Proyecto Fin de Carrera
Ingeniería de Telecomunicación

Detección de hojas en árboles mediante técnicas de aprendizaje automático

Autor: Carlos Carballo Ortiz

Tutor: Rubén Martín Clemente

Dpto. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2023

Proyecto Fin de Carrera
Ingeniería de Telecomunicación

Detección de hojas en árboles mediante técnicas de aprendizaje automático

Autor:

Carlos Carballo Ortiz

Tutor:

Rubén Martín Clemente

Profesor titular

Dpto. de Teoría de la Señal y Comunicaciones

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2023

Proyecto Fin de Carrera: Detección de hojas en árboles mediante técnicas de aprendizaje automático

Autor: Carlos Carballo Ortiz
Tutor: Rubén Martín Clemente

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2013

El secretario del Tribunal

A mi padre, mi referente.

A mis abuelos, Naci y Tano.

Y como no, a mi madre, mi impulsora, la que siempre confía, la que tira de mí y por la que hoy consigo este grado.

RESUMEN

Este trabajo está enfocado en el estudio de un campo que está en pleno auge de la Inteligencia Artificial, conocido como *Deep Learning*, y va a estar orientado en la visión por computadora. El estudio se va a enfocar en la comparación de distintas técnicas de segmentación de imágenes basadas en este campo, y para ello, se propone un caso práctico que se centra en la búsqueda de dos redes neuronales convolucionales capaces de de segmentar hojas en imágenes de plantas, mediante distintos procedimientos de segmentación. La implementación de las redes neuronales se hace mediante *Python*, con el apoyo de la librería *Keras*, tanto en el desarrollo de los modelos y la preparación y organización de los conjuntos de imágenes con los que se va a entrenar la red, como para el proceso de entrenamiento de la misma.

El primer modelo va a tratarse de una red neuronal convolucional de clasificación de imágenes. En el segundo modelo se va a aplicar la arquitectura U-Net, una red propuesta en 2015 por los investigadores de la Universidad de Friburgo Ronneberger, Fischer, y Brox. Su estructura peculiar en la familia de las redes convolucionales, basada en dos vías (*encoder-decoder*), permite segmentar y etiquetar imágenes automáticamente.

ABSTRACT

This work is focused on the study of an important field of Artificial Intelligence, known as Deep Learning, and will be oriented towards computer vision. The study will focus on the comparison of different image segmentation techniques based on this field, and for this purpose, this work proposes a practical case that focuses on the search for two convolutional neural networks capable of segmenting leaves in plant images, by means of different segmentation procedures. The implementation of the neural networks is done using Python, with the support of the Keras library, both in the development of the models and the preparation and organisation of the dataset, as well as for the training process.

The first model will be a convolutional neural network for image classification. The second one will use the U-Net architecture, a network proposed in 2015 by the University of Freiburg researchers Ronneberger, Fischer, and Brox. Its peculiar structure in the family of convolutional networks, based on two paths (encoder-decoder), allows images to be automatically segmented and labelled.

ÍNDICE

Resumen **I**
Abstract **III**
Índice **V**
Índice de Tablas **VII**
Índice de Figuras **IX**
Índice de Códigos **XI**
1 Introducción **1**
1.1 Visión por computador *1*
1.2 Segmentación *2*
1.3 Objetivos *3*
1.4 Estructura *3*
2 Fundamentos teóricos **5**
2.1. Inteligencia artificial, Machine Learning y Deep Learning *5*
2.1.1 Inteligencia artificial 5
2.1.2 Machine Learning 6
2.1.3 Deep Learning 7
2.2 Redes neuronales artificiales *7*
2.3.1 Evolución histórica 7
2.3.2 Terminología básica 8
2.3.3 El perceptrón 9
2.3.4 Redes multicapa 10
2.3.5 Funciones de activación 12
2.3.6 Entrenamiento de la red 14
2.3.7 Evaluación del modelo 19
2.4 Redes neuronales convolucionales *20*
2.3.1 Convolución 20
2.3.2 Pooling 24
2.4 Técnicas avanzadas de entrenamiento *25*
2.5.1 Data Augmentation 25
2.5.2 Dropout 25
2.5.3 Transfer Learning 25
3 Metodología **27**
3.1 Contexto *27*
3.2 Recursos empleados *27*
3.3 Preparación de los datos *29*
3.3.1 Cambio del espacio de color 29
3.3.2 Definición de umbrales 30
3.3.3 Transformaciones morfológicas 31

