelo ante la incorporación de nuevos datos, asegurando así un comportamiento más fiable en entornos reales y escenarios de producción.

Tabla 26: Resultados versión 7 de MegaClassifier_c

Modelo	Umbral	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	-	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v7.0	0,5000	0,957536	0,013577	0,968970	0,966925	0,939058	0,967946	0,993289
v7.0	0,5242	0,959403	0,013577	0,962271	0,976037	0,928188	0,969105	0,993289
v7.1	0,5000	0,961036	0,012531	0,970381	0,970723	0,942109	0,970552	0,994394
v7.1	0,5473	0,962669	0,012531	0,958745	0,984432	0,923228	0,971418	0,994394
v7.2	0,5000	0,963836	0,012781	0,965444	0,979606	0,934272	0,972474	0,994512
v7.2	0,5367	0,962203	0,012781	0,957687	0,984772	0,921466	0,971040	0,994512
v7.3	0,5000	0,973168	0,010435	0,979196	0,980233	0,959394	0,979714	0,996402
v7.3	0,5397	0,974335	0,010435	0,974612	0,986438	0,951482	0,980490	0,996402

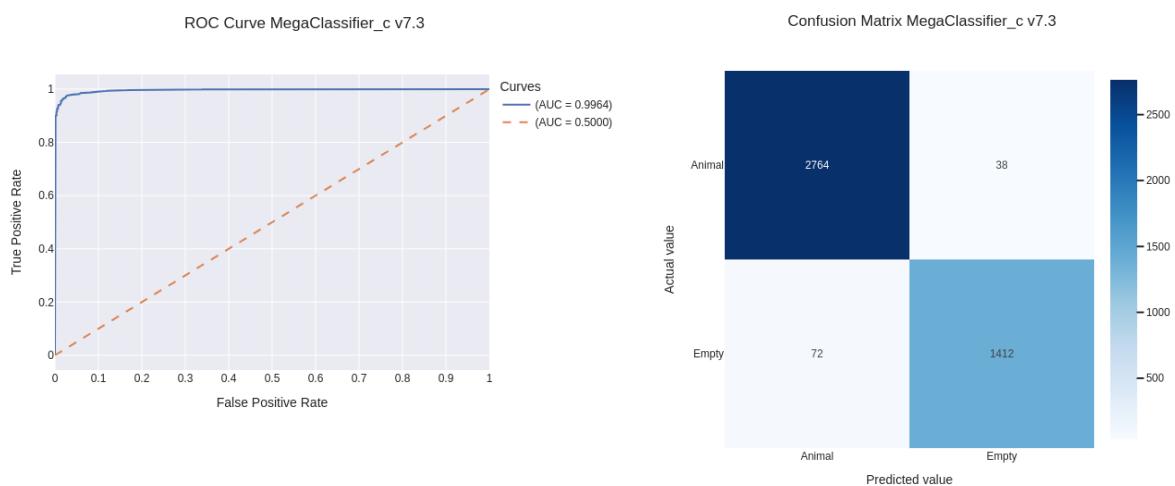


Figura 73: Gráfica Curva ROC y Matriz de confusión de MegaClassifier_c_7.3 con umbral=0.5397

En la comparación final de las versiones del modelo MegaClassifier_c, la versión v8.0 con umbral 0.5592 se perfila como la opción más adecuada para representar al modelo. A pesar de que ambas configuraciones evaluadas de la versión 8.0 ofrecen un rendimiento sobresaliente y prácticamente equivalente en términos de AUC (0.994662) y Accuracy (0.9643), el umbral ajustado a 0.5592 presenta una mejor especificidad (0.923579) con una ligera mejora en la precisión (0.958745), lo cual indica un equilibrio más robusto entre la detección de positivos reales y la minimización de falsos positivos. Esto se refuerza también por su excelente desempeño reflejado en la curva ROC, y una matriz de confusión que muestra un bajo número de errores de clasificación, tanto para la clase “Animal” como “Empty”.

Además, cabe destacar que la versión 8 fue desarrollada específicamente para aportar mayor estabilidad durante el entrenamiento, lo cual puede traducirse en mayor robustez frente a la inclusión de nuevos datos o escenarios más diversos. Esta capacidad de generalización, junto con sus métricas consistentemente altas, la consolidan como la mejor candidata para representar al modelo en su versión más refinada del proyecto.

Tabla 27: Resultados versión 8 de MegaClassifier_c

Modelo	Umbral	Accuracy	Loss	Precision	Recall	Specificity	F1-Score	AUC
MD_0.2	-	0,960803	-	0,959050	0,982722	0,917931	0,970742	-
v8.0	0,5000	0,964069	0,012201	0,971439	0,974187	0,944444	0,972811	0,994662
v8.0	0,5592	0,964302	0,012201	0,958745	0,986933	0,923579	0,972635	0,994662

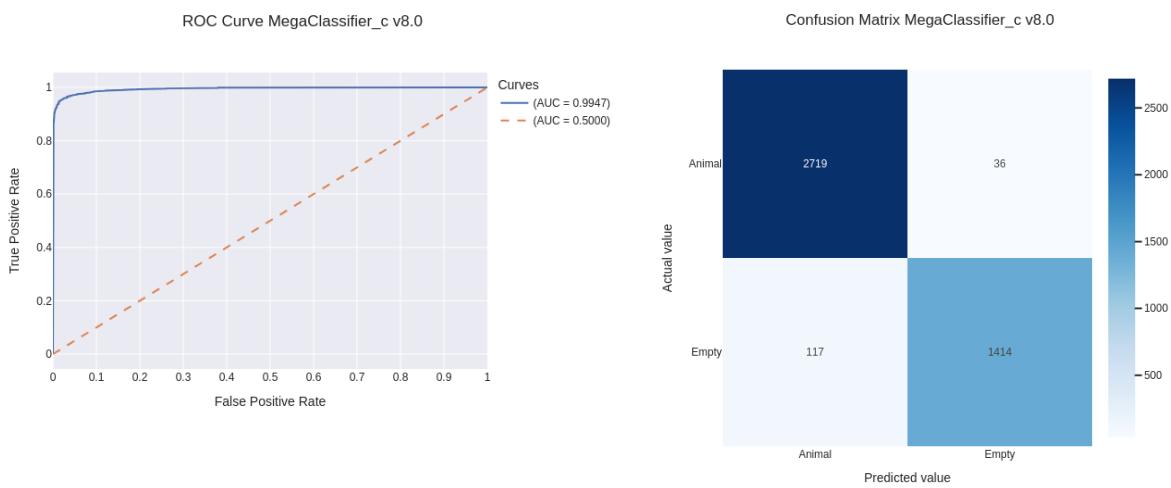


Figura 74: Gráfica Curva ROC y Matriz de confusión de MegaClassifier_c_8.0 con umbral=0.5592

5.2 Análisis y Discusión

Capítulo 6

Conclusiones y Trabajos Futuros

6.1 Conclusiones técnicas

6.2 Trabajos futuros

6.3 Valoración personal

Referencias

- [1] A. Mathison Turing, "On Computable Numbers, with an application to the Entscheidungsproblem," 1936. [Online]. Available: <https://doi.org/10.1112/plms/s2-42.1.230>.
- [2] W. S. McCulloch and W. H. Pitts, "A logical calculus of the ideas immanent in nervous activity," 1943. [Online]. Available: <https://doi.org/10.1007/BF02478259>.
- [3] A. M. Turin, "Computing Machinery and Intelligence," 1950. [Online]. Available: <https://doi.org/10.1093/mind/LIX.236.433>.
- [4] F. Rosenblatt, «The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain,» 1958. [En línea]. Available: <https://psycnet.apa.org/doi/10.1037/h0042519>.
- [5] B. Widrow, "An Adaptive 'ADALINE' Neuron Using Chemical 'Memistors," 1960. [Online]. Available: <https://isl.stanford.edu/~widrow/papers/t1960anadaptive.pdf>.
- [6] M. L. Minsky y S. A. Papert, «Perceptrons: An introduction to computational geometry,» 1969. [En línea]. Available: <https://direct.mit.edu/books/monograph/3132/PerceptronsAn-Introduction-to-Computational>.
- [7] D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning Representations by Back-Propagating Errors," 1986. [Online]. Available: <https://doi.org/10.1038/323533a0>.
- [8] K. Hornik, M. Stinchcombe y H. White, «Multilayer Feedforward Networks Are Universal Approximators,» 1989. [En línea]. Available: [https://doi.org/10.1016/0893-6080\(89\)90020-8](https://doi.org/10.1016/0893-6080(89)90020-8).
- [9] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard y L. D. Jackel, «Backpropagation Applied to Handwritten Zip Code Recognition,» 1989. [En línea]. Available: <https://doi.org/10.1162/neco.1989.1.4.541>.
- [1] S. Hochreiter, «Untersuchungen zu dynamischen neuronalen Netzen,» 1991. [En línea]. Available: <https://www.bioinf.jku.at/publications/older/3804.pdf>.
- [1] G. E. Hinton, S. Osindero and Y.-W. Teh, "A Fast Learning Algorithm for Deep Belief Nets," 2006. [Online]. Available: <https://doi.org/10.1162/neco.2006.18.7.1527>.
- [1] G. E. Hinton, «Training Products of Experts by Minimizing Contrastive Divergence,» 2002. [En línea]. Available: <https://doi.org/10.1162/089976602760128018>.
- [1] A. Krizhevsky, I. Sutskever y G. E. Hinton, «ImageNet Classification with Deep Convolutional Neural Networks,» 2012. [En línea]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

- [1] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke y A. Rabinovich, «Going Deeper with Convolutions,» 2014. [En línea]. Available: <https://doi.org/10.1109/CVPR.2015.7298594>.
- [1] K. He, X. Zhang, S. Ren y J. Sun, «Deep Residual Learning for Image Recognition,» 2015. [En línea]. Available: <https://doi.org/10.1109/CVPR.2016.90>.
- [1] A. Burnes, «NVIDIA DLSS 2.0: Un gran salto en la renderización de IA,» NVIDIA, 23 Marzo 2020.
- [6] [En línea]. Available: <https://www.nvidia.com/es-es/geforce/news/nvidia-dlss-2-0-a-big-leap-in-ai-rendering/>.
- [1] A. Hernandez, Z. Miao, L. Vargas, R. Dodhia y J. M. L. Ferres, «Pytorch-Wildlife: A Collaborative Deep Learning Framework for Conservation,» 2024. [En línea]. Available: <https://arxiv.org/pdf/2405.12930.pdf>.
- [1] C. Seijas, G. Montilla y L. Frassato, «Identificación de especies de roedores usando aprendizaje profundo,» 2019. [En línea]. Available: <https://doi.org/10.13053/cys-23-1-2906>.
- [1] G. Suing-Albito y L. R. B. Guamán, «Aplicación de métodos de deep learning en la identificación y conteo de animales salvajes,» 2022. [En línea]. Available: https://www.researchgate.net/publication/371754966_Aplicacion_de_metodos_de_deep_learning_en_la_identificacion_y_conteo_de_animales_salvajes.
- [2] E. B. HENRÍQUEZ, «Transfer Learning en modelos profundos,» 2019. [En línea]. Available: <https://telefonicatech.com/blog/transfer-learning-en-modelos-profundos>.
- [2] I. P. Borrero y M. E. G. Arias, «DEEP LEARNING Fundamentos , teoría y aplicación,» [En línea].
[1] Available: <https://play.google.com/store/books/details?id=kzsvEAAAQBAJ>.
- [2] X. G. y. Y. Bengio, Understanding the difficulty of training deep feedforward neural networks,
[2] 2008.
- [2] K. He, X. Zhang, S. Ren y J. Sun, «Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification,» 2015. [En línea]. Available: <https://doi.org/10.1109/ICCV.2015.123>.
- [2] H. Robbins y S. Monro, «A Stochastic Approximation Method,» 1951. [En línea]. Available: <https://doi.org/10.1214/aoms/1177729586>.
- [2] B. T. Polyak, «Some methods of speeding up the convergence of iteration methods,» 1964. [En línea]. Available: [https://doi.org/10.1016/0041-5553\(64\)90137-5](https://doi.org/10.1016/0041-5553(64)90137-5).
- [2] D. P. Kingma y J. Ba, «Adam: A Method for Stochastic Optimization,» 2015. [En línea].
[6] Available: <https://doi.org/10.48550/arXiv.1412.6980>.
- [2] K. He, X. Zhang, S. Ren y J. Sun, «Deep Residual Learning for Image Recognition,» 2015. [En línea]. Available: <https://doi.org/10.1109/CVPR.2016.90>.

Anexo Teórico

Apartado 1 Introducción al Deep-Learning

Al abordar la resolución de una tarea mediante el uso de un ordenador, es imprescindible proporcionar una secuencia ordenada de pasos que el sistema debe seguir para lograr su correcta ejecución, lo que se conoce comúnmente como algoritmo. Identificar estos pasos no siempre es fácil, sobre todo cuando nos enfrentamos a tareas muy complejas, como las relacionadas con la visión artificial, la conducción autónoma o el reconocimiento de voz.

Por consiguiente, debemos determinar meticulosamente los pasos que nuestro algoritmo debe seguir. Este proceso implica especificar con antelación las instrucciones que el sistema informático deberá seguir para completar la tarea deseada. Los algoritmos se introducen en el ordenador en forma de programas escritos en algún lenguaje de programación, lo que permite que la máquina ejecute las operaciones necesarias para alcanzar el objetivo deseado.

A partir de este planteamiento, se extrae que para realizar cualquier tarea con un ordenador primero tenemos que dar la secuencia de pasos correcta que, una vez ejecutada, nos lleve a su solución. Sin embargo, cuando nos enfrentamos a problemas más complejos, elaborar un algoritmo capaz de resolverlos se convierte en todo un reto para el desarrollador.

Este es el punto de partida de los algoritmos de *Deep Learning*; sus técnicas tienen como fin que el propio sistema que implementamos encuentre esos pasos para conseguir la resolución del problema.

El *Deep Learning* se inspira en el proceso de aprendizaje humano, un enfoque bioinspirado que se basa en nuestra capacidad innata para adquirir y perfeccionar habilidades a lo largo de la vida. Los seres humanos interiorizamos diversas tareas mediante un proceso de práctica y repetición, como aprender a hablar, caminar o conducir. De manera similar, los algoritmos de *Machine Learning* analizan datos y patrones para mejorar su rendimiento y ofrecer soluciones eficientes a problemas complejos.

Como hemos mencionado, el conjunto de algoritmos aprende a resolver problemas mediante el análisis de datos. Durante el proceso de entrenamiento, se alimenta a un algoritmo con numerosos ejemplos que representan el problema. Así, el algoritmo detecta patrones y relaciones en los datos sin necesidad de recibir instrucciones específicas sobre los datos que debe seguir.

El desarrollo de un sistema de *Machine Learning* consta de varias etapas clave. En primer lugar, el modelo se entrena con datos específicos y se perfecciona mediante la práctica. A continuación, en la fase de validación, el modelo se evalúa con nuevos datos para comprobar su efectividad, de manera similar a como los humanos se someten a exámenes para medir sus conocimientos.

Como se observa en la Figura 75¹⁹, todo proceso completo para obtener un modelo funcional consta de más etapas que las mencionadas anteriormente. A continuación, describimos y explicamos los objetivos de cada una de ellas:

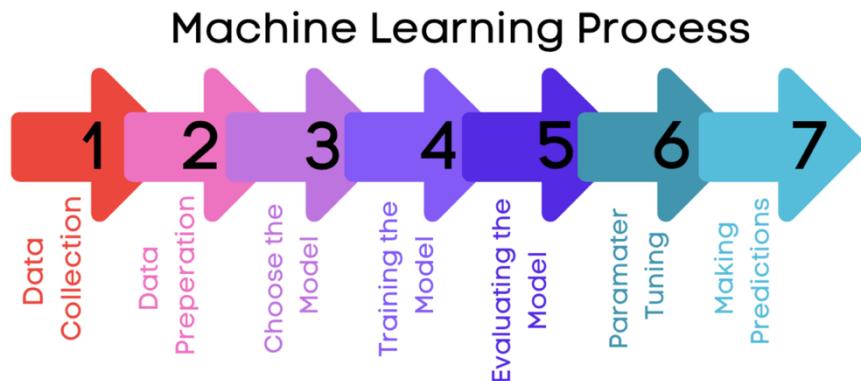


Figura 75: Etapas del proceso de Machine Learning

- **Etapa 1: Recopilación de datos.** Una vez definido el problema que hay que resolver, el primer paso será recopilar datos relevantes. Esto implica obtener datos de diferentes fuentes, como bases de datos, sensores, API o archivos. El objetivo es asegurarse de tener la información suficiente para entrenar un modelo robusto y representativo.
- **Etapa 2: Preparación de los datos.** Con los datos recopilados, se procede a tratarlos y limpiarlos para que estén listos para entrenar el modelo. Esto puede incluir la eliminación de valores faltantes, la corrección de datos erróneos, la normalización o estandarización de los datos y la transformación de variables categóricas en numéricas. El objetivo es sencillo: obtener un conjunto de datos limpio y estructurado que puede utilizarse eficazmente por los algoritmos.
- **Etapa 3: Selección del modelo.** Dentro del *Machine Learning* existe una gran variedad de algoritmos disponibles. La elección de uno u otro puede depender de factores como la naturaleza del problema, el tamaño del conjunto de datos y sus características específicas.
- **Etapa 4: Entrenamiento.** Como mencionamos anteriormente, en esta fase se utiliza el conjunto de datos de entrenamiento para ajustar los parámetros del modelo.
- **Etapa 5: Evaluación.** Una vez que se considera el modelo entrenado, se evalúa su rendimiento utilizando un conjunto de datos de validación o prueba. Para ello, se utilizan métricas de evaluación apropiadas para medir la precisión, la exactitud, etc. El objetivo es determinar qué tan bien se está desempeñando el modelo y si es necesario realizar ajustes.
- **Etapa 6: Ajuste de parámetros.** Llegados al punto de obtener un modelo que haya aprendido correctamente, podemos seguir realizando pruebas para optimizar los hiperparámetros del modelo y mejorar así su rendimiento. Una posible forma de proceder sería fijar un parámetro, por ejemplo, el tamaño de la red; y realizar varios entrenamientos

¹⁹ Fuente: <https://redmond.ai/wp-content/uploads/2023/05/word-image-583-2.png>

con diferentes *learning rate*²⁰ para poder comparar cuál de ellos presenta un mejor comportamiento.

- **Etapa 7: Predicción.** Cuando consideramos que nuestro modelo está preparado, lo evaluaremos con el *dataset* de prueba. De esta forma, podremos estimar cuál será su rendimiento futuro y si es adecuado para resolver la tarea inicialmente planteada.

Estas son las características más habituales en la mayoría de los desarrollos de *Machine Learning*. Dado que este campo posee una gran variedad de algoritmos con diferentes propósitos, pueden existir diferentes clasificaciones. Según la forma que se representa el conocimiento aprendido del modelo, se distinguen tres tipos:

- **Aprendizaje numérico:** Se basa en la obtención de conocimiento representado por valores numéricos, como los pesos en una red neuronal, sin que exista una relación directa con conceptos específicos. Ejemplos de este enfoque son las CNN, utilizadas en este trabajo, y las SVM²¹.
- **Aprendizaje simbólico:** En este caso, se centra en adquirir conocimientos que permiten representar conceptos mediante valores de atributos y reglas lógicas. Un ejemplo de este enfoque, utilizado para construir árboles de decisión es el algoritmo ID3²².
- **Aprendizaje mixto:** Combina elementos del aprendizaje numérico y simbólico, lo que permite la adquisición de conceptos a través de relaciones entre valores y atributos. Un ejemplo de este enfoque es XGBoost²³.

También podemos clasificar según la información proporcionada durante el entrenamiento:

- **Aprendizaje supervisado:** Este tipo se caracteriza por tener salidas conocidas para cada entrada, lo que permite ajustar los valores internos del algoritmo y aproximarse a los resultados correctos.
- **Aprendizaje no supervisado:** A diferencia del anterior, este tipo de aprendizaje opera sin salidas conocidas, por lo que se enfoca en la identificación de patrones en los datos de entrada.
- **Aprendizaje por refuerzo:** Tampoco se conocen las salidas, pero la diferencia estriba en la utilización de un sistema de recompensas y castigos para guiar al modelo hacia la optimización de sus resultados. Esto hace que vaya aprendiendo en función de las acciones tomadas en distintos estados.