3.3.4 Evaluación 32
3.4 Implementación de las redes neuronales: 33
3.4.1 Primer modelo: red neuronal de clasificación 33
3.4.2 Segundo modelo: red neuronal de segmentación 40
4 Evaluación 47
4.1 Primer modelo 47
4.1.1 Parámetros iniciales 47
4.1.2 Resultados 47
4.1.3 Entrenamiento 50
4.2 Segundo modelo 51
4.2.1 Parámetros iniciales 51
4.2.2 Resultados 52
4.2.3 Entrenamiento 53
5 Conclusión 55
Referencias 57
Anexo 59

ÍNDICE DE TABLAS

Tabla 1: Hiperparámetros configurados del modelo 1 47
Tabla 2: Métricas del entrenamiento del modelo 1 51
Tabla 3: Hiperparámetros configurados del modelo 2 51
Tabla 4: Métricas del entrenamiento del modelo 2 54

ÍNDICE DE FIGURAS

Figura 1: Comparación de la entrada y la salida de un sistema clásico y un sistema Machine Learning 6
Figura 2: Campos de estudio que engloban al Deep Learning 7
Figura 3: Representación visual de una estructura de red neuronal 9
Figura 4: Perceptrón simple 9
Figura 5: Tipos de capa de una red neuronal 11
Figura 6: Problema de regresión lineal 11
Figura 7: Problema de clasificación 12
Figura 8: Representación gráfica de la función sigmoide 13
Figura 9: Transformación softmax y codificación one-hot 14
Figura 10: Representación gráfica de la función ReLU 14
Figura 11: Proceso de aprendizaje de una red neuronal 16
Figura 12: Representación gráfica del algoritmo de descenso de gradiente sobre la función de error 17
Figura 13: Representación gráfica de una función de coste tridimensional 17
Figura 14: Funcionamiento de un learning rate grande sobre la función de coste 18
Figura 15: Funcionamiento de un learning rate pequeño sobre la función de coste 19
Figura 16: Ejemplo de matriz de confusión 20
Figura 17: Representación de un píxel de una imagen en blanco y negro 21
Figura 18: Ventana o filtro aplicable en una operación de convolución 21
Figura 19: Operación de convolución, que aplica el filtro (valores en rojo) a la primera zona 22
Figura 20: Operación de convolución, que aplica el filtro (valores en rojo) a la segunda zona 22
Figura 21: Ejemplo de filtro utilizado para localizar bordes en una imagen 23
Figura 22: Operación de convolución 23
Figura 23: Operación de pooling 24
Figura 24: Operación de pooling: las ventanas de la operación de pooling no se solapan entre ellas 24
Figura 25: Comparación visual del número de cores de una CPU y una GPU 28
Figura 26: Torre de integración de Keras, impulsado por hardware CPU, GPU y TPU 28
Figura 27: Logo OpenCV 29
Figura 28: Cilindro de representación del espacio HSV 30
Figura 29: Máscara que detecta los píxeles de color verde 31
Figura 30: Operación de Opening antes y después 31
Figura 31: Imagen planta (1) 32
Figura 32: Máscara antes y después de aplicar la transformación morfológica Opening 33
Figura 33: Estructura del conjunto de datos. Modelo 1 34
Figura 34: Estructura conjunto training. Modelo 1 34
Figura 35: Resultados en subimágenes 8x8 píxeles. Modelo 1 39
Figura 36: Resultado imagen original. Modelo 1 39
Figura 37: Estructura del conjunto de datos. Modelo 2 41
Figura 38: Arquitectura U-Net 43
Figura 39: Resultado (1). Modelo 2 45
Figura 40: Imagen planta (2) 48
Figura 41: Comparación máscara original (izquierda) y máscara modelo 1 (derecha) 48
Figura 42: Imagen original y resultado máscara modelo 1 (1) 49
Figura 43: Imagen original y resultado máscara modelo 1 (2) 49