Podríamos realizar una clasificación en función del tipo de problema que se pretende solucionar. Algunos de los problemas más comunes son:

- **Regresión:** Su objetivo es modelar la relación entre los atributos de entrada y la salida del sistema de naturaleza numérica o cuantitativa.

²⁰ Hiperparámetro clave en el entrenamiento de CNN que determina el tamaño de los ajustes en los pesos de la red.

²¹ Support Vector Machine, algoritmo de clasificación que encuentra el hiperplano óptimo para separar distintas clases en un espacio de alta dimensión.

²² Iterative Dichotomiser 3, basado en árboles de decisión que construye el modelo seleccionando el atributo más informativo en cada división.

²³ Extreme Gradient Boosting, algoritmo basado en árboles de decisión optimizados mediante boosting, diseñado para ser eficiente y preciso en tareas de clasificación y regresión.

- **Clasificación:** Su objetivo es modelar la relación entrada y la salida cualitativa o categórica del sistema.

A continuación, hablaremos sobre algunos de los algoritmos utilizados en el *Machine Learning* para resolver los problemas mencionados en la Figura 76²⁴.

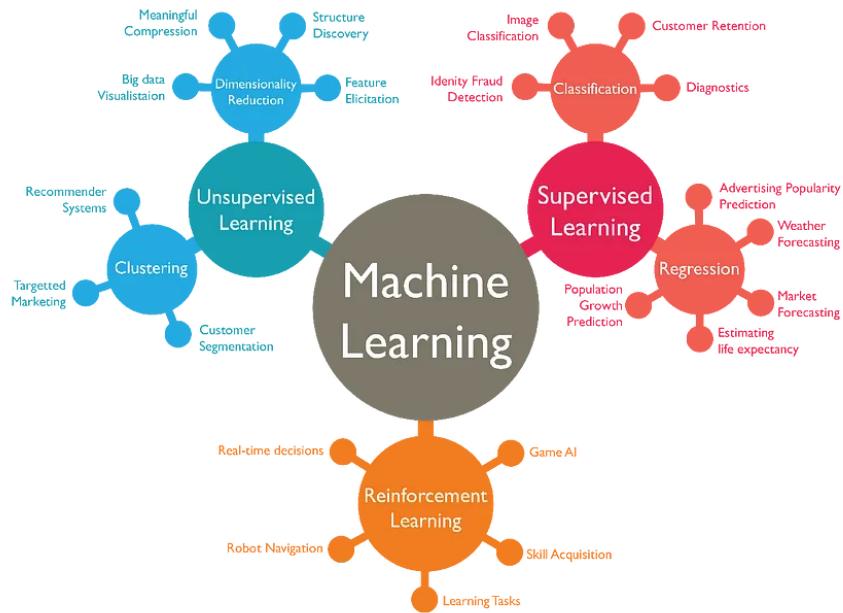


Figura 76: Esquema de aplicaciones del Machine Learning

- **Algoritmos de regresión:** son esenciales para predecir valores y hacer uso de la métrica del error para minimizarlo en cada iteración. Algunos ejemplos son: Regresión lineal, descenso por gradiente, etc.
- **Algoritmos de reducción de dimensión:** Se aplican en aprendizaje no supervisado y su peculiaridad es que permiten reducir las características de un modelo complejo para facilitar su interpretación. Algunos ejemplos de este tipo de algoritmos son *Principal Component Analysis* (PCA), *Linear Discriminant Analysis* (LDA), etc.
- **Algoritmos Bayesianos:** Se utilizan en problemas de clasificación y regresión y se basan en teoremas de probabilidad, especialmente en el teorema de Bayes. Algunos ejemplos son: *Bayesian Network*, *Naive Bayes*, etc.
- **Máquina vector soporte:** Se ha mencionado en apartados anteriores. Conocida originalmente como Support Vector Machine, crea modelos que pueden mapear datos a un espacio de mayor dimensión, lo que permite encontrar un hiperplano que separe las clases y maximice el margen entre ellas.
- **Algoritmos de Clustering:** Se utilizan principalmente para agrupar datos de los que desconocemos sus características en común. Por este motivo, se utilizan en el aprendizaje no supervisado. Un ejemplo muy conocido es K-nearest Neighbors (KNN).
- **Algoritmos de Redes neuronales:** Se basan en el funcionamiento del cerebro humano. Suelen utilizarse para la clasificación y la regresión, aunque tienen un gran potencial para resolver problemas variados y son la base del *Deep Learning*. Los ejemplos más habituales son: Perceptrón, perceptrón multicapa, *backpropagation*, etc.

²⁴ Fuente: [https://media.linkedin.com/dms/image/D5612AQHT5uzEz5mZ8g/article-cover_image-shrink_600_2000/0/...](https://media.linkedin.com/dms/image/D5612AQHT5uzEz5mZ8g/article-cover_image-shrink_600_2000/0/)

- **Algoritmos de Deep Learning:** Han surgido como la evolución de las redes neuronales anteriormente mencionadas. Su gran crecimiento se debe al abaratamiento de la tecnología actual, a la reducción de los costes computacionales y a la gran cantidad de datos de los que disponemos hoy en día. Algunos ejemplos de este tipo de algoritmos son: Convolutional Neural Network (CNN), Hierarchical Convolutional Deep Maxout Networks (HCDMN), Long Short Term Memory Neural Networks (LSTMNN), etc.

Un aspecto en la metodología de la gran mayoría de los algoritmos de Machine Learning es la correcta selección de los datos que tomaremos como representativos para la entrada. La correcta formación del modelo está estrechamente vinculada a la forma en que se representan los datos previamente.

Como vimos en la Figura 75, en la primera etapa del proceso, que consiste en la obtención de datos, es importante especificar las características que mejor definen cada una de las instancias del problema que se va a tratar. La correcta selección de estas características favorecerá que el aprendizaje alcance un nivel óptimo de calidad en los resultados. En caso contrario, si no se definen correctamente estas características, puede ser necesario utilizar un mayor número de ellas y, por tanto, un modelo más complejo, que en el peor de los casos puede impedir que este alcance un resultado aceptable.

Una de las maneras más generales de seleccionar estas características es utilizar aquellas que estadísticamente mejor definen el problema. Otra metodología consiste en consultar a expertos en la materia, ya que gracias a su experiencia pueden proporcionar información valiosa y relevante para realizar una selección correcta.

Llegados a este punto, nos damos cuenta de una de las grandes diferencias entre los algoritmos “convencionales” de *Machine Learning* y las nuevas técnicas de *Deep Learning*. En los primeros, hay que indicar previamente las características más relevantes, mientras que en las técnicas de *Deep Learning* es el propio algoritmo quien las encuentra por sí mismo durante la fase de entrenamiento. (ver Figura 77)²⁵.

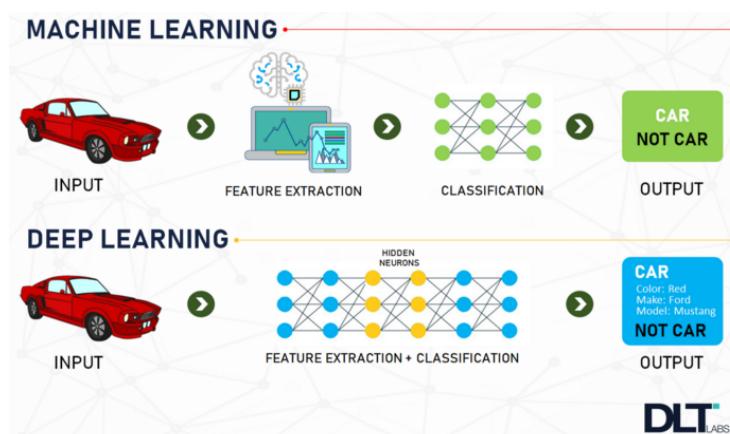


Figura 77: Comparativa Machine Learning VS Deep Learning

²⁵ Fuente: [https://media.licdn.com/dms/image/D4D12AQH1_eSvKLwTiA/article-cover_image-shrink_600_2000/0/...](https://media.licdn.com/dms/image/D4D12AQH1_eSvKLwTiA/article-cover_image-shrink_600_2000/0/)

Apartado 2

Redes Neuronales

Las redes neuronales artificiales han emergido como una de las técnicas más revolucionarias dentro del campo del aprendizaje profundo. Inspiradas en la estructura y el funcionamiento del cerebro humano, estas redes están compuestas por unidades de procesamiento llamadas “neuronas artificiales”, organizadas en capas interconectadas que permiten modelar relaciones complejas en los datos. Su capacidad de aprendizaje ha sido fundamental en múltiples aplicaciones, desde el reconocimiento de imágenes hasta el procesamiento del lenguaje natural, impulsando avances significativos en inteligencia artificial. En este capítulo se presentan los fundamentos de las redes neuronales, abordando su evolución, estructura y funcionamiento.

2.1 Fundamentos

El desarrollo de las redes neuronales artificiales se basa en principios matemáticos y computacionales que permiten crear modelos capaces de aprender patrones a partir de los datos. Para comprender su funcionamiento, es fundamental conocer los conceptos básicos en los que se basan, que expondremos a continuación.

2.1.1 Perceptrón

Para conocer los orígenes del perceptrón, es preciso retrotraerse al año 1943, un período que precede significativamente la conceptualización del término *inteligencia artificial*. Fue en dicho año cuando Warren McCulloch y Walter Pitts divulgaron un modelo matemático que procuraba representar de manera simplificada el funcionamiento de una neurona biológica [2]. Este modelo se fundamenta en una estructura lógica que procesa la información mediante entradas y salidas binarias, sentando así los cimientos para el desarrollo de las redes neuronales artificiales.

La neurona biológica, en su configuración más básica, se constituye a partir de tres componentes principales: el soma o cuerpo celular, que alberga el núcleo y regula la actividad neuronal derivada de los impulsos nerviosos recibidos de otras neuronas a través de los canales de entrada, denominadas dentrinas. Estas extensiones ramificadas, juegan un papel crucial en la transmisión de los impulsos nerviosos. Cuando el impulso recibido supera un umbral determinado, la neurona se activa, generando un impulso a través del canal de salida, el axón. Éste último se conecta a las dentrinas de otra neurona mediante lo que se conoce como conexión sináptica, permitiendo la transferencia de información entre las células neuronales. La comunicación entre neuronas se lleva a cabo mediante señales eléctricas dentro de la célula y señales químicas en las sinapsis, lo que permite el procesamiento de información en el cerebro y el sistema nervioso. (ver Figura 78)²⁶.

²⁶ Fuente: <https://ml4a.github.io/images/neuron-anatomy.jpg>

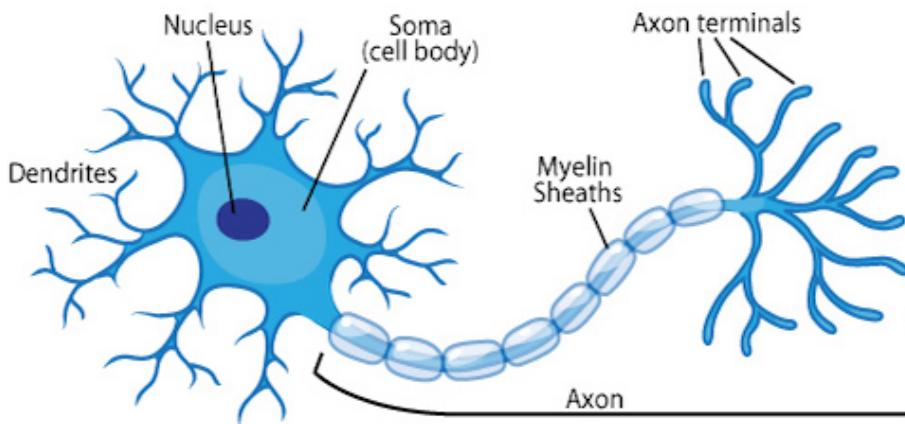


Figura 78: Anatomía de una neurona biológica

En este sentido, McCulloch y Pitts desarrollaron su modelo matemático, el cual recibe dos tipos de entradas binarias: excitadoras (x_i) e inhibidoras (x_j^*). Dichas entradas son procesadas mediante la suma de estas. Si dicha suma supera un umbral prefijado, la neurona se activará y su salida valdrá uno. En caso de no superar el umbral, su salida valdrá cero. Es importante destacar que, en presencia de alguna entrada inhibidora activa, la salida tomará el valor cero, independientemente de si la agregación excede el umbral.

Este primer modelo se caracteriza por su limitación a las funciones booleanas, determinada por su naturaleza intrínseca. Adicionalmente, carece de un mecanismo para el autoajuste del valor del umbral, el cual debe ser definido de antemano. Asimismo, es posible que no sea deseable que todas las entradas sean iguales, sino que se prefiera otorgar mayor importancia a algunas entradas frente a otras.

En 1958, Frank Rosenblatt desarrolló el perceptrón con el propósito de resolver las limitaciones observadas en la neurona de McCulloch-Pitts [4]. Este modelo posee la capacidad de procesar cualquier entrada real, lo que resulta en la eliminación de las entradas inhibidoras. En consecuencia, una entrada negativa generaría un comportamiento equivalente. Este modelo opera recibiendo múltiples entradas, cada una con un peso asociado, que se suman y procesan mediante una función de activación, la cual, a diferencia del anterior modelo, ya no se encuentra en la propia neurona sino como elemento posterior. En contraste con el modelo de McCulloch-Pitts, el perceptrón puede modificar sus pesos a través de un algoritmo de aprendizaje supervisado. (ver Figura 79)²⁷.

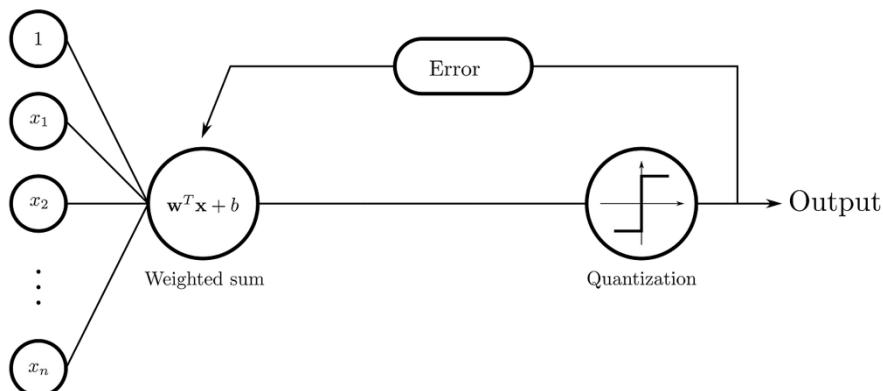


Figura 79: Perceptrón y su proceso de aprendizaje

²⁷ Fuente: https://miro.medium.com/v2/resize:fit:1400/1*BM8HuBXy9XYLvSGIBZwL0g.png

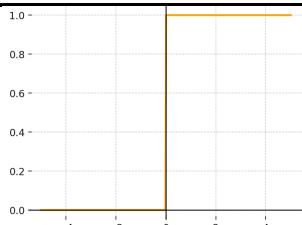
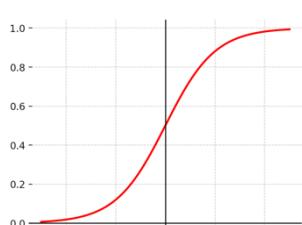
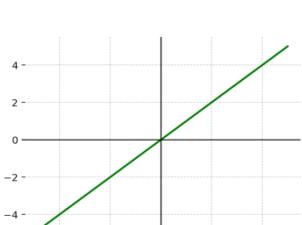
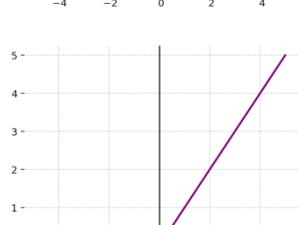
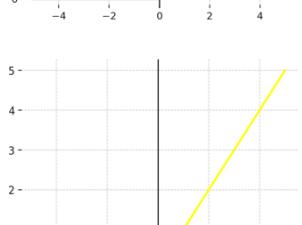
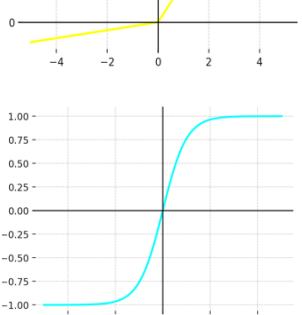
En 1960, Bernard Widrow y Marcian Hoff introdujeron el modelo ADALINE (Adaptive Linear Neuron) [5], que supuso una mejora significativa sobre el perceptrón al emplear una función de activación lineal en lugar de una función escalón. Este cambio permitió la aplicación de técnicas de optimización para ajustar los pesos de manera más eficiente. A diferencia del perceptrón, que ajusta los pesos únicamente en función del resultado final de la clasificación, el modelo ADALINE los ajusta considerando la diferencia entre la salida deseada y la salida real antes de la aplicación de la función de activación. Estas mejoras contribuyeron significativamente al desarrollo de la versión actual del perceptrón.

2.1.2 Funciones de activación

Como se ha expuesto en la sección precedente, el desarrollo del perceptrón implica que la función de activación deja de ser parte integrante del mismo, pasando a ser un elemento posterior que actúa sobre su salida. Este cambio permite la modificación de la salida del perceptrón para que esta tenga sentido en el contexto del problema a resolver.

Como se ha mencionado anteriormente, se ha expuesto el modo en el que las primeras aplicaciones utilizaban la función booleana. Posteriormente, la neurona de McCulloch-Pitts hacía uso de la función escalonada. Este paradigma cambió con la llegada del perceptrón, que trajo consigo el uso de valores reales. Aunque teóricamente podría utilizarse cualquier función, se expondrán las más populares. (ver Tabla 28)

Tabla 28: Principales funciones de activación

Nombre	Gráfica	Ecuación
Escalonada		$f(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$
Sigmoide		$f(x) = \frac{1}{1 + e^{-x}}$
Identidad		$f(x) = x$
ReLU		$f(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$
PReLU		$f(x) = \begin{cases} ax, & x < 0 \\ x, & x \geq 0 \end{cases}$
TanH		$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

2.1.3 Tipo de capas

Para proceder a la descripción de la organización y las conexiones de las neuronas que constituyen una red neuronal, es imperativo iniciar la investigación con el primer componente, que es la capa. Se denomina capa a la agrupación de neuronas de una red que reciben las mismas entradas, como se muestra en la Figura 80²⁸.

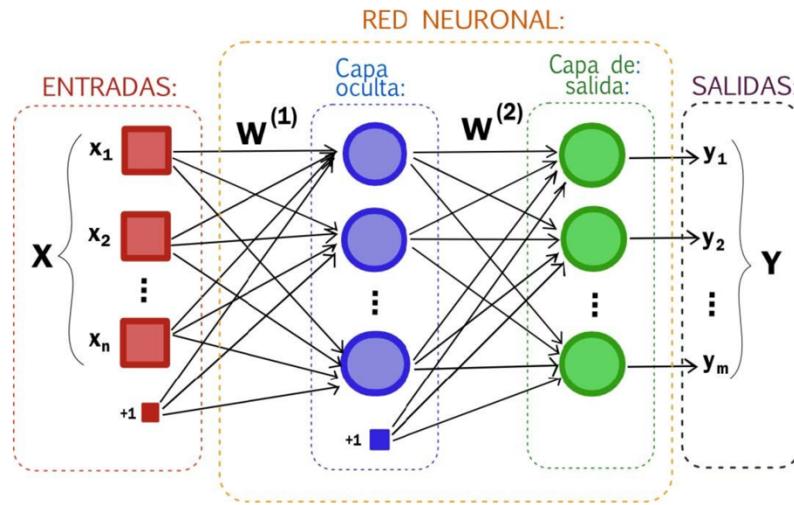


Figura 80: Estructura básica de una red neuronal multicapa

Como se ha expuesto en investigaciones previas, se ha determinado que la ecuación de un perceptrón en formato vectorial es:

$$y = W^T x$$

Ecuación 1: Ecuación del perceptrón en formato vectorial

En términos generales, las n entradas se organizan en un vector $x = [x_0, x_1, \dots, x_n]^T$, donde el primer componen $x_0 = 1$. Por otro lado, las salidas de las m neuronas de la capa se agrupan en el vector salida de la capa $y = [y_0, y_1, \dots, y_m]^T$. En lo que respecta a los pesos de cada neurona, $w^i = [w_0^i, w_1^i, \dots, w_n^i]^T$, son agrupados en la matriz de pesos $W \in \mathbb{R}^{(n+1) \times m}$, donde $(n + 1)$ representa el número de entradas más el sesgo. Como podemos observar en la Ecuación 1, se ha omitido la inclusión de la función de activación, dado que por lo general las neuronas de una misma capa cuentan con las mismas conexiones y función de activación y que ésta se aplica al valor de cada salida por separado, podemos reescribir la ecuación de forma que la función se aplica a cada componente del vector:

$$y = f(W^T x)$$

Ecuación 2: Ecuación del perceptrón aplicando función de activación a cada componente del vector

Una vez que se han introducido las ecuaciones y sus componentes, se puede proceder a la introducción de los tres tipos de capas.