Figura 44: Tasa de acierto (izquierda) y función de coste (derecha) modelo 1 50
Figura 45: Representación de las zonas de la matriz de confusión 51
Figura 46: Imagen original (izquierda), máscara original (centro), máscara resultado modelo 2 (1) 52
Figura 47: Imagen original (izquierda), máscara original (centro), máscara resultado modelo 2 (2) 53
Figura 48: Tasa de acierto (izquierda) y función de coste (derecha) modelo 2 53

ÍNDICE DE CÓDIGOS

Código 1: Función para cambiar el espacio de color de RGB a HSV 30
Código 2: Umbrales del espacio HSV para el color verde 31
Código 3: Función para verificar los píxelesque están dentro del umbral 31
Código 4: Aplicación de operación Opening en las máscaras 32
Código 5: División de imágenes en subimágenes 8x8 que se guardan en listas según contengan hojas o no. 35
Código 6: Se coge una muestra aleatoria de 10.000 datos de validación que contengan hojas 35
Código 7: Se combinan las listas de entrenamiento y de validación 35
Código 8: Se instancian dos objetos ImageDataGenerator con sus argumentos de transformación de imágenes. Modelo 1 36
Código 9: Implementación de los generadores de lotes de datos. Modelo 1 37
Código 10: Estructura de capas. Modelo 1 37
Código 11: Hiperparámetros. Modelo 1 38
Código 12: Inicio del entrenamiento. Modelo 1 38
Código 13: Se instancian dos objetos ImageDataGenerator con sus argumentos de transformación de imágenes. Modelo 2 41
Código 14: Implementación de los generadores de lotes de datos. Modelo 2 42
Código 15: Combinación generadores. Modelo 2 42
Código 16: Estructura de capas. Modelo 2 44
Código 17: Implementación de generador de lotes de datos para testing. Modelo 2 45

1 INTRODUCCIÓN

En este trabajo, se presenta un estudio sobre el aprendizaje profundo (también *Deep Learning* en inglés), un campo de la Inteligencia Artificial que está experimentando en la última década un crecimiento exponencial. El interés por esta ciencia se va a reflejar, concretamente, en el aprendizaje profundo orientado a la visión por computadora. Se va a estudiar su ámbito teórico, la técnica algorítmica en las que se basa, conocida como **red neuronal**, y se va a profundizar en un caso práctico de un tipo de técnica concreta, que va a implicar la **implementación de dos modelos de red neuronal convolucional**, que con diferentes estructuras y procedimientos van a solucionar el objetivo común de **detección de hojas en planta**.

El propósito de este proyecto es comparar diferentes técnicas de segmentación de imágenes basadas en Inteligencia Artificial (IA). Con segmentación nos referimos al proceso de dividir una imagen en regiones de manera que permita la extracción de información específica. Se ha seleccionado ilustrar esta segmentación en hojas de arboles porque es bastante representativo, es decir, que viene de un contexto concreto y fácilmente comprensible.

Para el caso práctico, se utiliza un conjunto de datos obtenido del ámbito personal, que contiene imágenes de distintos tipos de plantas con diversas clases de hoja para que el modelo pueda generalizar sus resultados ante una gran variedad de posibilidades. De los modelos se obtiene como resultado las máscaras de las imágenes que segmenten la zona donde se encuentran las hojas, y para ello, previamente se aplica un aprendizaje supervisado en el que se entrena al modelo con las imágenes mencionadas y sus máscaras correspondientes. Estas primeras máscaras se obtendrán a partir de una aplicación implementada en *Python* que se apoya en *OpenCV*, que servirá para formar ese conjunto de datos.