- **Capa de entrada:** En el ámbito de la inteligencia artificial, es una práctica común representar los datos de entrada como una capa adicional de la red neuronal, lo que se

²⁸ Fuente: [https://media.linkedin.com/dms/image/D4E12AQHGrL3AHLPo2Q/article-cover_image-shrink_720_1280/0/...](https://media.linkedin.com/dms/image/D4E12AQHGrL3AHLPo2Q/article-cover_image-shrink_720_1280/0/)

conoce como *capa 0*, donde los elementos de la capa son los componentes del vector $x = [x_0, x_1, \dots, x_n]^T$. Por otro lado, la salida de esta capa y^0 , se corresponde directamente con el vector de entrada x , haciendo que la utilización de esta capa simplifique la notación de las operaciones.

- **Capa oculta y Capa de salida:** Capas que se ubican posteriores a la capa de entrada. Las neuronas que conforman estas capas han sido expuestas en el apartado 2.1.1 Perceptrón, mediante el estudio de los perceptrones. Estos componentes poseen pesos que desempeñan un papel crucial en la transformación de los datos de entrada, junto con la función de activación que determina su comportamiento y respuesta. La principal diferencia entre la capa oculta y la capa de salida radica en la ubicación dentro de la red neuronal. Específicamente, la capa de salida constituye la última capa de la red, mientras que la capa oculta se ubica entre la capa de entrada y la capa de salida.

2.1.4 Aplicación a problemas de clasificación

Las redes neuronales exhiben la ventaja de poseer múltiples neuronas en la capa de salida, lo que facilita la resolución de problemas que resultan inviables de ser abordados por un solo perceptrón, como es el caso de la clasificación en múltiples categorías.

A modo ilustrativo, en el contexto del reconocimiento de dígitos en imágenes, se podría optar por la implementación de diez neuronas en la capa de salida, donde cada una estaría encargada de identificar un número específico dentro de la imagen ingresada. Sin embargo, no se puede asegurar que solo una neurona se active o que ninguna lo haga.

Para gestionar esta situación, se implementa la función de activación identidad en la capa de salida y se aplica la función *softmax* al vector de salida. Ésta última es una generalización de la función sigmoide, que transforma el vector de salida en una distribución de probabilidad.

$$S(y_i) = \frac{e^{y_i}}{\sum_{j=1}^n e^{y_j}}$$

Ecuación 3: Función softmax

El empleo de esta función permite interpretar la salida de cada neurona como una probabilidad. No obstante, esto no implica que la red haya llevado a cabo una clasificación definitiva para la entrada. Para obtener una categoría concreta, es preciso convertir el vector de probabilidades en una representación que refleje la clase asignada. Por lo general, esto se realiza seleccionando la clase correspondiente a la neurona con la mayor probabilidad. Un método común para lograrlo es la codificación *one-hot*, que consiste en asignar el valor 1 a la componente del vector con la probabilidad más alta (en caso de empate, se elige una de ellas) y 0 a las demás. (ver Figura 81) [21].

$$y = \begin{bmatrix} 1.2 \\ 0.9 \\ 0.4 \end{bmatrix} \xrightarrow{\text{softmax}} \begin{bmatrix} 0.46 \\ 0.34 \\ 0.20 \end{bmatrix} \xrightarrow{\text{one-hot}} \begin{bmatrix} 1.0 \\ 0.0 \\ 0.0 \end{bmatrix}$$

Figura 81: Transformación softmax y codificación one-hot

Al examinar la Figura 81, se evidencia que el vector de salida, al ser sometido a la aplicación de la transformación *softmax*, da lugar a una representación de distribución de probabilidad. En consecuencia, al codificar este vector de probabilidades mediante el método de *one-hot*, se obtiene la clasificación de la entrada, entendida como la clase correspondiente al valor 1 asignado a dicho vector.

2.1.5 Aplicación a imágenes

En el campo de la inteligencia artificial, las redes neuronales han demostrado su capacidad para abordar problemas de gran complejidad, que trascienden la mera clasificación multiclase. Al representar una imagen como un vector de píxeles, esta puede ser procesada por una red neuronal, pudiendo incluso generar una imagen como salida. La clave del proceso radica en la utilización de una cantidad adecuada de neuronas en la capa de salida, proporcional al número de píxeles que se desean obtener.

Es crucial reconocer que una de las restricciones fundamentales de las redes neuronales radica en la necesidad de que tanto la entrada como la salida estén representadas como vectores. En consecuencia, para procesar datos con estructuras espaciales, como las imágenes, es imperativo transformarlos en una representación vectorial. Este procedimiento se denomina *flattening* o aplastamiento.

Este proceso implica la transformación de una estructura de datos multidimensional, como una imagen, en un vector unidimensional para su procesamiento por parte de una red neuronal. En lo que respecta a su aplicación en imágenes, que comúnmente se representan como matrices de píxeles con una o más dimensiones (como en imágenes en escala de grises o en color con canales RGB), el proceso de *flattening* reorganiza estos datos en una única fila de valores (ver Figura 82)²⁹. Este paso resulta de vital importancia en el contexto de las redes neuronales con capas *fully connected*, ya que estas requieren que la información de entrada esté en formato vectorial. No obstante, en arquitecturas especializadas, tales como las redes neuronales convolucionales (CNNs), se busca evitar el flattening en etapas intermedias con el fin de preservar la estructura espacial de los datos y mejorar el rendimiento en tareas como el reconocimiento de imágenes.

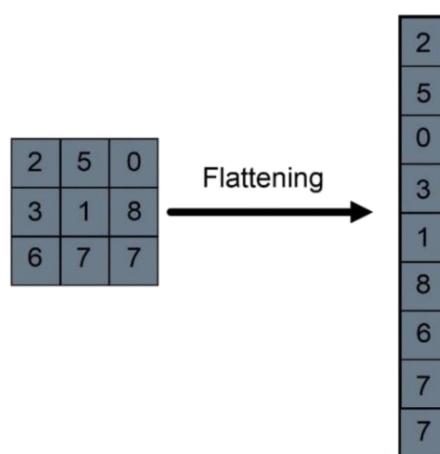


Figura 82: Ejemplo de flattening

²⁹ Fuente: [https://www.researchgate.net/publication/359174861/figure/fig5/...](https://www.researchgate.net/publication/359174861/figure/fig5/)

2.2 Desarrollo de redes neuronales

Una vez examinados los componentes, aplicaciones y limitaciones de las redes neuronales, el siguiente paso a considerar es el desarrollo de estas. Para construir una red neuronal, es fundamental definir primero su arquitectura, lo cual generalmente se realiza mediante un proceso de prueba y error. La estrategia más común es basarse en arquitecturas populares que han demostrado buenos resultados en diversos problemas. Otro aspecto esencial en la creación y aplicación de redes neuronales es el manejo de los datos. El análisis exhaustivo de los problemas abordados con estas redes revela su esencia como un proceso de aproximación de funciones, donde la función objetivo se modela a partir de un conjunto de ejemplos representativos.

En el ámbito de la investigación en machine learning, se ha observado una tendencia a distinguir entre la fase de obtención del modelo y la fase de uso de este. Este enfoque metodológico se ha convertido en una práctica común en el desarrollo de modelos de machine learning, con el objetivo de facilitar la comprensión y la aplicación de estos sistemas en diversos contextos. Como se refleja en la Figura 83³⁰.

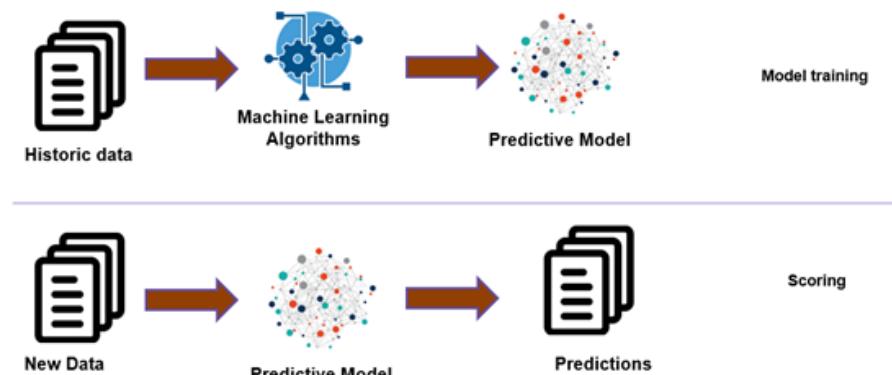


Figura 83: Etapa de entrenamiento e inferencia de un modelo de machine learning

- **Etapa de entrenamiento:** La etapa de entrenamiento constituye un componente esencial en el desarrollo de una red neuronal, dado que en esta fase la red aún no ha establecido sus pesos. Para ajustarlos, se emplean los datos disponibles con el propósito de que la red alcance la aproximación más precisa posible a la función deseada. Este ajuste de pesos puede llevarse a cabo de diversas maneras: manualmente, de manera aleatoria o de forma automática mediante un algoritmo de entrenamiento. En el contexto de las redes neuronales, esta fase se denomina etapa de aprendizaje, durante la cual la red optimiza sus parámetros para mejorar su rendimiento en la tarea específica. El resultado de este proceso es una red neuronal con un conjunto de pesos definidos que le permiten realizar predicciones o clasificaciones con mayor precisión.
- **Etapa de inferencia:** La etapa de inferencia en una red neuronal se produce cuando los pesos dejan de ser actualizados y la red es utilizada exclusivamente para realizar predicciones. Esta fase resulta esencial para evaluar la efectividad del entrenamiento y determinar si los resultados obtenidos son satisfactorios o si es necesario continuar ajustando la red. Una de las principales ventajas de las redes neuronales radica en su capacidad para generar salidas a partir de cualquier dato de entrada, no únicamente aquellos con los que fueron entrenadas. Esta capacidad permite que la red procese información nueva y realice predicciones sobre

³⁰ Fuente: <https://blogger.googleusercontent.com/img/b/R29vZ2xl/...>

datos nunca vistos previamente. En esta etapa, la red no se ve sometida a un proceso de aprendizaje o modificación de sus parámetros, sino que simplemente aplica el conocimiento adquirido durante el entrenamiento para inferir resultados. El único producto de esta fase es la salida generada por la red a partir de los datos de entrada proporcionados.

2.2.1 Etapa de entrenamiento

Para asegurar el correcto funcionamiento de una red neuronal, resulta imperativo disponer de procedimientos que posibiliten la evaluación de su rendimiento. Las métricas de evaluación satisfacen esta necesidad, puesto que no solo facilitan el monitoreo del entrenamiento, sino que también brindan una medida objetiva para la comparación de redes con diferentes pesos o arquitecturas.

Para calcular estas métricas, es imperativo conocer la clase o el valor real de cada dato de entrada, ya que se requiere comparar la salida esperada con la salida generada por la red. En consecuencia, las métricas de evaluación no pueden aplicarse durante la fase de inferencia, ya que en esta etapa la red procesa datos sin una referencia conocida para validar sus respuestas. En contraste, estas métricas pueden ser utilizadas en los conjuntos de datos de entrenamiento, validación y prueba, donde se cuenta con etiquetas o valores de referencia que permiten medir con precisión el rendimiento de la red.

Como se ha mencionado anteriormente, para entrenar y evaluar de manera efectiva una red neuronal, es común dividir los datos en tres conjuntos principales, cada uno con un propósito específico:

- **Conjunto de entrenamiento:** este conjunto de datos se utiliza para ajustar los pesos de las neuronas durante la etapa de entrenamiento. La red aprende a reconocer patrones a partir de estos ejemplos, optimizando su desempeño en la tarea asignada.
- **Conjunto de validación:** evalúa el comportamiento de la red utilizando datos que no han sido observados anteriormente. A diferencia del conjunto de entrenamiento, estos datos no se utilizan para ajustar los pesos de la red, sino para medir su rendimiento durante el proceso de entrenamiento y para tomar decisiones sobre ajustes en la arquitectura o hiperparámetros.
- **Conjunto de prueba:** se utiliza para evaluar el rendimiento final del modelo una vez completado el entrenamiento. A diferencia del conjunto de validación, este conjunto no se emplea en ninguna fase del ajuste del modelo, sino que se usa únicamente al final para estimar el comportamiento que tendrá la red durante la fase de inferencia.

Durante el proceso de entrenamiento, la red es sometida a múltiples evaluaciones utilizando el conjunto de validación, conforme a los parámetros establecidos, con el propósito de monitorear su rendimiento. En caso de que se evidencie una estabilización o incluso un deterioro en las métricas de rendimiento, se recomienda interrumpir el entrenamiento y considerar la implementación de ajustes en la estrategia o la revisión del código en busca de errores. Además, la evaluación en el conjunto de entrenamiento sirve como referencia para compararla con la de validación, ya que se espera que ambas métricas sean similares. Si la diferencia entre ellas es significativa, puede indicar problemas como sobreajuste o subajuste.

Las métricas de evaluación no son universales y exhiben variaciones en función del tipo de problema que se esté abordando. En el contexto de los problemas de clasificación, las métricas predominantes se fundamentan en la interpretación del vector de salida de la red, donde la

asignación de la clase se determina predominantemente según la posición del valor máximo en dicho vector. No obstante, este enfoque presenta una limitación fundamental: existe la posibilidad de que una entrada sea clasificada incorrectamente en una categoría, incluso si la probabilidad asignada es demasiado baja.

Para abordar esta problemática, se implementa un umbral, como 0.5, que delimita el rango de probabilidad. En caso de que el valor de probabilidad más alto en la salida supere este umbral, se asigna la clase correspondiente. Sin embargo, es importante destacar que este procedimiento no garantiza una clasificación exacta, ya que el valor máximo puede situarse por debajo del umbral o, incluso, no coincidir con la clase esperada, en caso de que supere el umbral. En consecuencia, en múltiples circunstancias se implementan métricas complementarias, tales como la precisión, la exhaustividad o la métrica F1, con el propósito de obtener una valoración más precisa del rendimiento del modelo.

La evaluación del rendimiento de una red neuronal se fundamenta en la comparación entre las clases predichas por la red y las clases esperadas. Mediante esta comparación, es posible obtener cuatro métricas fundamentales:

- **True Positives (TP):** Se produce cuando la red clasifica correctamente una instancia como positiva, es decir, cuando la clase asignada por la red coincide con la clase positiva esperada.
- **True Negatives (TN):** Se produce cuando la red identifica correctamente una instancia como negativa, es decir, la clase asignada como negativa coincide con la clase negativa esperada.
- **False Positives (FP):** Suceden cuando la red clasifica erróneamente una instancia como positiva cuando en realidad pertenece a la clase negativa. Este error se conoce como *falso positivo* o *error tipo I*.
- **False Negatives (FN):** Se producen cuando la red clasifica incorrectamente una instancia como negativa cuando en realidad pertenece a la clase positiva. Este error se conoce como *falso negativo* o *error de tipo II*.

Estas métricas se organizan en una estructura denominada matriz de confusión, la cual funciona como un contador que permite visualizar el rendimiento de la red en la clasificación de los datos. En esta matriz, las filas representan las clases reales (esperadas) de las instancias de entrada, mientras que las columnas corresponden a las clases predichas por el modelo. En este contexto, cada predicción realizada por la red se traduce en un incremento del valor de la celda correspondiente, según la coincidencia o el error entre la clase esperada y la clase predicha. Es pertinente señalar que la construcción de la matriz de confusión puede abarcar todas las clases contenidas en el modelo. (ver Figura 84)³¹.

		Walleye	Largemouth Bass	Bluegill	Rainbow Trout
		Walleye	Largemouth Bass	Bluegill	Rainbow Trout
Actual value	Walleye	TP			
	Largemouth Bass		TP		
Bluegill				TP	
					TP
		Walleye	Largemouth Bass	Bluegill	Rainbow Trout

Figura 84: Ejemplo de matriz de confusión para 4 clases

³¹ Fuente: [https://www.ibm.com/content/dam/connectedassets-adobe-cms/...](https://www.ibm.com/content/dam/connectedassets-adobe-cms/)

En problemas de clasificación multiclas, si la salida de la red se encuentra en formato *one-hot encoding* o si se emplea un umbral de probabilidad, es posible generar una matriz de confusión para cada clase. En este caso, se considera la clase de interés como la clase positiva y el resto de las clases como negativas. Este enfoque, denominado método de matrices de confusión individuales por clase, es el más prevalente y posibilita la evaluación del rendimiento del modelo en cada categoría de manera independiente. (ver Figura 85)³².

		Positive	TP	FN
	Actual value			
Negative			FP	TN
		Positive		Negative

Figura 85: Matriz de confusión considerando únicamente las clases positivas y negativas

El análisis de la matriz de confusión proporciona información relevante sobre los errores de clasificación, lo que permite optimizar el modelo para mejorar su precisión y capacidad de generalización.

Una vez generada la matriz de confusión para cada categoría, es posible estimar diversas métricas de evaluación que permiten la valoración del rendimiento del modelo de clasificación. Las métricas más frecuentemente empleadas incluyen:

- **Precision:** constituye una métrica que permite evaluar la proporción de predicciones correctas entre todas las predicciones positivas emitidas por el modelo. Este cálculo se determina mediante la siguiente fórmula:

$$Precision = \frac{TP}{TP + FP}$$

Un alto nivel de precisión indica que el modelo presenta un número reducido de falsos positivos.

- **Recall:** También conocido como *Sensitivity*, se refiere a la aptitud del modelo para identificar de manera precisa las instancias positivas. Su cálculo se determina mediante la siguiente fórmula:

$$Recall = \frac{TP}{TP + FN}$$

Un alto índice de recall indica que el modelo detecta la mayoría de los casos positivos, aunque puede resultar en una mayor cantidad de falsos positivos.

³² Fuente: <https://www.ibm.com/content/dam/connectedassets-adobe-cms/...>

- **Specificity:** evalúa la capacidad del modelo para identificar correctamente las instancias negativas. Se define como:

$$Specificity = \frac{TN}{TN + FP}$$

Un modelo con alta especificidad tiene pocos falsos positivos.

- **F1 Score:** La métrica en cuestión constituye la media armónica entre precisión y recall, proporcionando un equilibrio entre ambas métricas. Su cálculo se determina mediante la siguiente fórmula:

$$F1\ Score = \frac{Precision \times Recall}{Precision + Recall}$$

La relevancia de esta métrica radica en su capacidad para alcanzar un balance óptimo entre precisión y recall, lo que la convierte en un instrumento de gran utilidad en diversos contextos.

- **Accuracy:** expresa el porcentaje de predicciones correctas sobre el total de predicciones efectuadas. Su cálculo se determina mediante la siguiente fórmula:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Si bien constituye una métrica global, puede resultar menos representativa en conjuntos de datos desbalanceados.

El inconveniente principal asociado a las métricas de clasificación radica en su dependencia de la selección de un valor umbral para la determinación de la pertenencia de una instancia a la clase positiva o negativa. La elección de dicho umbral constituye un aspecto de suma importancia, puesto que incide de manera directa en el equilibrio entre las distintas métricas de evaluación.

En el caso de que se determine un umbral bajo, el modelo en cuestión clasificará un mayor número de instancias como positivas, incrementando así la sensibilidad. Este fenómeno implica que el modelo identificará la mayoría de los casos positivos, aunque también puede generar un alto número de falsos positivos.

Por el contrario, al establecer un umbral más elevado, se reducirá la cantidad de falsos positivos, incrementando la especificidad y asegurando que solo los casos con alta probabilidad sean clasificados como positivos. No obstante, esta estrategia puede conducir a un incremento en los falsos negativos, lo que implica que ciertos casos positivos podrían pasar desapercibidos para el modelo.