El diseño, entrenamiento y evaluación de los modelos de red neuronal y la gestión del conjunto de datos se implementará con *Keras*, una librería de *Python* de alto nivel utilizada ampliamente en el campo del aprendizaje automático para redes neuronales.

1.1 Visión por computador

La visión por computador, también conocida como *visión artificial*, es una disciplina científica formada por un conjunto de técnicas que permite capturar, procesar y analizar imágenes con un ordenador, con el objetivo de extraer información útil que muestre una comprensión real de su contenido. Dicha información servirá para conseguir un fin, como la clasificación y búsqueda de imágenes o el reconocimiento facial.

Es obvio que la visión por computador ha pretendido emular siempre la capacidad sensorial del ser humano utilizando máquinas, aunque lo cierto es que todavía existen limitaciones que indican estar aún muy lejos de alcanzar ese nivel. Algunas limitaciones se presentan en la rapidez del ser humano para recoger y tratar gran cantidad de información, comprenderla, interpretar partes no visibles, reconocerla desde diferentes puntos

de vista o con iluminación cambiante. Por el contrario, aunque la máquina puede emular procesos como reconocimiento de bordes y movimientos, requiere un tiempo de proceso elevado. No obstante, el sistema humano también posee desventajas donde la visión por computador se hace necesaria, como las medidas de magnitudes físicas, como longitudes o áreas, o la buena respuesta a tareas rutinarias o que requieran gran concentración, como los procesos de control de calidad [1].

A la visión por computadora se le pueden atribuir un gran número aplicaciones pertenecientes a campos muy variados, por ejemplo:

- **Militares**: *Detección de seguimiento, análisis del terreno, armas inteligentes.*
- **Robótica:** *Guiado de robots industriales, navegación de robots móviles.*
- **Agricultura:** *Análisis de las plantaciones (crecimiento, enfermedades), análisis de imágenes tomadas por satélite.*
- **Biometría:** *Identificación automática de huellas dactilares, reconocimiento facial.*
- **Control de tráfico:** *Identificación de matrículas de vehículos, control del tráfico en la vía.*
- **Control de calidad:** *Verificación de etiquetas, inspecciones de contenedores, inspección de motores, inspección de cristales, control de calidad de comida, inspección de soldaduras, inspección de circuitos impresos, inspección de madera, tela, fundiciones, papel.*
- **Biomedicina:** *Análisis de imágenes tomadas por rayos X, análisis de imágenes tomada por ultrasonido, análisis de ADN.*
- **Seguridad:** *Vigilancia de edificios, detección de explosivos por rayos X* [2].

1.2 Segmentación

La segmentación es una tarea fundamental en la visión artificial, ya que permite extraer los objetos de la imagen para su posterior descripción y reconocimiento. Se basa en el proceso de dividir la imagen en regiones u objetos cuyos píxeles posean atributos similares, como niveles de gris, color, textura, bordes o movimiento. Dos técnicas para esta división se basan en:

- La previa identificación de los píxeles que configuran la frontera, quedando definida la forma del objeto y los píxeles que lo integran.
- Enfocar la segmentación como un problema de clasificación de píxeles o grupos de píxeles, siguiendo la norma de que los píxeles de una región deben ser similares, y los píxeles de regiones distintas no deben serlo.

La segmentación se basa en 3 propiedades:

1. Similitud: cada uno de los píxeles de un elemento tiene valores parecidos para alguna propiedad.
2. Discontinuidad: Los objetos destacan del entorno y por tanto tienen bordes definidos
3. Conectividad: Los píxeles del objeto son contiguos.

Estas condiciones no son fáciles de cumplir y llevan a posibles fallos y limitaciones de la segmentación, pues en la primera condición puede haber una existencia de brillo o presencia de ruido que enmascare la similitud, en la segunda condición, los bordes pueden no estar bien definidos, y con la tercera, puede haber un objeto que

tape la continuidad de lo que queremos segmentar.