La sensibilidad y la especificidad se erigen como métricas opuestas que definen el comportamiento del modelo y su idoneidad para un problema específico. En contextos donde prevalece la necesidad de minimizar los falsos negativos, como en el diagnóstico de enfermedades, se tiende a optar por un umbral bajo que garantice una alta sensibilidad. En contraste, en contextos donde se prioriza la minimización de los falsos positivos, tales como en la detección de fraudes, se recomienda establecer un umbral alto para maximizar la especificidad.

Cuando se procede a la comparación de distintos modelos sin la necesidad de establecer un umbral concreto, se hace factible la utilización de métricas como el Área Bajo la Curva ROC (AUC-ROC). Esta métrica permite la evaluación del rendimiento global del modelo, mediante el análisis de la relación entre la sensibilidad y la especificidad, trazando la curva ROC (*Receiver Operating Characteristic*).

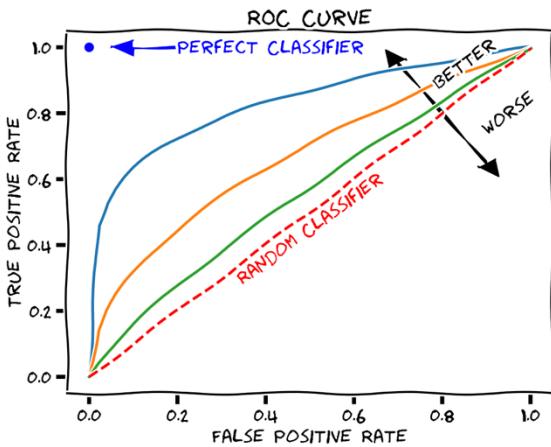


Figura 86: Ejemplo de gráfica de Curva ROC

Esta curva representa la tasa de verdaderos positivos (sensibilidad) en el eje Y, y la tasa de falsos positivos (especificidad) en el eje X. La ubicación de la curva en el espacio de decisión es relevante para la evaluación de la calidad del modelo. Un posicionamiento elevado y hacia la izquierda indica un modelo de mayor eficacia, ya que implica una detección precisa de los positivos y una minimización de los falsos positivos.

2.2.1.1 Estimación del error

El propósito esencial de la etapa de entrenamiento en una red neuronal radica en la identificación de los pesos óptimos que faciliten la realización de predicciones precisas por parte de la red, empleando el conjunto de datos de entrenamiento. La modificación de dichos pesos requiere la evaluación de la discrepancia entre las predicciones de la red y los valores esperados, lo que se denomina error de la red.

No obstante, no todos los errores tienen el mismo impacto en el rendimiento del modelo. En consecuencia, se implementa una estimación que mide la gravedad de ciertos valores de error, denominada pérdida. La función matemática encargada de calcular dicha pérdida, a partir de la diferencia entre la salida de la red y la salida esperada, se denomina función de error, función de pérdida o función de coste.

Existen diversas funciones de pérdida y su eficacia dependerá del tipo de problema que estemos abordando, pero en general cualquier función derivable podría servir como *loss function* puesto que nuestro objetivo será minimizarla. Por ejemplo, para problemas de clasificación una de las funciones más comúnmente utilizadas es:

- **Cross-Entropy (CE):** también conocida como *log error*, es especialmente útil en clasificaciones donde la salida representa la probabilidad de pertenencia a una clase, con valores entre 0 y 1. En problemas de clasificación binaria, donde solo existen dos clases, esta

función también se conoce como *Binary Crossentropy (BCE)* y se formula de la siguiente manera:

$$BCE = - \sum_{i=1}^n [y_i * \ln(p_i) + (1 - y_i) * \ln(1 - p_i)]$$

Siendo n el número de ejemplos, y_i la salida esperada y p_i la salida de la red, es decir, probabilidad de que el ejemplo pertenezca a la clase positiva.

En problemas de clasificación multiclase, se utiliza la función conocida como *Categorical Crossentropy (CCE)*. Esta función calcula la probabilidad de pertenecer a cada una de las clases, se calcula de la siguiente forma:

$$CCE = - \sum_{i=1}^n \sum_{j=1}^m y_j^i * \ln(p_j^i)$$

Siendo n el número de ejemplos, y_j^i sería la salida esperada para cada clase j , p_j^i salida predicha por el modelo para la clase j y m el número de clases del problema.

- **Mean Square Error (MSE):** También conocido como *L2 loss*, consiste en la media de las diferencias entre la salida esperada y la salida obtenida al cuadrado, siendo su fórmula:

$$MSE = \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^m (p_j^i - y_j^i)^2$$

2.2.1.2 Inicialización de los pesos

La asignación de valores iniciales apropiados constituye un paso crucial en el proceso de entrenamiento. Esta asignación, ya sea mediante la generación autónoma del diseño de la red o la utilización de una red preexistente como punto de partida, se erige como un componente fundamental para el entrenamiento efectivo de las redes neuronales. En este sentido, resulta imperativo asignar valores a todos los filtros de las capas de convolución y a las conexiones de las capas densas de la red, así como a cualquier capa que posea parámetros configurables.

Una de las propuestas más recurrentes es la de asignar ceros u otro valor específico en todos los pesos. Sin embargo, esta estrategia puede tener implicaciones indeseadas debido a la simetría. La aplicación de esta práctica a todos los pesos de la red resulta en una modificación uniforme que impide el aprendizaje efectivo. Ante esta situación, se han desarrollado técnicas de inicialización alternativas que introducen variabilidad en la red, permitiendo así el desarrollo de representaciones útiles de los datos y la mejora de la capacidad de generalización durante el entrenamiento.

Una segunda propuesta para evitar la simetría en la inicialización de los parámetros es realizarlo de forma aleatoria, aunque con cierto control, ya que, si los valores iniciales son demasiado pequeños, los valores de entrada que se propagan a lo largo de la red también serán muy pequeños y, por tanto, los valores que lleguen a las capas finales de la red no serán relevantes para el entrenamiento. De manera análoga, en el caso de valores excesivamente

pequeños, si los valores iniciales son demasiado pequeños, los valores de entrada que se propagan a lo largo de la red también serán muy pequeños y, en consecuencia, los valores que alcancen las capas finales de la red no serán relevantes para el entrenamiento. De manera similar, en el caso de valores excesivamente grandes, si los valores iniciales son demasiado grandes, en las capas finales los valores serán demasiado elevados y dificultarán el aprendizaje.

Por tanto, se torna imperativo establecer un rango adecuado de valores iniciales que evite la simetría y que permita una propagación efectiva de la información a lo largo de la red, garantizando así un entrenamiento eficiente y una mayor capacidad de generalización de la red.

Para inicializar los parámetros con valores iniciales óptimos, es posible recurrir a dos métodos populares que no son puramente aleatorios:

- **Inicialización de Xavier/Glorot:** Esta técnica, presentada en 2009 por Xavier Glorot y Yoshua Bengio [21]. Es una solución al problema de la inicialización aleatoria. El método inicializa los pesos de la red con una distribución uniforme de forma que la varianza y la desviación típica de estos sea igual a 1, teniendo en cuenta el número de unidades de la capa, de forma que su fórmula es:

$$w_j^i \sim N \left(0, \sqrt{\frac{1}{n}} \right)$$

- **Inicialización de He:** Método creado en 2015 por los creadores de ResNet [22], que ganó la competición de ImageNet, consiguiendo bajar el error poder debajo del valor del error humano (considerado como un 5%). Tienen en cuenta funciones de activación no lineales, en la que destaca la función ReLu. Este procedimiento posibilita la convergencia de modelos sumamente profundos, para los cuales la inicialización Xavier no resultaba efectiva. Su metodología se fundamenta en la asignación de un valor aleatorio tomado de una distribución uniforme:

$$w_j^i \sim U \left(0, \sqrt{\frac{2}{n}} \right)$$

Existe una alternativa a los métodos previamente mencionados, que consiste en la utilización de una red previamente entrenada en lugar de iniciar desde cero. Este procedimiento se basa en la premisa de que una red ha sido utilizada con éxito para resolver un problema complejo y general, o que un investigador ha compartido una red con pesos ya ajustados.

En lugar de iniciar el entrenamiento con una inicialización aleatoria para un nuevo problema, se aprovecha una red entrenada previamente, lo que permite reutilizar el conocimiento adquirido durante su entrenamiento y acelerar el aprendizaje. Este enfoque, denominado *transfer learning*, consiste en aplicar lo aprendido en un problema a otro, que suele ser más simple que el original. Este método permite que el entrenamiento comience desde una configuración más cercana al óptimo en comparación con una inicialización aleatoria, lo que resulta en una mejora en la eficiencia y el rendimiento del modelo.

2.2.1.3 Algoritmos de optimización

Como se explicó en apartados precedentes, el cálculo del error cometido permite el ajuste de los parámetros de la red. De este modo, el proceso se simplifica a un problema de optimización, en el que se deben encontrar los valores óptimos para cada parámetro de la red.

Para lograrlo, se procederá a la estimación de “la pendiente” de la función en un punto específico, o lo que es lo mismo, se efectuará el cálculo de la derivada de la función de error. Dicha pendiente muestra la dirección de máximo decremento del gradiente para cada uno de los pesos de la red, los cuales serán ajustados desde la última capa hacia las primeras con la intención de reducir el error en futuros ejemplos.

Como se ha mencionado anteriormente, la pendiente representa la dirección de la máxima disminución, la cual es calculada mediante la derivada parcial de cada uno de los coeficientes de la red, expresada en forma de vector:

$$\nabla_{\theta} J(\theta) = \left(\frac{\partial J(\theta)}{\partial \theta_0}, \frac{\partial J(\theta)}{\partial \theta_1}, \dots, \frac{\partial J(\theta)}{\partial \theta_n} \right)$$

Siendo J la función de error, θ el conjunto de pesos que constituyen la red y n el número total de pesos de la red.

A partir de esto, se procede a la actualización de cada uno de los pesos de la red, mediante la adición o la sustracción, según lo indicado como el opuesto a la dirección del gradiente (máximo descenso) (ver Figura 87)³³. Este procedimiento da como resultado:

$$\theta_{nuevo} = \theta_{antiguo} - \alpha \nabla_{\theta} J(\theta)$$

Donde α es el *learning rate*, establece si el valor de la actualización del peso del parámetro se hace en mayor o en menor medida.

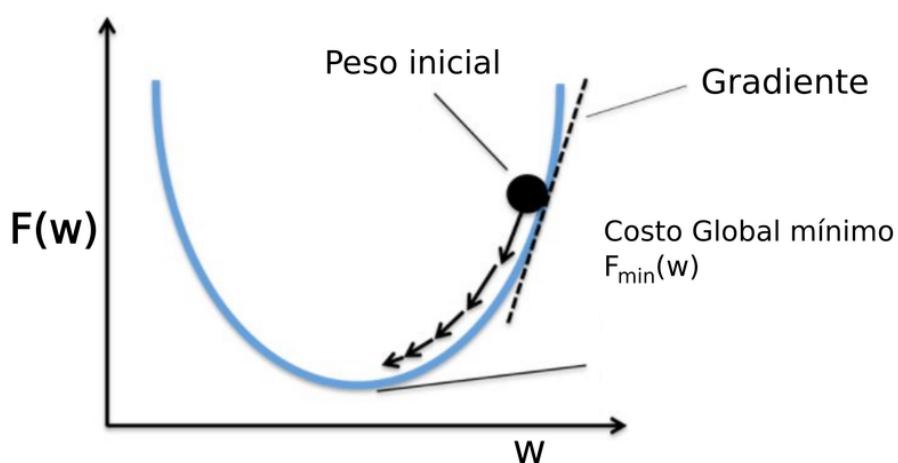


Figura 87: Representación gráfica de la dirección del gradiente

³³ Fuente: [https://www.researchgate.net/profile/Juan-Vasquez-Gomez/publication/329453137/figure/fig10/...](https://www.researchgate.net/profile/Juan-Vasquez-Gomez/publication/329453137/figure/fig10/)

A esta técnica se le conoce como descenso por gradiente y es muy utilizada por los principales algoritmos optimizadores. (ver Figura 88)³⁴.

- **Stochastic Gradient Descent (SGD):** Introducido por Herbert y Sutton en 1951 [24], el descenso por gradiente estocástico constituye una variación del descenso por gradiente original. En esta variante, se procede a la actualización de cada uno de los pesos de la red para cada uno de los ejemplos, en función de los errores cometidos:

$$\theta_{nuevo} = \theta_{antiguo} - \alpha \nabla_{\theta} J(\theta; x^{(i)}, y^{(i)})$$

Este algoritmo acelera el entrenamiento ya que se incrementa la actualización de los pesos y, por otro lado, uno de los principales inconvenientes es la actualización de todos y cada uno de los pesos por cada uno de los ejemplos, lo que eleva el computo.

- **SGD with Momentum:** Con el paso del tiempo, Boris T. Polyak propuso su metodología, denominada Polyak's Heavy Ball Method [25], la cual planteaba la incorporación de un factor de “memoria” con el objetivo de optimizar la convergencia en algoritmos iterativos. La implementación de un término de memoria en el Stochastic Gradient Descent constituye una optimización que propicia la atenuación de las actualizaciones del gradiente, con el propósito de fomentar la convergencia y acelerar el proceso de optimización. Este refinamiento se erige como un imperativo para mitigar oscilaciones superfluas y potenciar la celeridad en el ámbito del aprendizaje profundo. En contraste con la actualización de los parámetros mediante el gradiente actual, Momentum almacena una “velocidad” acumulada del gradiente pasado, permitiendo que la optimización se desenvuelva de manera más fluida y eficiente. Se introduce una variable v_t que representa la acumulación del gradiente en iteraciones previas y se actualiza según la siguiente fórmula:

$$v_t = \beta v_{t-1} + (1 - \beta) \nabla_{\theta} J(\theta)$$

$$\theta_{nuevo} = \theta_{antiguo} - \alpha v_t$$

- **Adagrad:** El algoritmo Adagrad, en su búsqueda de optimizar el algoritmo SGD, implementa un *learning rate* adaptativo para cada peso de la red, en lugar de emplear un único valor de tasa de aprendizaje para todos los parámetros. Adagrad ajusta de manera individualizada cada parámetro en función del gradiente acumulado. La premisa fundamental que subyace en Adagrad radica en que los pesos que exhiben mayores oscilaciones deben recibir una tasa de aprendizaje reducida, mientras que aquellos con variaciones más pequeñas deben mantener una tasa de aprendizaje elevada. De esta manera, el algoritmo se centra en ajustar los pesos más relevantes y minimizar la influencia de aquellos que no contribuyen de manera estable al descenso del gradiente.

$$\theta_{nuevo} = \theta_{antiguo} - \frac{\alpha}{\sqrt{G_{antiguo} + \epsilon}} \nabla_{\theta} L(\theta_{antiguo})$$

Donde α es la tasa de aprendizaje inicial, $G_{antiguo}$ es la suma acumulada de los cuadrados de los gradientes de la iteración previa, ϵ es un pequeño valor para evitar la división por cero.

- **RMSprop:** Este algoritmo se presenta como una solución al problema de Adagrad, mediante la adaptación del *learning rate* de cada peso utilizando la media móvil exponencial del

³⁴ Fuente: https://tiddler.github.io/assets/img/2016-12-07-optimizers/cifar_cnn.png

gradiente. En contraste con la simple acumulación de gradientes de Adagrad, la media móvil exponencial permite que los gradientes más antiguos se “olviden”, lo que impide una parada prematura del entrenamiento. La regla de actualización empleada por este algoritmo se presenta a continuación:

$$v_t = \rho v_{t-1} + (1 - \rho) [\nabla_\theta L(\theta)]^2$$

$$\theta_{nuevo} = \theta_{antiguo} - \frac{\alpha}{v_t + \epsilon} \nabla_\theta L(\theta)$$

Donde ρ es el *decay rate* y ϵ es un pequeño valor para evitar la división por cero.

- **Adam:** Adaptative Moment Estimation, es un algoritmo presentado en 2014 por Diederik Kingma [26], que calcula un *learning rate* adaptativo para cada parámetro basándose en la media del primer y segundo momento del gradiente. Combina las ideas de SGD con Momentum y RMSprop, utilizando momentos de primer y segundo orden para ajustar dinámicamente la tasa de aprendizaje en cada parámetro. Este algoritmo requiere la elección de tres parámetros: el *learning rate* y los *decay rates* del primer (β_1) y segundo (β_2) momento. Utilizando estos parámetros, Adam ajusta el *learning rate* mediante los siguientes estimadores:

$$v_t = \beta_1 * v_{t-1} - (1 - \beta_1) * g_t$$

$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2$$

$$\Delta w_t = -\alpha \frac{v_t}{\sqrt{s_t + \Delta \epsilon}} * g_t$$

$$w_{t+1} = w_t + \Delta w_t$$

Donde g_t representa el gradiente en w en el momento t .

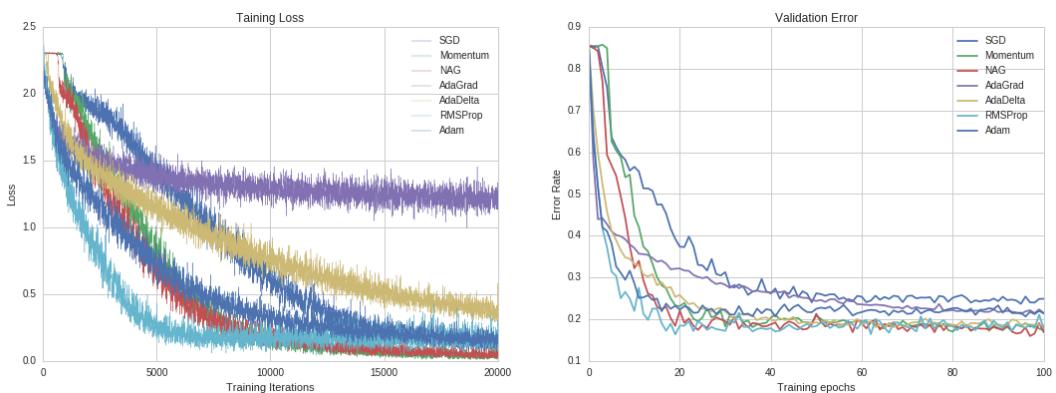


Figura 88: Comparación de algoritmos de optimización

2.2.1.5 Backpropagation

Tras el análisis de las limitaciones del perceptrón llevado a cabo por Minsky y Papert, las redes neuronales cayeron en el olvido durante un tiempo. Sin embargo, en la década de 1980 volvieron a cobrar relevancia gracias al trabajo de David E. Rumelhart y Geoffrey E. Hinton. Hinton y Ronald J. Williams [7], quienes demostraron cómo utilizar el algoritmo de propagación hacia atrás para entrenar redes neuronales con múltiples capas.

Este algoritmo se fundamenta en la regla de la cadena, un procedimiento empleado para calcular la derivada de una composición de funciones. Mediante este método, es factible determinar el gradiente de la función de pérdida con respecto a los pesos de las capas ocultas. A pesar de que no se cuente con el error en estas capas debido a la falta de una salida esperada, sí es posible identificar la influencia de cada peso en el error final de la red. Este proceso se logra mediante la descomposición del problema en una serie de operaciones encadenadas que permiten propagar el error desde la salida hacia las capas anteriores, ajustando los pesos de manera eficiente.

El algoritmo en cuestión se compone de dos fases principales, a saber, la propagación hacia adelante y la retropropagación del error. Mediante la implementación de estas dos fases, se logra el objetivo de minimizar la función de pérdida $J(\theta)$, mediante la optimización de los pesos W de manera que la red produzca salidas que se aproximen más a los valores reales. A continuación, se procederá a un análisis detallado de estas fases.

En la primera fase, también conocida en inglés como *Forward Pass*, la información se transmite a través de la red neuronal desde la entrada hasta la salida, atravesando las capas ocultas. Inicialmente, se introduce una entrada X en la red y cada neurona en las capas subsiguientes calcula su activación en función de los pesos W y los sesgos b actuales. La activación en cada capa l se obtiene mediante la ecuación:

$$z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}$$

$$a^{(l)} = \sigma(z^{(l)})$$

Donde σ es la función de activación (como por ejemplo ReLU, sigmoide, etc) que introduce la no linealidad en la red. Este proceso se repite capa por capa hasta alcanzar la capa de salida, donde se genera una predicción \hat{Y} basada en los pesos actuales. Posteriormente se realiza una comparación entre la predicción \hat{Y} y la salida real Y mediante una función de pérdida $J(\theta)$, que mide la eficacia de la red en su tarea. Esta función de pérdida será la que se use en la siguiente fase para ajustar los pesos y mejorar la precisión del modelo.