Las técnicas, por tanto, se basan en la búsqueda de partes uniformes de la imagen y, contrariamente, de partes donde se produzca un cambio [2].

1.3 Objetivos

El propósito de este trabajo es conocer tanto la visión teórica como la práctica de las técnicas que se pueden plantear en el Deep Learning. El enfoque lo centramos en la segmentación de imágenes, y para ello se va a profundizar de manera más específica en las redes neuronales convolucionales, que son las más utilizadas para este tipo de tareas. Se van a implementar dos modelos de red neuronal, que van a abordar un mismo problema de diferente manera.

El trabajo no tiene otro fin que el de aprender a manejar las herramientas que se proponen para el diseño y desarrollo de las redes neuronales, y adquirir los primeros conocimientos de un campo de gran interés que se está convirtiendo rápidamente en una habilidad esencial en el campo tecnológico actual.

1.4 Estructura

La organización que se sigue en el proyecto es la siguiente:

- **Capítulo 1. Introducción:** Se habla sobre la visión artificial y la segmentación en la actualidad.
- **Capítulo 2. Fundamentos teóricos:** Introducción a la IA, Machine Learning y Deep Learning. A continuación, se habla de las redes neuronales, abarcando su historia, el funcionamiento del algoritmo, su estructura y conceptos clave. Por último, profundiza en las redes neuronales convolucionales.
- **Capítulo 3. Metodología:** Introduce la parte práctica del trabajo, planteando el problema que se va a llevar a cabo, la preparación de los datos, y el diseño y desarrollo de los modelos implementados.
- **Capítulo 4. Evaluación:** se prueba el funcionamiento de los modelos y se mide su eficiencia y resultados.
- **Capítulo 5: Conclusión:** Interpretación de resultados y mejoras.

2 FUNDAMENTOS TEÓRICOS

Para dar comienzo a este punto, creo que es esencial que antes de meternos en materia de Aprendizaje Profundo, sepamos distinguir entre Inteligencia Artificial, Machine Learning y Deep Learning, de donde vienen estos conceptos y la relación que tienen entre ellos.

2.1. Inteligencia artificial, Machine Learning y Deep Learning

2.1.1 Inteligencia artificial

Una primera definición simple y generalista podría ser que la inteligencia artificial se refiere a aquella rama de las ciencias de la computación que estudia el comportamiento inteligente de las máquinas en contraste con la inteligencia natural de los humanos. Por lo tanto, la IA se relaciona con el desarrollo de métodos y algoritmos que permitan automatizar tareas intelectuales que normalmente realizan los humanos, como por ejemplo el reconocimiento de colores y formas, el reconocimiento de voz o mantener una conversación.

El término "inteligencia artificial" fue acuñado en 1956 en una conferencia en la Universidad de Dartmouth organizada por John McCarthy para reunir en un taller de dos meses de duración a investigadores estadounidenses interesados en la teoría de autómatas, las redes neuronales y el estudio de la inteligencia [3].

Durante gran parte de la historia, el ser humano ha tratado de comprender y teorizar, de alguna manera, sobre nuestra forma de razonar. En los últimos tiempos, este intento de comprensión ha ido más allá, tratando de generar una inteligencia "no natural" que fuera capaz de asemejarse a las capacidades humanas e, incluso, superarlas. Este movimiento se empezó a manifestar a principios del siglo XX cuando surgieron los grandes avances científicos que permitirían ver este objetivo como una posibilidad real. Para ello, se siguieron dos principales fuentes de pensamiento:

- **Simbolismo**: se basa en la suposición de que la inteligencia de una máquina puede lograrse mediante la manipulación de sistemas simbólicos de la lógica tradicional. De aquí surgieron los primeros modelos de programación en los años 30. A través de esta metodología, era posible saber en todo momento la decisión tomada por el algoritmo, lo que ofrece una mayor confianza en la resolución de problemas. Fue muy fructífera en las primeras etapas de la IA, pero se llegó a la conclusión de que era limitante si se pretendía que las máquinas adquirieran capacidad de evolución por sí mismas.