En la segunda fase, denominada en inglés *Backward Pass*, el propósito radica en ajustar los pesos de la red neuronal con el fin de minimizar la incertidumbre asociada a las predicciones. Este procedimiento se inicia en la capa de salida, donde se calcula la disparidad entre la salida esperada Y y la salida predicha \hat{Y} , generando un error que se manifiesta en términos del gradiente de la función de pérdida:

$$\delta^{(L)} = \frac{\partial J}{\partial a^{(L)}} * \sigma'(z^{(L)})$$

En consecuencia, el error se propaga hacia atrás a través de las capas de la red utilizando la regla de la cadena, lo que permite distribuir la contribución del error a cada peso en las capas anteriores:

$$\delta^{(l)} = (W^{(l+1)})^T \delta^{(l+1)} * \sigma'(z^{(l)})$$

Con estos valores de error en cada capa, se calculan los gradientes respecto a los pesos, siendo $\delta^{(l)}$ el error de cada capa:

$$\frac{\partial J}{\partial W^{(l)}} = \delta^{(l)} a^{(l-1)T}$$

Finalmente, los pesos de la red se actualizan utilizando el método de descenso por gradiente, (que vimos en el apartado anterior 2.2.1.3 Algoritmos de optimización), ajustándolos en la dirección opuesta al gradiente para minimizar la función de pérdida, donde α representa la tasa de aprendizaje:

$$W^{(l)} = W^{(l)} - \alpha \frac{\partial J}{\partial W^{(l)}}$$

Este proceso de retropropagación se reitera en cada iteración del entrenamiento hasta que la función de pérdida alcanza un valor óptimo, permitiendo que la red aprenda representaciones más precisas para la tarea que se pretende resolver.

2.2.2 Etapa de entrenamiento: Hiperparámetros

El entrenamiento de una red neuronal se fundamenta en la selección meticulosa de diversos parámetros, los cuales deben ser escrupulosamente definidos con anterioridad al inicio del proceso. La arquitectura de la red, *learning rate* y *batch size*³⁵, constituyen meramente una muestra de las múltiples decisiones que deben ser tomadas de manera precisa y anticipada.

Estos valores se denominan hiperparámetros, puesto que no son adquiridos por la red, sino que establecen las condiciones en las cuales se efectuará el entrenamiento. Su configuración incide directamente en la capacidad del modelo para hallar los valores óptimos de los parámetros (pesos y sesgos) que permitirán a la red realizar predicciones con precisión.

La selección de los hiperparámetros se fundamenta en múltiples criterios, tales como la experiencia previa, el examen de enfoques empleados en el estado de la técnica para problemas afines y la ejecución de pruebas preliminares sobre el conjunto de datos de entrenamiento. El proceso de ajuste de hiperparámetros tiende a llevarse a cabo de manera iterativa, mediante la realización de múltiples experimentos en los que se buscan mejoras progresivas respecto a configuraciones previas. Esto puede implicar la exploración de nuevas combinaciones de hiperparámetros o un enfoque en la recolección y mejora de los datos, garantizando que el modelo tenga información más representativa y de mayor calidad para su aprendizaje.

A continuación, exponemos los principales hiperparámetros involucrados en el entrenamiento de una red, junto con algunas recomendaciones para su definición.

³⁵ Término de origen inglés que se emplea para denotar el tamaño del lote de entrenamiento.

2.2.2.1 Arquitectura de red

La arquitectura de una red neuronal determina su capacidad de aprendizaje y debe estar alineada con la complejidad del problema que se busca resolver. Una estrategia común es utilizar una arquitectura previamente probada en problemas similares y con resultados contrastados, lo que puede ahorrar tiempo y esfuerzo en la configuración del modelo. Sin embargo, según el teorema *No Free Lunch* [27], no existe un modelo universalmente superior para todos los problemas, en otras palabras, si un algoritmo funciona bien para un conjunto específico de problemas, inevitablemente tendrá un rendimiento deficiente en otros.³⁶ En consecuencia, la selección de la arquitectura debe fundamentarse en pruebas empíricas, ajustes iterativos y una comprensión profunda de las características del problema en cuestión.

2.2.2.2 Técnica de inicialización de los pesos

La inicialización de los pesos en una red neuronal constituye un hiperparámetro crítico que ejerce una influencia significativa en la velocidad y estabilidad del entrenamiento. Diversas estrategias han sido propuestas para definir estos valores antes de iniciar el proceso de ajuste de los parámetros. Se recomienda, en la medida de lo posible, recurrir a la técnica de *transfer learning*, mediante la reutilización de los pesos de una red previamente entrenada en un problema de naturaleza similar. Esta práctica permite iniciar el proceso de entrenamiento desde una configuración más cercana al óptimo, lo que se traduce en una reducción del tiempo de entrenamiento y un aumento de la probabilidad de alcanzar una convergencia más efectiva.

En ausencia de un modelo preentrenado, es posible recurrir al uso de métodos de inicialización, como los mencionados en el apartado anterior 2.2.1.2 Inicialización de los pesos, los cuales contribuyen a mejorar la propagación de la señal a lo largo de la red y evitan problemas como la saturación de las funciones de activación. En contraste, una inicialización completamente aleatoria puede ocasionar un aumento en el tiempo de entrenamiento o incluso impedir que el modelo converja de manera apropiada.

2.2.2.3 Número de épocas

El entrenamiento de una red neuronal se estructura en un número determinado de épocas, cada una de las cuales representa un ciclo completo en el que el modelo procesa todo el conjunto de datos de entrenamiento.

En el caso de que el número de épocas sea insuficiente, es posible que la red no disponga de un tiempo suficiente para ajustar sus pesos, lo que puede resultar en un modelo con un alto error y un rendimiento deficiente, lo que se conoce como *underfitting*. Por otro lado, un número excesivo de épocas puede conducir al sobreajuste, denominado en inglés *overfitting*, en el que el modelo aprende de forma excesiva los datos de entrenamiento, pero pierde capacidad de generalización en datos nuevos, como se observa en la Figura 89³⁶.

³⁶ Fuente: <https://cdn.analyticsvidhya.com/wp-content/uploads/2020/02/Screenshot-2020-02-06-at-11.09.13.png>

El número óptimo de épocas es un aspecto crucial que debe determinarse en función de diversos factores. Estos factores incluyen la complejidad del problema, el tamaño del conjunto de datos y la arquitectura de la red. Para evitar el sobreentrenamiento, se recomienda el uso de técnicas como *early stopping*, que detiene el entrenamiento cuando el rendimiento en el conjunto de validación deja de mejorar.

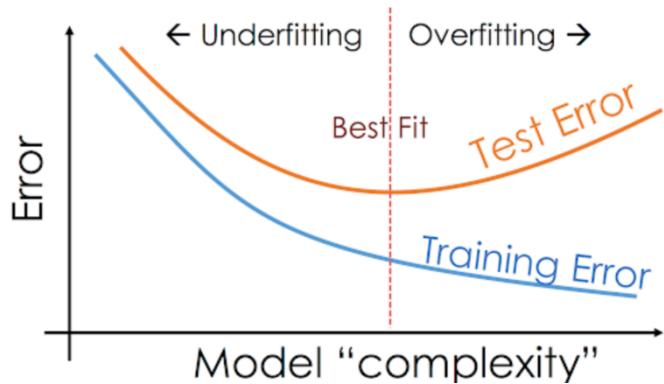


Figura 89: Comparativa de underfitting y overfitting en una gráfica train vs validation de la métrica loss del entrenamiento de una red

2.2.2.4 Batch Size

En lo que respecta al tamaño del lote de ejemplos, es pertinente señalar que la cantidad de datos empleados en el cálculo del valor de pérdida ejerce una influencia significativa en la configuración de la superficie de la función de pérdida y, por consiguiente, en la complejidad que enfrentará el algoritmo de optimización al intentar alcanzar el mínimo. En este contexto, se hace común el uso de potencias de dos para definir el tamaño del lote, siendo los valores 16, 32, 64 y 128 los más prevalentes.

2.2.2.5 Almacenamiento de los pesos

El almacenamiento de los pesos de la red constituye un aspecto de vital importancia en el marco del entrenamiento, toda vez que el propósito del experimento radica en la obtención de los pesos óptimos para la problemática en cuestión. La práctica más aconsejable consiste en el almacenamiento de los pesos en el momento en el que se evidencie una mejora en la función de pérdida o en alguna métrica de validación, tales como la precisión o el AUC-ROC. Esta práctica garantiza que los parámetros almacenados correspondan al mejor desempeño alcanzado durante el entrenamiento. Para gestionar este proceso, se suelen emplear estrategias como el *early stopping* con *checkpointing*, donde se guarda el mejor modelo en función de un criterio específico y se evita el sobreentrenamiento.

2.2.2.6 Algoritmo de optimización

El algoritmo de optimización determina la forma en que se emplean las derivadas obtenidas mediante el método de *backpropagation* para actualizar los pesos de la red neuronal. Si bien *backpropagation* constituye el único método para calcular los gradientes en las capas ocultas, la manera en que se aplican estos gradientes depende del optimizador seleccionado.

Algunos algoritmos, como el descenso de gradiente, son más rápidos de calcular en comparación con otros más avanzados, como Adam. Una de las razones por las que se opta por el descenso de gradiente es su simplicidad, lo que facilita el entrenamiento y reduce el riesgo de sobreajuste. Además, en el caso de que el algoritmo llegue a un punto de equilibrio, es probable que este sea un mínimo aplanado en lugar de un mínimo pronunciado, lo que aumenta las posibilidades de que se haya encontrado el mínimo global y no un mínimo local. (ver Figura 90)³⁷.

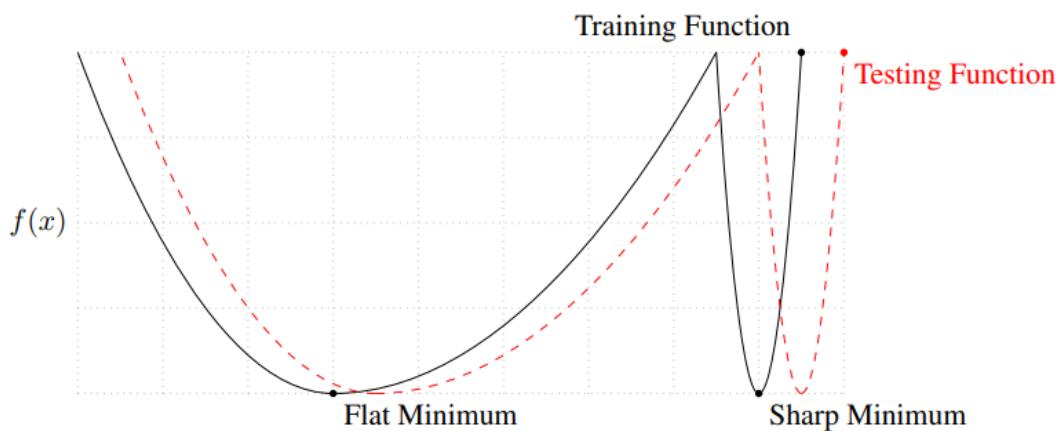


Figura 90: Gráfica conceptual de los mínimos aplanados y los profundos, donde el eje-Y indica el valor de la función de pérdida y el eje-X los parámetros

No obstante, se ha comprobado que optimizadores como Adam, RMSprop o Adagrad incluyen mejoras como tasas de aprendizaje adaptativas y *momentum*, lo que permite una convergencia más rápida y estable en problemas con funciones de pérdida más. Por tanto, se puede concluir que la elección del algoritmo de optimización dependerá del problema, el tamaño del conjunto de datos y la arquitectura de la red.

Entre los parámetros del algoritmo, el *learning rate* es uno de los más críticos, no solo en el contexto del algoritmo de optimización, sino también en el proceso global de entrenamiento. Este parámetro determina la magnitud de los ajustes efectuados en los pesos de la red durante cada iteración. Un *learning rate* excesivamente alto puede provocar que el modelo no converja o

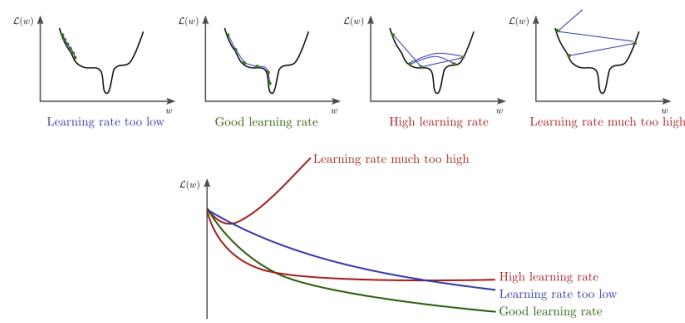


Figura 91: Efecto en un entrenamiento aplicando diferentes learning rate

³⁷ Fuente: <https://imgur.com/hgGxHZe>

incluso diverja, oscilando sin alcanzar un mínimo. Por otro lado, un *learning rate* excesivamente bajo puede ocasionar que el entrenamiento se vuelva excesivamente lento y quede atrapado en mínimos locales. (ver Figura 91)³⁸.

2.2.2.7 Métricas

En la evaluación de modelos, se observa la existencia de una diversidad de métricas diseñadas para satisfacer los requerimientos específicos de diversos tipos de problemas. La selección de dichas métricas debe estar en consonancia con los objetivos del modelo, ya que son estas métricas las que determinan su rendimiento y, simultáneamente, sirven como criterio para la determinación del momento oportuno para almacenar los pesos que posteriormente serán empleados en la fase de inferencia.

En problemas de clasificación, se suelen emplear métricas como la precisión, el recall, la F1-score y el AUC-ROC, mientras que en regresión, métricas como el error cuadrático medio, el error absoluto medio o el coeficiente de determinación son más adecuadas.

La elección de las métricas de evaluación también depende del tipo de problema que se está abordando, en tanto que, en ciertos escenarios, algunas métricas pueden adquirir una relevancia superior a otras. A modo ilustrativo, en el ámbito de la detección de enfermedades, es fundamental minimizar los falsos negativos, habida cuenta de que una predicción errónea que clasifique a un paciente sano como enfermo podría acarrear consecuencias fatales. En contraste, en problemas como la detección de fraudes financieros, se busca minimizar los falsos positivos para evitar que transacciones legítimas sean erróneamente clasificadas como fraudulentas.

Por lo tanto, al seleccionar las métricas, no solo se debe considerar el rendimiento general del modelo, sino también el impacto que diferentes tipos de errores pueden tener en el contexto específico del problema.

2.2.2.8 Función de pérdida

La función de pérdida constituye un componente esencial en el entrenamiento de una red neuronal. Su rol radica en cuantificar la discrepancia entre la salida predicha por el modelo y el valor esperado, permitiendo la optimización de los parámetros de la red. En lo que respecta a la selección de la función de pérdida, se permite la utilización de cualquier función derivable, garantizando así la capacidad del algoritmo de *backpropagation* para calcular los gradientes y actualizar los pesos de manera eficiente.

La elección de la función de pérdida se determina en función del tipo de problema que se busca resolver. En el caso de la regresión, se emplean comúnmente funciones como el error cuadrático medio (*MSE*) o el error absoluto medio (*MAE*), mientras que en la clasificación se utilizan funciones como la entropía cruzada (*Cross-Entropy Loss*) o su variante binaria (*Binary Cross-Entropy, BCE*). En problemas más específicos, es posible diseñar funciones de pérdida personalizadas que penalicen ciertos errores de manera diferente según su impacto en la tarea a resolver.

³⁸ Fuente: <https://i0.wp.com/spotintelligence.com/wp-content/uploads/2024/02/...>

Una vez que se han establecido los hiperparámetros, se procederá a la realización de un experimento, y en función de las métricas obtenidas, se determinará la siguiente prueba a ejecutar. Este proceso iterativo permite ajustar el modelo progresivamente hasta alcanzar un rendimiento óptimo.

En la siguiente sección, se presentan algunas directrices para garantizar un entrenamiento exitoso e identificar posibles problemas que puedan surgir durante el proceso. La identificación de estos problemas permitirá optimizar la selección de los hiperparámetros, comprender qué parámetros requieren ser modificados y cómo, por ende, mejorar el desempeño del modelo de manera más eficiente.

2.2.3 Etapa de entrenamiento: Control y seguimiento

Es habitual llevar un registro de los valores de pérdida y de las métricas a lo largo del proceso de entrenamiento, con el propósito de realizar un análisis de su evolución. En este sentido, se almacena los valores de pérdida en cada iteración y, cuando se considera pertinente, las métricas calculadas en cada paso. Asimismo, al concluir cada época, se almacenan los valores promedio tanto para el conjunto de entrenamiento como para el conjunto de validación.

Durante el proceso de entrenamiento, se anticipa una disminución progresiva en los valores de pérdida y una mejora en las métricas de rendimiento. Además, se espera que los resultados obtenidos en el conjunto de entrenamiento sean similares a los del conjunto de validación. En caso de que se observen comportamientos anómalos, tales como una falta de mejora o una divergencia significativa entre ambas curvas, esto puede indicar problemas como sobreajuste, subajuste o una mala configuración de los hiperparámetros, lo que requerirá ajustes en el modelo o en la estrategia de entrenamiento.

El análisis de los valores de pérdida y métricas se lleva a cabo mediante una representación gráfica en la que el eje X representa las iteraciones o épocas del entrenamiento, mientras que el eje Y muestra el valor de pérdida o la métrica seleccionada, como se muestran en Figura 92³⁹.

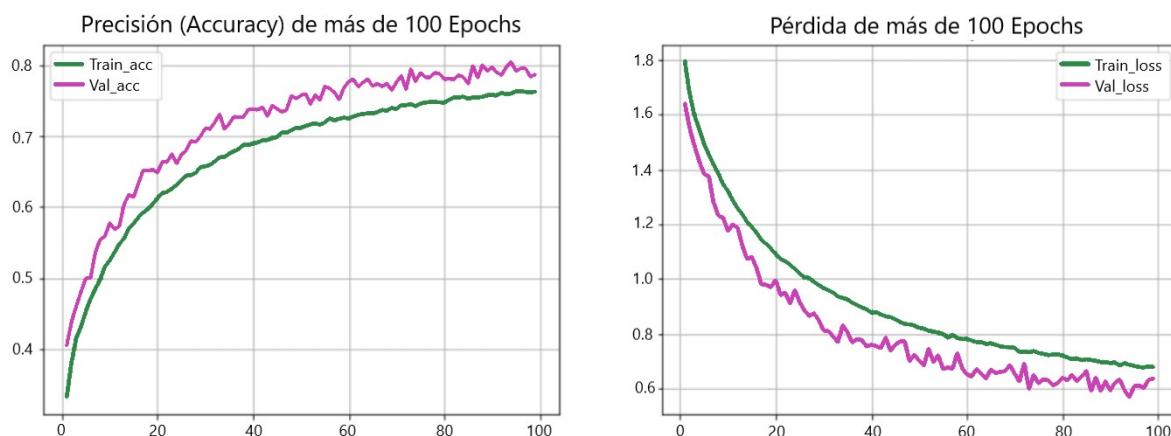


Figura 92: Ejemplo de gráficas de loss y acccuracy en un entrenamiento de 100 épocas

En los estadios iniciales del entrenamiento, el valor de pérdida tiende a ser elevado debido a que los pesos de la red han sido asignados de acuerdo con la técnica de inicialización

³⁹ Fuente: https://sitiobigdata.com/wp-content/uploads/2023/09/precision_perdida_epoch.jpg

seleccionada. No obstante, conforme progresó el entrenamiento y el algoritmo de optimización ajusta los parámetros, la pérdida comienza a disminuir de manera significativa tras un determinado número de épocas.

Se observa que las métricas de evaluación exhiben una tendencia a iniciar con valores bajos y a presentar un incremento progresivo a medida que se reduce la pérdida. Este fenómeno se explica por la correlación entre la reducción de la pérdida y la capacidad del modelo para realizar predicciones más precisas, lo que se manifiesta en un aumento de las métricas de precisión y recall.

Se espera que la disminución de la pérdida esté asociada con la optimización de las métricas, dado que ambas reflejan la eficacia del modelo. Sin embargo, si se evidencia que la pérdida continúa reduciéndose mientras las métricas se estabilizan o muestran un deterioro.

El análisis de las gráficas de pérdida y métricas permite detectar la mayoría de los problemas que pueden surgir durante el entrenamiento, como el sobreajuste o el infraajuste. En el caso de que la pérdida registrada en el conjunto de entrenamiento sea reducida, mientras que la pérdida en el conjunto de validación se mantenga en niveles elevados, se puede inferir que el modelo se encuentra sobreajustando los datos de entrenamiento, lo que sugiere una incapacidad para generalizar a nuevos datos. En esta situación, la red memoriza el conjunto de entrenamiento en lugar de desarrollar habilidades para aprender patrones generales, lo que resulta en un rendimiento deficiente en lo que respecta a la validación. Para corregir este sobreajuste, se han propuesto diversas estrategias, tales como el incremento del tamaño del conjunto de datos, la reducción de la complejidad del modelo mediante la eliminación de capas o neuronas, y la implementación de técnicas de regularización como *dropout* o L_2 .

En contraste, cuando existe una correlación entre la pérdida del conjunto de entrenamiento y la de validación, ambas permanecen en niveles elevados, se evidencia un subajuste en el modelo. Este fenómeno sugiere una limitación en la capacidad de la red para asimilar adecuadamente los patrones asociados al problema, lo que impide una reducción significativa del error. En tal caso, la solución propuesta consiste en incrementar la complejidad del modelo mediante la incorporación de capas o neuronas adicionales. Esta medida se anticipa que permitirá a la red exhibir una capacidad de representación ampliada, lo que a su vez facilitará la reducción continua del error.

Una vez identificado un problema en el entrenamiento de la red neuronal, es posible implementar diversas técnicas para corregirlo y optimizar la convergencia del modelo. A continuación, se presentan algunas de las estrategias más comúnmente utilizadas:

- **Equilibrar el número de muestras de las clases:** El equilibrio en el número de ejemplos por clase es una estrategia clave para evitar problemas de sobreajuste cuando el conjunto de datos está desbalanceado. No siempre el sobreajuste es causado por un modelo demasiado grande, sino que también puede deberse a que una clase es predominante dentro del conjunto de entrenamiento.

Para evitar este problema, se acude a varias técnicas. En el proceso de validación de datos, es crucial mantener un equilibrio entre los ejemplos presentados, garantizando que el conjunto de datos de validación contenga una cantidad uniforme de ejemplos por cada categoría o clase de datos. Este equilibrio es fundamental para asegurar la precisión y representatividad de las métricas obtenidas durante el proceso de validación. En lo que respecta a la distribución de las clases en los lotes de entrenamiento, se ha observado que la implementación de un balanceo en los lotes resulta en una distribución más uniforme de las

clases dentro de cada lote. Este procedimiento se ha implementado con el propósito de evitar que la red aprenda patrones erróneos basados en la frecuencia de aparición de una clase particular. Otras de las técnicas más utilizadas es la de remuestreo, se pueden aplicar métodos como el submuestreo de la clase mayoritaria o el sobre muestreo de la clase minoritaria para balancear el conjunto de entrenamiento.

- **Aplicar criterios de actualización del *learning rate*:** En los algoritmos de optimización que no emplean un *learning rate adaptativo*, es común aplicar políticas de actualización del *learning rate* para evitar que este mantenga un valor constante durante todo el proceso de entrenamiento. Al inicio del entrenamiento, un *learning rate* elevado permite que el modelo descienda rápidamente en la función de pérdida, acercándose más rápidamente al mínimo.

No obstante, en etapas posteriores, este valor puede volverse problemático, impidiendo que el modelo refine su ajuste si los pasos de actualización son demasiado grandes. Para evitar este problema, se pueden utilizar diferentes estrategias para actualizar el *learning rate*. Una de ellas es la disminución por épocas fijas, que consiste en reducir el valor del *learning rate* después de un cierto número de épocas predefinidas.

En segundo lugar, se encuentra la reducción basada en la pérdida, que se aplica cuando el valor de la pérdida se mantiene estable durante varias épocas, lo que permite ajustes más finos.

Por último, se puede emplear la *decay exponencial*, que se caracteriza por usar una función exponencial para reducir progresivamente el *learning rate* a lo largo del entrenamiento.

Estas políticas permiten optimizar el proceso de aprendizaje, asegurando una convergencia estable y evitando que el modelo quede atrapado en un mínimo local o no termine de ajustar correctamente los pesos.

- **Normalización:** la normalización de los datos constituye un paso fundamental en el proceso de preprocessamiento, ya que facilita la optimización de la red neuronal y garantiza su estabilidad. En este sentido, cuando los datos poseen una media de cero y una desviación típica de uno, el modelo puede alcanzar una capacidad de aprendizaje más eficiente, ya que todas las dimensiones se encuentran en un rango similar.

Sin embargo, si alguna de las dimensiones de entrada presenta valores significativamente superiores en comparación con las demás, la superficie de pérdida puede experimentar una elongación, lo que complica la optimización y genera oscilaciones en el entrenamiento. Estas oscilaciones pueden dificultar el proceso de descenso del algoritmo de optimización hacia el mínimo óptimo de la función de pérdida.

- **Regularización:** La regularización constituye una metodología empleada para mitigar la complejidad del modelo y prevenir el sobreajuste. Este procedimiento implica la incorporación de una penalización en el cálculo del error, la cual está determinada en función del valor de los pesos de la red neuronal. Los modelos de mayor complejidad y propensos al sobreajuste tienden a exhibir valores elevados en sus pesos, lo que resulta en la generación de cambios bruscos en la salida a partir de pequeñas variaciones en la entrada. La regularización, por tanto, se presenta como un mecanismo que mitiga el efecto anteriormente descrito, promoviendo la reducción de los valores de los pesos y, por ende, suavizando la función de decisión y potenciando la capacidad de generalización del modelo.
- **Batch normalization:** es una técnica que permite la normalización de las activaciones de las distintas capas de la red neuronal, en contraposición a la normalización de los datos de

entrada. A medida que los datos se propagan a través de las capas de la red, sus distribuciones pueden experimentar cambios significativos, lo que puede resultar en valores extremos y dificultar la convergencia del modelo.

Para abordar esta problemática, se implementa el ajuste de la distribución de las activaciones en cada capa mediante la normalización de los valores dentro de cada mini-lote, lo que implica la resta de la media y la división por la desviación estándar del lote actual de ejemplos. Posteriormente, se aplican dos parámetros adicionales de escalado y desplazamiento para facilitar la adquisición de distribuciones más flexibles, en caso de ser requerido.

Dropout: su propósito es evitar el sobreajuste en dichos sistemas. Su funcionamiento se fundamenta en la desactivación aleatoria de un conjunto de neuronas pertenecientes a una capa específica durante el proceso de entrenamiento. Esta estrategia transforma la red en un sistema compuesto por múltiples redes de menor complejidad, las cuales aprenden a cooperar para resolver el problema en cuestión, promediando sus salidas. En el caso de que Dropout se aplique en una capa con n neuronas, cada entrenamiento se realizará sobre una versión reducida de la red, lo que equivale a entrenar hasta 2^n subredes diferentes. En cada iteración, se seleccionará una de estas subredes y se ajustarán los pesos solo de las neuronas activas.

- **Data augmentation:** este enfoque de mejora de la generalización de un modelo sin la necesidad de modificar su arquitectura ni aplicar técnicas adicionales de regularización. Este enfoque implica la ampliación artificial del conjunto de datos, una estrategia particularmente beneficiosa cuando se cuenta con un número limitado de muestras y se busca evitar el sobreajuste.

En el contexto de un problema de clasificación de imágenes médicas, donde el conjunto de datos proviene de un grupo reducido de pacientes, existe la probabilidad de que la red neuronal termine memorizando las características específicas de dichas imágenes en lugar de aprender patrones generales. Para abordar esta situación, se pueden aplicar diversas transformaciones a las imágenes originales, tales como rotaciones, volteos, recortes, escalados, variaciones en el brillo y contraste, o la adición de ruido aleatorio.

La selección de técnicas de aumento de datos se determina en función del tipo de problema específico. En el ámbito del procesamiento de lenguaje natural, se pueden generar sinónimos, reorganizar frases o eliminar palabras irrelevantes para la tarea. En el campo del reconocimiento de voz, se puede modificar la velocidad o el tono de los audios. El objetivo principal es generar nuevas muestras a partir de las existentes que sean lo suficientemente distintas del dato original, pero que conserven su significado para que el modelo pueda seguir identificándolas.

Apartado 3

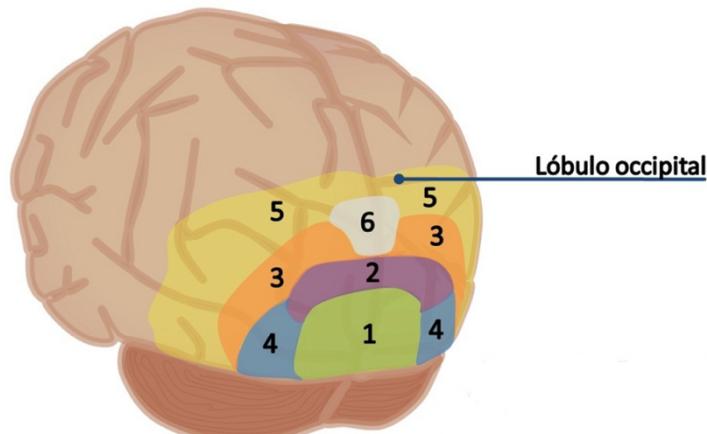
Redes Neuronales Convolucionales

3.1 Introducción

Como vimos en el capítulo anterior, las redes neuronales artificiales tienen una inspiración biológica en la estructura del cerebro humano. En el cerebro, millones de neuronas están interconectadas entre sí y se comunican mediante impulsos eléctricos; si una neurona recibe suficiente estímulo, se activa y genera una nueva señal hacia otras neuronas. Esta cadena de estímulos que da lugar a los comportamientos complejos del ser humano es lo que se intenta imitar con las redes neuronales artificiales.

Dentro de las redes neuronales encontramos diversas clases entre las cuales destacan las redes neuronales convolucionales (CNN). Como todas las redes neuronales, se basa en el cerebro humano, pero en este caso específicamente en el córtex visual. El córtex es responsable de percibir y reconocer elementos y formas que el ojo humano observa, comenzando con elementos simples como líneas o curvas (corteza visual primaria o región V1) y progresando a patrones más complejos como rostros (áreas visuales corticales extraestriadas o regiones V2, V4) (ver Figura 93)⁴⁰.

Área encargada de la visión:



Lóbulo occipital: Lóbulo cerebral posterior dedicado fundamentalmente a la visión.

Áreas:

- 1.- Exploración e inscripción general.
- 2.- Visión estereoscópica.
- 3.- Profundidad y distancia.
- 4.- Color.
- 5.- Movimiento.
- 6.- Determinación de la posición absoluta del objeto.

Figura 93: Áreas del lóbulo cerebral posterior

⁴⁰ Fuente: <https://blogger.googleusercontent.com/img/b/R29vZ2xl/...>

Para replicar este comportamiento las CNNs utilizan diferentes tipos de capas más allá de las densamente conectadas de las redes tradicionales. Estas nuevas capas permiten procesar datos en formatos bidimensionales, lo cual supera las capacidades de las redes clásicas y permite abordar problemas relacionados con la visión artificial.

La principal diferencia entre las redes tradicionales y las convolucionales se encuentra en la capacidad que tienen estas últimas para manejar estructuras bidimensionales, en las clásicas las entradas se expresan en una única capa plana, lo que hacía inviable introducir una imagen directamente, ya que se perdía toda la información estructural y el número de entradas sería excesivo haciendo que el experto debía analizar las imágenes y extraer los descriptores clave, para alimentar la red. Con las CNNs esto no ocurre así, las imágenes se pueden recibir directamente como entrada eliminando la necesidad del análisis previo.

Esta extracción de características se efectúa a través de varias capas. Las capas de convolución son las encargadas de aprender a reconocer patrones en las imágenes, combinadas con otras operaciones y capas son capaces de detectar elementos estructurales cada vez más complejos a medida que la red se profundiza, desde la detección de bordes simples hasta formas elaboradas como rostros o gatos. (Figura 94)⁴¹.

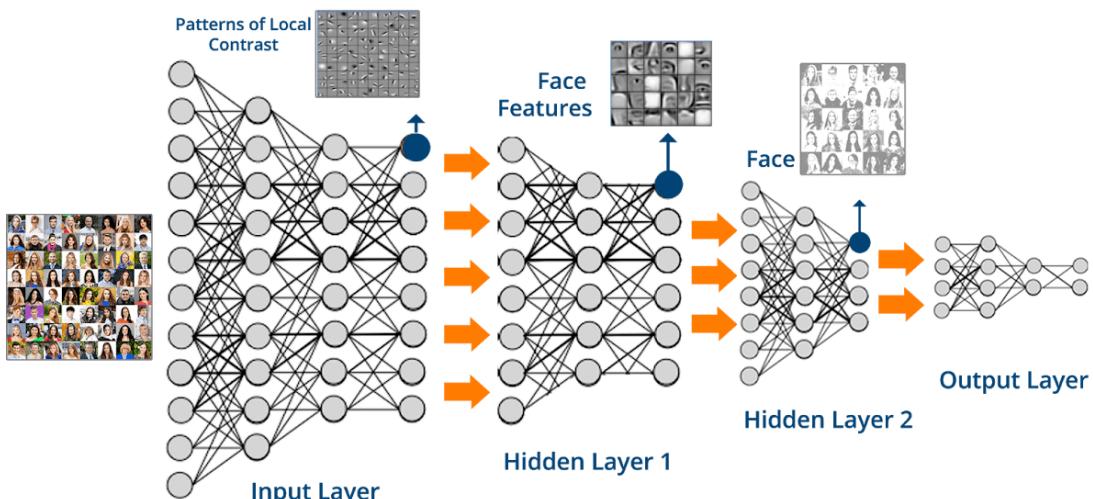


Figura 94: Ejemplo de extracción de características en capas de convolución

Entrenar redes convolucionales es similar a entrenar redes neuronales clásicas, pero con la diferencia de que también se ajustan los valores de los kernels de convolución. Las capas de convolución ayudan a la red a extraer características cada vez más representativas. Sin embargo, encontrar los valores óptimos para los filtros depende de la arquitectura de la red. Tener más capas y filtros permite reconocer características más complejas, pero también incrementa el número de parámetros a ajustar, lo que aumenta el tiempo de entrenamiento. Si hay demasiados parámetros y la red no está bien inicializada, es posible que el modelo no logre converger.

Para entrenar una red neuronal convolucional es necesario realizar múltiples pruebas con diferentes estructuras para determinar el número óptimo de capas para el problema en cuestión. Además, es importante ajustar la cantidad y el tamaño de los filtros para encontrar un equilibrio

⁴¹ Fuente: https://4.bp.blogspot.com/-pUmPcAL5LEc/XbLLk_UzhNI/AAAAAAAAB8M/...

entre eficiencia y eficacia. En el resultado final influyen la correcta inicialización de los parámetros, la elección del algoritmo de entrenamiento, las funciones de error a utilizar, ...

Es común aplicar una capa de pooling después de una operación de convolución para reducir el tamaño de los datos y mantener las características más relevantes, disminuyendo así el número de neuronas y parámetros y reduciendo significativamente el tiempo de entrenamiento.

3.2 Tipos de capas

Las redes neuronales convolucionales, como mencionamos anteriormente, están compuestas por una multitud de capas diferentes a las de las redes clásicas, que emplean otro tipo de capas. Estas capas permiten realizar una variedad de operaciones dentro de la red. Cada capa tiene características únicas y cumple una función específica. A continuación, describiremos algunas de las capas más utilizadas en las redes neuronales convolucionales y daremos ejemplos de su aplicación

3.2.1 Capas de convolución

Esta capa se encarga de extraer y transformar las características presentes en las matrices que recibe como entrada. Estas matrices pueden ser la imagen original o la salida de una capa previa de la red. Dado que el objetivo principal de esta capa es procesar las características de los datos, denominaremos a sus entradas y salidas como *features maps* cuando estas provengan de una capa previa o se dirigen a una capa siguiente.

El procedimiento mediante el cual esta capa identifica y procesa las características de los datos es aplicando filtros o *kernels* diseñados para el reconocimiento de diferentes patrones y formas en los *features maps*. (ver Figura 95)⁴².

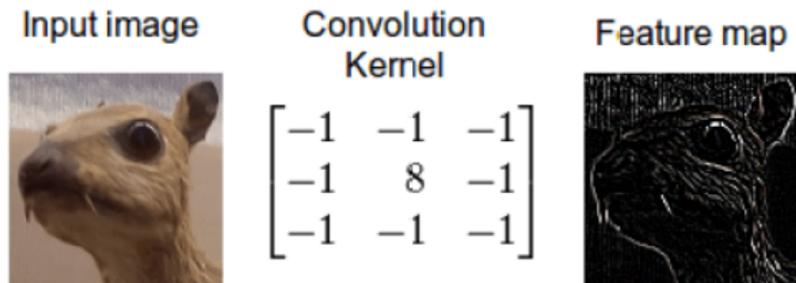


Figura 95: Detección de bordes en imagen mediante convolución

La convolución es una operación matemática simple que involucra dos matrices de entrada generalmente de tamaños diferentes, pero con el mismo número de dimensiones, que se combinan para generar una tercera matriz. Estas matrices de entrada pueden ser los mapas de

⁴² Fuente: <https://i0.wp.com/timdettmers.com/wp-content/uploads/2015/03/convolution.png?resize=500,193&ssl=1>

características recibidos de una capa previa o la propia imagen original, y el *kernel* pertenece a la capa de convolución actual.

El proceso de convolución consiste en deslizar el filtro o kernel sobre todas las posiciones del mapa de características de entrada donde dicho filtro encaje. Es decir, todos los valores de la matriz del filtro se superponen a un valor del mapa de características. Por cada posición en la que se sitúa el filtro sobre la matriz de entrada, se realiza una suma del producto de cada elemento del filtro por el valor subyacente de la matriz sobre la que está situado. El resultado de esta operación se almacena en un nuevo valor en el mapa de características de salida, como se puede

$$\begin{array}{|c|c|c|c|c|} \hline 35 & 40 & 41 & 45 & 50 \\ \hline 40 & 40 & 42 & 46 & 52 \\ \hline 42 & 46 & 50 & 55 & 55 \\ \hline 48 & 52 & 56 & 58 & 60 \\ \hline 56 & 60 & 65 & 70 & 75 \\ \hline \end{array} * \begin{array}{|c|c|c|} \hline -2 & -1 & 0 \\ \hline -1 & 1 & 1 \\ \hline 0 & 1 & 2 \\ \hline \end{array} = \begin{array}{|c|c|c|c|c|} \hline & & & & \\ \hline & & 78 & & \\ \hline & & & & \\ \hline & & & & \\ \hline & & & & \\ \hline \end{array}$$

$$35 \cdot (-2) + 40 \cdot (-1) + 41 \cdot 0 + 40 \cdot (-1) + 40 \cdot 1 + 42 \cdot 1 + 42 \cdot 0 + 46 \cdot 1 + 50 \cdot 2 = 78$$

$$\begin{array}{|c|c|c|c|c|} \hline 35 & 40 & 41 & 45 & 50 \\ \hline 40 & 40 & 42 & 46 & 52 \\ \hline 42 & 46 & 50 & 55 & 55 \\ \hline 48 & 52 & 56 & 58 & 60 \\ \hline 56 & 60 & 65 & 70 & 75 \\ \hline \end{array} * \begin{array}{|c|c|c|} \hline -2 & -1 & 0 \\ \hline -1 & 1 & 1 \\ \hline 0 & 1 & 2 \\ \hline \end{array} = \begin{array}{|c|c|c|c|c|} \hline & & & & \\ \hline & & 87 & & \\ \hline & & & & \\ \hline & & & & \\ \hline & & & & \\ \hline \end{array}$$

$$40 \cdot (-2) + 41 \cdot (-1) + 45 \cdot 0 + 40 \cdot (-1) + 42 \cdot 1 + 46 \cdot 1 + 46 \cdot 0 + 50 \cdot 1 + 55 \cdot 2 = 87$$

Figura 96: Operación de convolución

apreciar en la Figura 96⁴³.

Esta capa es fundamental porque permite a la red neuronal convolucional extraer y reconocer patrones complejos en los datos de entrada, lo que es esencial para tareas como el reconocimiento de imágenes y la detección de objetos.

Matemáticamente la operación de convolución de aplicar un filtro K bidimensional de tamaño $m \times n$ sobre una imagen I también bidimensional de tamaño $M \times N$, se define con el operador $*$ como:

$$(I * K)(i, j) = \sum_{k=1}^m \sum_{l=1}^n I(i+k-1, j+l-1) \cdot K(k, l)$$

donde i y j son las posiciones de la imagen resultante y recorren todas las filas y columnas respectivamente de la imagen I en donde el filtro K puede superponerse. En otras palabras, el resultado del filtro está desplazado.

Existe además la posibilidad de agregar un término constante más a la operación, llamado *bias*. De esta forma se asemeja más a los modelos matemáticos originales de las neuronas, quedando la fórmula:

$$(I * K)(i, j) = \sum_{k=1}^m \sum_{l=1}^n I(i+k-1, j+l-1) \cdot K(k, l) + b$$

⁴³ Fuente: [https://www.researchgate.net/publication/292187589/figure/fig1/...](https://www.researchgate.net/publication/292187589/figure/fig1/)

Durante el entrenamiento, la elección de los filtros de convolución se realiza de forma automática ya que la etapa de convolución se aplica a un conjunto de filtros permitiendo al algoritmo seleccionar las características más relevantes.

Aunque estos filtros parezcan no tener sentido para la percepción humana, para la máquina pueden ser descriptores clave. Sin embargo, algunos parámetros importantes de la capa de convolución, como el tamaño inicial de la entrada de las imágenes, el tamaño de los kernels y la forma en que se aplican son definidos por el experto que desarrolla la red. Estos parámetros son cruciales para el rendimiento y la precisión del modelo. Vamos a explicarlos en más detalle:

- **Tamaño del filtro:** Es necesario indicar las dimensiones que tendrá el filtro de la convolución. Cuando mayor es el *kernel*, las características que se detecten podrán ser mayores.
- **Número de convoluciones:** Cantidad de mapas de características que se obtendrán como salida después de aplicar la capa de convolución actual. Cada operación de convolución recibe como entrada todos los mapas de características de la capa previa.
- **Padding:** Técnica utilizada para evitar la pérdida de píxeles en los extremos de una imagen. Normalmente al aplicar un *kernel*, los píxeles de los bordes pueden quedar fuera del alcance, lo que reduce el tamaño de la imagen resultante. El *padding* resuelve este problema ampliando los límites de la imagen original, de manera que el *kernel* pueda encajar en todas las posiciones, incluidos los bordes. Los píxeles añadidos durante el *padding* suelen rellenarse con ceros, replicar el valor del píxel más cercano o utilizar una forma de espejo.
- **Stride:** Define cómo se desliza el *kernel* sobre la matriz o imagen de entrada durante la convolución. Su valor determina el tamaño del "salto" que el *kernel* da mientras se desplaza. Por ejemplo, con un *stride* de 1, el *kernel* se aplica a cada elemento de la imagen, pero con un *stride* de 3, se aplicaría cada 3 elementos. Esto significa que genera una salida con un tamaño aproximadamente igual a un tercio del tamaño de la entrada.

La combinación de parámetros como el tamaño de *padding*, el *kernel* y el *stride* que escogamos influye directamente en la dimensión de la imagen de salida y en el tamaño de entrada a la siguiente capa de la red.

Por lo tanto, estos parámetros deben elegirse con cuidado. Para calcular las dimensiones de los mapas de características resultantes después de aplicar una operación de convolución, se puede utilizar la siguiente fórmula:

$$\text{Dimensión de salida} = \frac{(W - K + 2P)}{S} + 1$$

donde W es el tamaño de entrada, K es el tamaño del *kernel*, S es el tamaño del *stride* y P es el tamaño del *padding*.

3.2.2 Capas de activación

Al igual que en las redes neuronales clásicas, en las redes neuronales convolucionales también se pueden utilizar estas funciones después de las capas de convolución. Esto añade una característica de no-linealidad ampliando la variedad de problemas que se pueden abordar al permitir que la red aprenda patrones más complejos.

La función de activación se aplica a cada elemento del mapa de características obtenido en la capa de convolución previa, alterando sus valores, pero no sus dimensiones. (ver Tabla 28: Principales funciones de activación)

3.2.3 Capas de pooling

Las capas de *pooling* o submuestreo, tienen como principal objetivo reducir las dimensiones de los mapas de características generados por una capa previa, generalmente una capa de convolución, sin perder las características más importantes presentes en dichos mapas. Esto permite disminuir el número de parámetros de la red y reduce el tiempo de entrenamiento y ejecución.

El *pooling* es especialmente útil en redes diseñadas para problemas de clasificación, donde es crucial condensar la información de las imágenes en tamaños muy reducidos, manteniendo únicamente los elementos que optimicen la separabilidad entre clases. Esto facilita la conexión a una capa final que actúe como clasificador.

Funcionalmente, la operación de *pooling* es similar a la convolución en el sentido de que ambas operan sobre la imagen de manera localizada, deslizando una ventana (equivalente al *kernel* en la convolución) sobre la imagen. Por lo tanto, el *pooling* comparte con la convolución parámetros como el tamaño del *kernel* y el *stride*.

A diferencia de la capa de convolución, en la operación de *pooling* el valor de salida no se calcula mediante una operación que involucre elementos propios de la capa actual, sino únicamente los valores del mapa de características. Esto convierte a la capa de *pooling* en una capa no entrenable de la red, cuyo comportamiento se define en el momento de construir la red. Las dos opciones más empleadas son (ver Figura 97)⁴⁴:

- **MaxPooling:** Se asigna el máximo valor de todos los valores de la ventana.
- **AveragePooling:** Se asigna la media de los valores de la ventana.

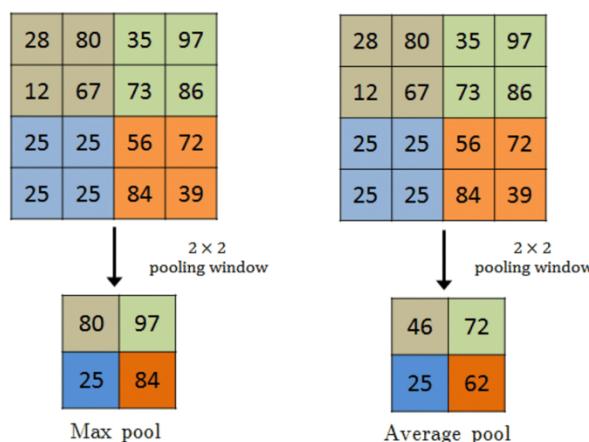


Figura 97: Operaciones maxpooling y averagepooling

⁴⁴ Fuente: [https://www.researchgate.net/publication/340255272/figure/fig4/...](https://www.researchgate.net/publication/340255272/figure/fig4/)

3.2.4 Capas upsampling y transposed convolution

Se utilizan principalmente en redes neuronales convolucionales cuyo objetivo es devolver una imagen en la salida. En una red neuronal convolucional típica, las capas de convolución y *pooling* reducen el tamaño de la entrada original de datos a medida que la información avanza a lo largo de la red. Sin embargo, en ciertas aplicaciones, como las redes de segmentación, es necesario que la salida mantenga las mismas dimensiones que la imagen de entrada para corresponder cada píxel con su clase asignada.

La operación de *upsampling* se encarga de re-escalar la matriz de entrada al tamaño deseado, calculando el valor de cada elemento mediante métodos de interpolación. Los métodos más utilizados para esta tarea incluyen:

- **Nearest Neighbor Interpolation:** Asigna a cada píxel re-escalado el valor del píxel más cercano en la imagen original (ver Figura 98)⁴⁵.

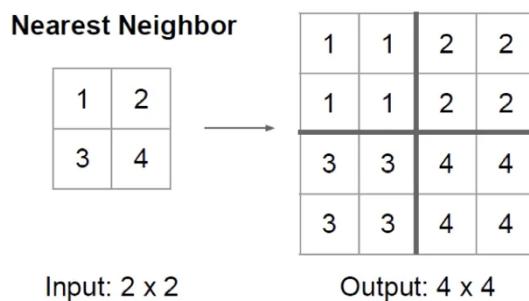


Figura 98: Técnica de Nearest Neighbor

- **Bed of nails:** Se copian los valores de los pixeles de entrada y se colocan en la misma posición y se rellenan el resto de las posiciones de ceros (ver Figura 99)⁴⁶.

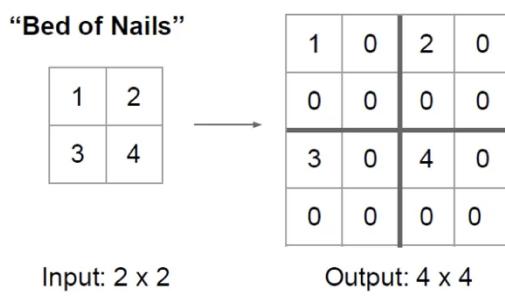


Figura 99: Técnica de Bed of nails

- **Max-unpooling:** En esta operación, recordando las posiciones de los pixeles del *max-pooling* de una capa de *pooling* anterior, a la cual debe estar conectada, estos pixeles de la entrada se

⁴⁵ Fuente: https://miro.medium.com/v2/resize:fit:720/format:webp/1*9N9FVYalaVAk-aalcBVYaA.png

⁴⁶ Fuente: https://miro.medium.com/v2/resize:fit:720/format:webp/1*LQVVqK8YI4ndcU6NS4UOxA.png

re-escalalarán y se colocarán en las posiciones de los píxeles de la capa de *max-pooling* anterior a la que este conectada, y se rellena el resto de los píxeles de la matriz con ceros. Para comprenderlo mejor podemos observar la Figura 100⁴⁷.

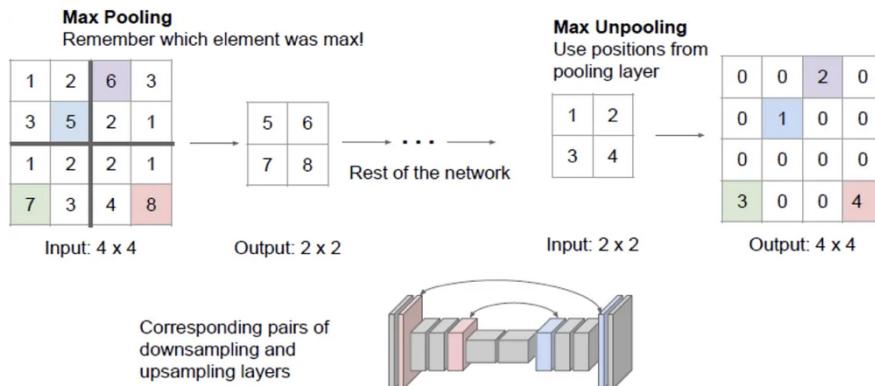


Figura 100: Técnica de max-pooling y max-unpooling

Sin embargo, en lugar de utilizar métodos de interpolación para aumentar los mapas de características, ya que nuestro interés radica en conservar las propiedades más importantes, podemos aplicar una operación que revierta los cambios efectuados por una capa de convolución. Esta operación es realizada por la capa de *transposed convolution*.

Para comprender mejor esta operación y los resultados que produce, primero plantearemos la operación de convolución de otra manera equivalente, mediante multiplicaciones de matrices.

Cuando realizamos una convolución sobre una imagen de tamaño 4x4 desplazando un *kernel* de tamaño 2x2, con un *stride* igual a 1 y sin *padding*, existirán 9 posiciones distintas en la imagen sobre las que el *kernel* se situará. Estas 9 ventanas se pueden representar como 9 matrices de 4x4 (el tamaño de la imagen) en las que todos los valores son 0, excepto en las posiciones donde se sitúa el filtro de la convolución, que contendrán los valores del filtro. A continuación, estas 9 matrices se apllanan en vectores de una dimensión, y se concatenan creando una matriz de 9 filas y 16 columnas. Si aplreamos la imagen de entrada de 4x4 en un vector columna de longitud 16, podemos aplicar una multiplicación matricial entre la matriz descrita anteriormente y la imagen, resultando en un vector columna de longitud 9. La imagen resultante de la convolución, dados los parámetros mencionados y empleando la fórmula descrita, tendrá un tamaño de 3x3. Reorganizando el vector columna en una matriz de dimensión 3x3, obtenemos la imagen tras aplicar la convolución. Este proceso se ilustra en la Figura 102.

Una vez comprendido el proceso de la convolución como una simple multiplicación de matrices, podemos aplicar un enfoque similar para aumentar las dimensiones de un mapa de características, de modo que los nuevos valores dependan de un *kernel* cuyos valores se ajustan durante el entrenamiento de la red.

Si deseamos ampliar un mapa de características de tamaño 3x3 a 4x4, la matriz que realizará esta operación debe cumplir con un requisito fundamental: sus dimensiones. Como se describió anteriormente, las imágenes de entrada y salida se apllanan en forma de vectores de una dimensión. En este ejemplo, la entrada tendría una longitud de 9 y la salida una longitud de 16. Por lo tanto, la matriz intermedia entre la entrada y la salida deberá tener 16 filas y 9 columnas, siguiendo la propiedad fundamental del producto de matrices (Figura 101). Existe una relación evidente entre las dimensiones de la matriz utilizada para realizar la convolución y la matriz

⁴⁷ Fuente: https://miro.medium.com/v2/resize:fit:1100/format:webp/1*b0NUJ-7IJnrljrzAc07BzQ.png

utilizada para realizar la operación inversa. Ambas matrices tienen el mismo tamaño, pero sus dimensiones están intercambiadas, son transpuestas. Esta relación entre las matrices da nombre a la operación que estamos tratando: *transposed convolution*

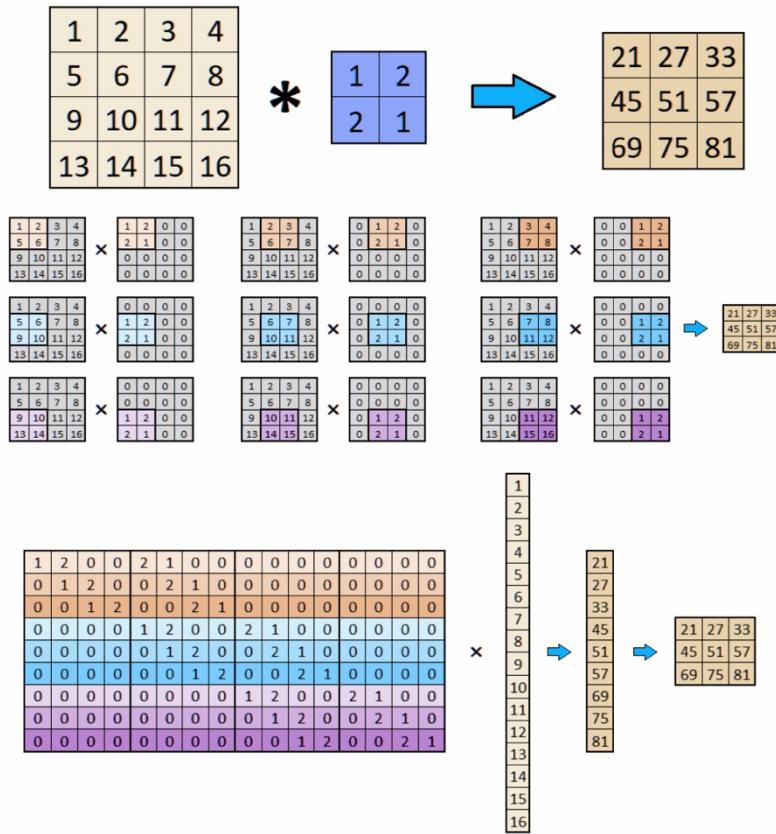


Figura 102: Operación de convolución como multiplicación de matrices

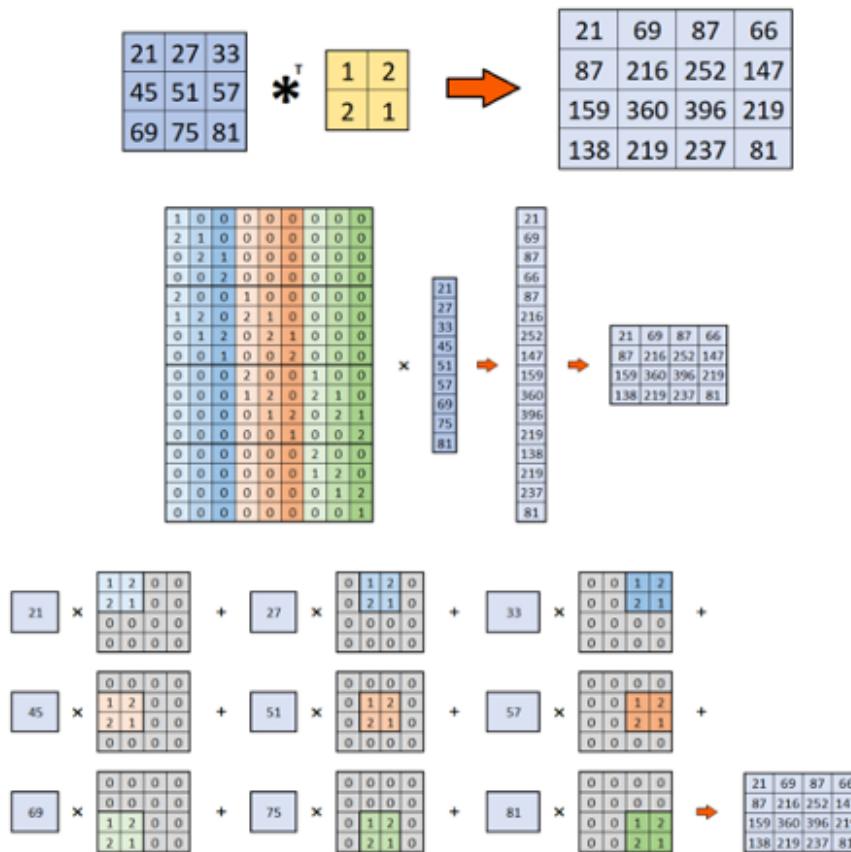


Figura 101: Operación de convolución traspuesta

3.2.5 Capas de softmax

Empleadas cuando se desea transformar la salida de una red neuronal en probabilidades, permitiendo así que la salida indique la probabilidad de que la entrada pertenezca a cada una de las clases. La operación que realiza la capa *softmax* se define como vimos en apartados anteriores, mediante la Ecuación 3. Esta transformación asegura que las salidas sumen a uno y cada salida sea interpretada como una probabilidad, lo que hace que su uso resulte sencillo en tareas de clasificación en problemas con varias clases.

3.2.6 Capas Fully Connected

También conocidas como capas densas, son esenciales en los modelos diseñados para la clasificación de imágenes. Cada neurona de la capa está conectada con todas las neuronas de la capa anterior, suelen ubicarse al final de las redes neuronales convolucionales, después de que las capas de convolución y *pooling* hayan extraído y simplificado las características necesarias de la imagen de entrada.

El proceso generalmente implica que, antes de pasar de una capa convolucional a una capa *fully connected*, la salida convolucional (que es una matriz $M \times N$) pasa por una capa *flatten*. Esta capa *flatten* convierte la matriz en un vector unidimensional, preparándolo para la entrada en la capa *fully connected*.

Para configurar estas capas, ajustaremos los siguientes parámetros:

- **Número de neuronas:** Son las neuronas que forman la capa y definen el número de salidas de esta. El número de entradas de esta capa depende del tamaño que tengan las capas de convolución previas.
- **Función de activación:** Para esta función suele elegirse una de las que se usan en las capas de activación, que definimos en el apartado 2.1.2 Funciones de activación.

3.3 Arquitecturas populares

3.3.1 AlexNet

Desarrollada en 2012 por Alex Krizhevsky, Ilya Sutskever y Geoffrey Hinton [13], AlexNet⁴⁸ marcó un hito en la historia de la visión por ordenador al ganar la competición ImageNet, con una mejora significativa en precisión respecto a modelos anteriores.

Esta red introdujo una arquitectura más profunda y compleja que LeNet (arquitectura que ya se introdujo en el apartado anterior 2.1.2 Conexiónismo (1980 – 1995)), compuesta por cinco capas convolucionales combinadas con capas de *max-pooling* y tres capas totalmente conectadas al final. Una de sus principales innovaciones fue el uso de la función de activación ReLU, que permitió una convergencia más rápida durante el entrenamiento. Además, implementó técnicas como *dropout* para reducir el sobreajuste y entrenamiento distribuido en GPU, lo que facilitó el manejo de grandes volúmenes de datos. AlexNet supuso el inicio de una nueva era en el uso de redes neuronales profundas para la clasificación de imágenes a gran escala.

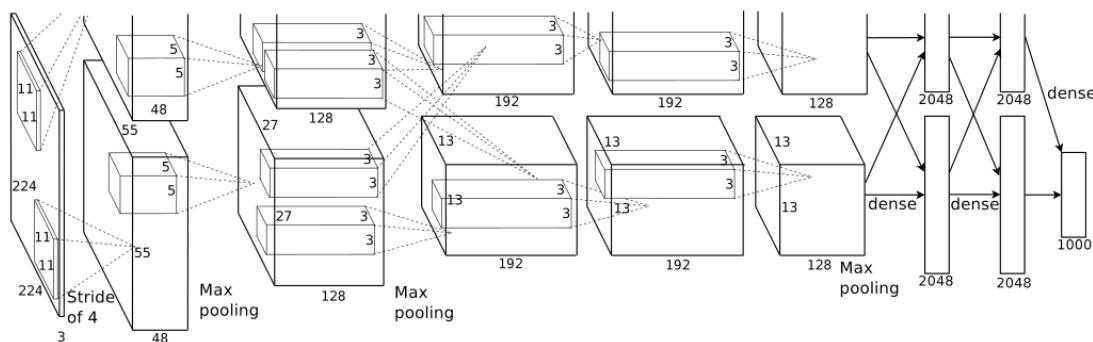


Figura 103: Arquitectura AlexNet

3.3.2 VGG

El modelo de aprendizaje profundo VGG fue desarrollado en 2014 por Karen Simonyan y Andrew Zisserman, investigadores del Visual Geometry Group de la Universidad de Oxford [28].

Su contribución más significativa radica en la demostración de que la implementación sistemática de convoluciones pequeñas (3x3) y capas con estructura repetitiva puede optimizar de manera notable el desempeño en tareas de clasificación de imágenes.

La red se presentó en dos versiones principales, VGG16⁴⁹ y VGG19, con 16 y 19 capas de profundidad, respectivamente, y obtuvo el segundo lugar en la competición ImageNet de ese año. El modelo VGG emplea bloques de capas convolucionales seguidos de *max-pooling*, finalizando con tres capas *fully connected* y una capa *Softmax* para la clasificación. A pesar de que sus modelos poseen un elevado número de parámetros, VGG se distinguió por su simplicidad

⁴⁸ Fuente: https://en.wikipedia.org/wiki/AlexNet#/media/File:AlexNet_Original_block_diagram.svg

⁴⁹ Fuente: https://miro.medium.com/v2/resize:fit:720/format:webp/0*0M8CobXpNwFDCmOQ

estructural y su eficacia, lo que la convirtió en una arquitectura base ampliamente utilizada en tareas de *transfer learning* y visión por computadora aplicada.

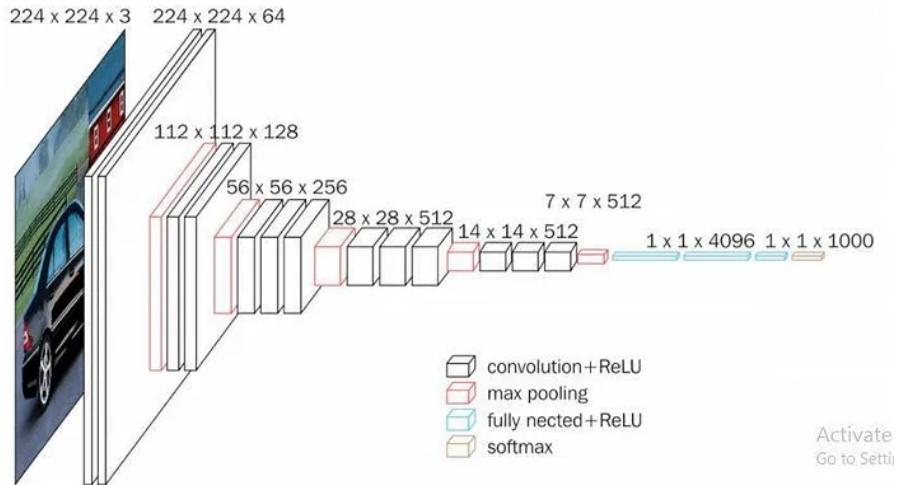


Figura 104: Arquitectura VGG16

3.3.3 GoogleLeNet

En el año 2014, el modelo de red neuronal GoogLeNet⁵⁰, también conocida como Inception V1, fue presentado por el grupo de investigación de Christian Szegedy et al. [14]. Este modelo resultó ganador del concurso ImageNet de ese mismo año, con un error de clasificación del 6,67 %.

La innovación principal consistió en la introducción del módulo Inception, que posibilita la realización de múltiples operaciones convolucionales de diversas dimensiones (1x1, 3x3 y 5x5) de manera simultánea dentro del mismo bloque, combinando sus salidas de manera eficiente. Este avance posibilitó la concepción de una red profunda, pero computacionalmente optimizada, lo que resultó en una reducción significativa del número de parámetros en comparación con otras arquitecturas coetáneas. Además, se implementó el uso de global average-pooling en lugar de capas fully-connected, lo que contribuyó a disminuir el riesgo de sobreajuste y el consumo de memoria. Con una profundidad total de 22 capas, GoogLeNet representó un cambio de paradigma en el diseño de arquitecturas CNN, influenciando múltiples versiones posteriores de Inception.

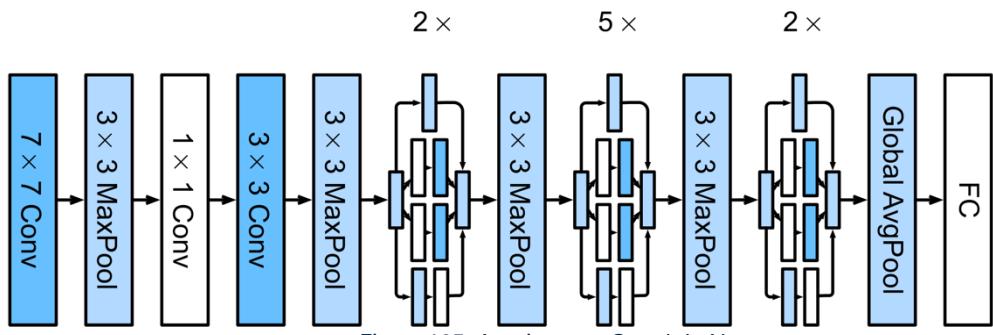


Figura 105: Arquitectura GoogLeNet

⁵⁰ Fuente: <https://upload.wikimedia.org/wikipedia/commons/thumb/c/ca/...>

3.3.4 ResNet

ResNet (Residual Network) fue introducida en 2015 por Kaiming He et al. [29], y supuso un avance decisivo en el diseño de redes neuronales profundas al resolver eficazmente el problema del desvanecimiento del gradiente. Su arquitectura se basa en el concepto de conexiones residuales (*skip connections*), que permiten a la red aprender funciones residuales en lugar de funciones directas.

Estas conexiones permiten el flujo de información a través de la red y posibilitan el entrenamiento de modelos significativamente más complejos sin pérdida de rendimiento. La versión más conocida, ResNet-50, consta de 50 capas, aunque existen variantes más profundas como ResNet-101 y ResNet-152. ResNet-50 (ver **Error! Reference source not found.**)⁵¹ obtuvo el primer puesto en la competición ImageNet 2015, alcanzando un error de clasificación del 3,57 %, y desde entonces se ha convertido en un modelo de referencia en múltiples aplicaciones de visión artificial, tanto en tareas de clasificación como de detección.

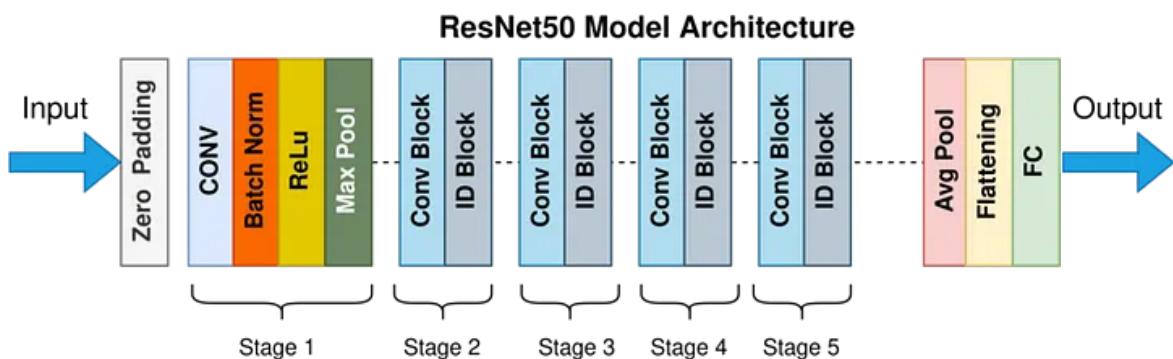


Figura 106: Arquitectura ResNet-50

3.3.5 DenseNet

DenseNet (Red Neuronal Convolucional Conectada Densamente) (ver Figura 107)⁵² fue presentada en 2017 por Gao Huang et al. como una evolución del diseño residual introducido en ResNet. Su característica principal es la conectividad densa entre capas, la cual se define como la capacidad de las redes neuronales de transmitir la salida de una capa a todas las capas siguientes dentro del mismo bloque.

Este enfoque ha demostrado una mejora significativa en la propagación del gradiente, fomentando la reutilización de características y reduciendo la redundancia de parámetros. Como resultado, se ha logrado entrenar modelos más eficientes en términos de memoria y computación. DenseNet ha demostrado un alto rendimiento en la clasificación de imágenes, logrando una excelente relación entre precisión y número de parámetros. Se han propuesto variantes como DenseNet-121, DenseNet-169 y DenseNet-201, donde el número hace referencia a la profundidad de la red.

⁵¹ Fuente: https://miro.medium.com/v2/resize:fit:720/format:webp/1*VM94wVftxP7wkiKo4BjfLA.png

⁵² Fuente: <https://amaarora.github.io/images/densenet.png>

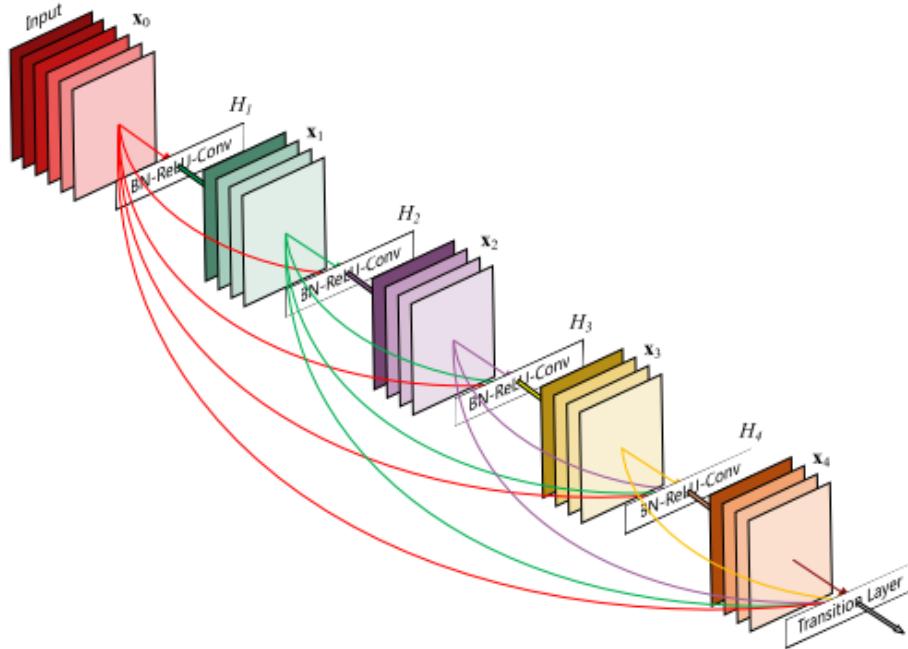


Figura 107: Arquitectura DenseNet

3.4 Arquitectura utilizada en este trabajo

Para el desarrollo de este proyecto, se implementó la arquitectura EfficientNet-B5, seleccionada tras una fase comparativa entre distintas redes preentrenadas disponibles en TensorFlow para aplicar técnicas de transfer learning con pesos ajustados sobre el dataset ImageNet. Esta evaluación incluyó arquitecturas como MobileNetV2, ResNet50, InceptionV3 y EfficientNet en sus diferentes variantes.

EfficientNet-B5 exhibió un equilibrio notable entre precisión, eficiencia computacional y capacidad de generalización, lo que la posicionó como la opción más apropiada para los objetivos planteados. Su escalado compuesto de profundidad, anchura y resolución permite maximizar el rendimiento sin incurrir en un coste computacional excesivo, lo que resulta especialmente ventajoso en contextos donde se requiere un alto nivel de precisión sin disponer de recursos de cómputo ilimitados.

Con el propósito de asistir a toda aquella persona que esté empleando este documento como referencia para su proyecto, se ha elaborado un resumen que detalla las principales arquitecturas que se consideran candidatos para el desarrollo del proyecto, junto con sus métricas más relevantes. Este resumen se ha elaborado con el objetivo de facilitar la toma de decisiones en función de las necesidades específicas de cada caso. Toda esta información puede verse en Tabla 29, disponibles en el enlace de la [clase applications de keras \(tensorflow\)](#).

Retomando nuestra arquitectura seleccionada, EfficientNet-B5 es una de las variantes de la familia EfficientNet, introducida en 2019 por Mingxing Tan y Quoc V. Le [30]. Esta arquitectura se basa en el escalado compuesto, una técnica que ajusta de manera equilibrada la profundidad, anchura y resolución de la red para mejorar la eficiencia y precisión.

EfficientNet-B5 hace uso de bloques *MBConv* con convoluciones invertidas y módulos *Squeeze-and-Excitation (SE)*, lo cual optimiza la extracción de características con un bajo costo computacional. Con una resolución de entrada de 456x456 píxeles, este modelo exhibe un alto rendimiento en ImageNet, destacándose por su equilibrio entre precisión y eficiencia en comparación con arquitecturas más extensas y onerosas en términos computacionales.

Tabla 29: Tabla comparativa de las principales arquitecturas disponibles en tensorflow

Modelo	Precisión Top-1 (%)	Precisión Top-5 (%)	Tamaño de imagen	Nº de parámetros
MobileNetV2	71.8%	91.0%	224x224	3.5M
ResNet50	76.2%	92.9%	224x224	25.6M
InceptionV3	77.9%	93.7%	299x299	23.8M
VGG16	71.3%	89.8%	224x224	138M
VGG19	71.7%	90.0%	224x224	143.7M
DenseNet121	74.9%	92.2%	224x224	8M
EfficientNetB0	77.1%	93.3%	224x224	5.3M
EfficientNetB1	79.1%	94.4%	240x240	7.8M
EfficientNetB2	80.1%	94.9%	260x260	9.2M
EfficientNetB3	81.6%	95.7%	300x300	12M
EfficientNetB4	82.9%	96.4%	380x380	19M
EfficientNetB5	83.6%	96.7%	456x456	30M
EfficientNetB6	84.0%	96.8%	528x528	43M
EfficientNetB7	84.4%	97.0%	600x600	66M

La arquitectura EfficientNet se fundamenta en una serie de módulos que utilizan un enfoque optimizado para mejorar la eficiencia computacional y el rendimiento en la clasificación de imágenes. A continuación, se abordará la descripción de cada uno de los módulos⁵³ en relación con EfficientNet-B5:

- **Módulo 1:** Se implementa una convolución separable en profundidad (*Depthwise Conv2D*), seguida de una normalización por lotes (*Batch Normalization*) y una función de activación. Posteriormente, se implementa una operación de promedio *global-pooling*, seguida de una capa de reescalado y dos convoluciones adicionales. Este módulo puede representar el bloque final de la red antes de la capa de clasificación.

⁵³ Fuente: https://miro.medium.com/v2/resize:fit:828/format:webp/1*cwMpOJNhwoeosjwW-usYvA.png

- **Módulo 2:** Se aborda una temática similar a la del módulo anterior, si bien se incorpora el concepto de *Zero Padding*. Este último puede ser parte de un bloque convolucional con tamaños de filtro que requieren ajuste en los bordes de la imagen de entrada. Esta práctica es común en las etapas más profundas de la red.
- **Módulo 3:** Un bloque característico de EfficientNet, donde se integran capas de convolución con normalización y reescalado. La implementación de *Global Average Pooling* sugiere que constituye un componente esencial en la etapa de clasificación final.
- **Módulo 4:** En este sentido, se evidencia la operación de multiplicación, lo cual es indicativo del mecanismo de *Squeeze-and-Excitation* (SE), utilizado en EfficientNet para mejorar la capacidad de la red en la selección de características relevantes. Se implementa una combinación de convoluciones con normalización por lotes para estabilizar el entrenamiento.
- **Módulo 5:** Este módulo presenta similitudes con el anterior, sin embargo, se ha incorporado *Dropout*, una técnica empleada para prevenir el sobreajuste. Esta observación sugiere que el módulo en cuestión pertenece a la porción final de la red, ubicada antes de la capa de salida.

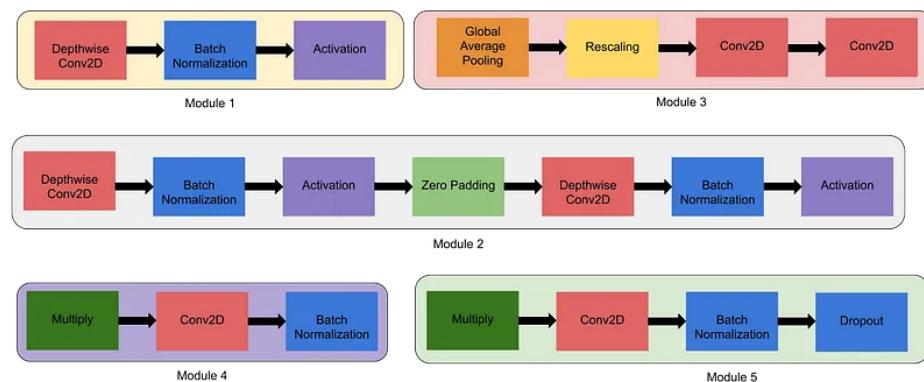


Figura 108: Diferentes tipos de módulos de EfficientNet

Una vez presentados los diversos módulos, es posible proceder a la comprensión de la configuración de la arquitectura EfficientNet-B5, así como de la organización de sus bloques de convolución y módulos internos, como se evidencia en la Figura 109⁵⁴.

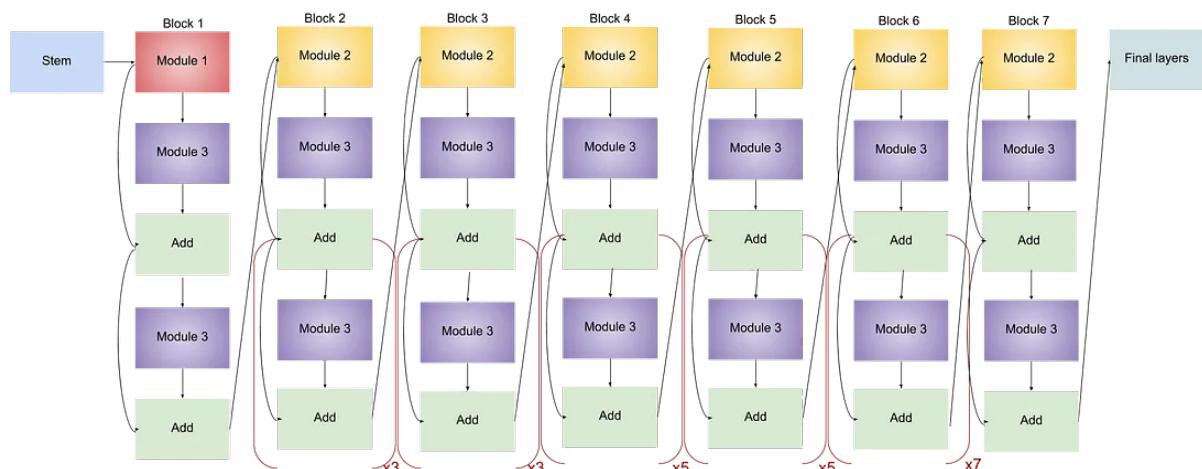


Figura 109: Arquitectura EfficientNet-B5

⁵⁴ Fuente: https://miro.medium.com/v2/resize:fit:1100/format:webp/1*6vH0nsxj0_-kHxF09tPFlg.png

Apartado 4
MegaDetector