- **Conexionismo**: se centra más en el estudio del cerebro y el misterio de sus funciones congnitivas y la compleja arquitectura de su entramado de neuronas. Se busca que la máquina tenga la capacidad de reconocer patrones en los datos a partir del procesamiento de una estructura de nodos conectados cuya distribución modela el problema que se presenta. El principal modelo son las **redes neuronales,** herramienta principal en la que se centrará el trabajo [4] [5].

2.1.2 Machine Learning

Traducido al castellano como "aprendizaje automático" (nombre que lo autodefine a la perfección), se podría catalogar como el subcampo de la Inteligencia Artificial que aporta a los ordenadores una capacidad de aprendizaje de manera automática. Funciona de manera que el programador establece unas reglas (estructura, algoritmo de entrenamiento e hiperparámetros) y una selección y ajuste de los datos de entrenamiento que, posteriormente, el ordenador particulariza y adapta, encontrando patrones relevantes de esos datos con los que realiza predicciones o clasificaciones precisas. Esto abre un nuevo paradigma en la codificación.

Mientras que en la programación clásica se introducen reglas y datos que deben procesarse según esas reglas para obtener la respuesta (paradigma de la IA simbólica que se hablaba en el punto anterior [*sección 2.1.1*]), en Machine Learning, se introducen datos y las respuestas que se esperan obtener de ellos, y la propia máquina es la que determina las reglas que debe aplicar para llegar a esas respuestas a partir de los datos recibidos. Esto es lo que se conoce como entrenamiento (un sistema de aprendizaje automático no se programa explícitamente, sino que se entrena). Luego, estas reglas pueden aplicarse a nuevos datos para obtener nuevas respuestas originales [6].

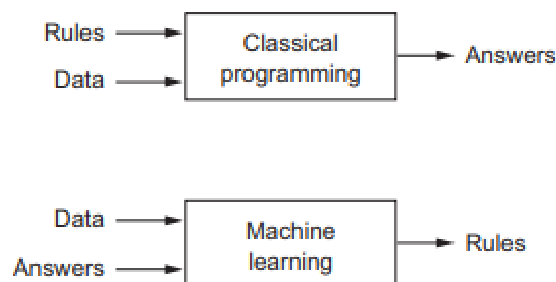


Figura 1: Comparación de la entrada y la salida de un sistema clásico y un sistema Machine Learning [1]

El área de Machine Learning comprende un amplio catálogo de complejos algoritmos de predicción que se pueden dividir en tres grandes categorías:

- *Aprendizaje supervisado*: nos referimos a este tipo de algoritmos cuando en los datos de entrada para el entrenamiento del modelo se incluyen los resultados deseados. En este tipo de algoritmos se encuentran las **redes neuronales**, los *decision trees*, la regresión lineal o la regresión logística.

- *Aprendizaje no supervisado*: en este caso, no se incluyen a los datos deseados entre los datos de entrada, sino que es el propio algoritmo el que tiene que sacar las conclusiones. Un ejemplo sería el establecimiento de perfiles de comportamiento entre varias entidades para encontrar anomalías significativas.

- *Aprendizaje por refuerzo*: el algoritmo determina las acciones con prueba y error, aprendiendo por sí mismo a través de las recompensas y penalizaciones que obtiene en sus acciones. Permite ser combinado con otros tipos, y se mueve mediante lo que se conoce como políticas (estrategia que determina para tratar de obtener la mayor recompensa en el menor tiempo) [7].

En general, todos los algoritmos de Machine Learning buscan encontrar automáticamente transformaciones que convierten los datos en representaciones más útiles (dentro de un espacio predefinido de posibilidades) para una tarea determinada. Esto permite resolver una gama extraordinariamente amplia de tareas intelectuales, desde el reconocimiento del habla hasta la conducción autónoma de vehículos.

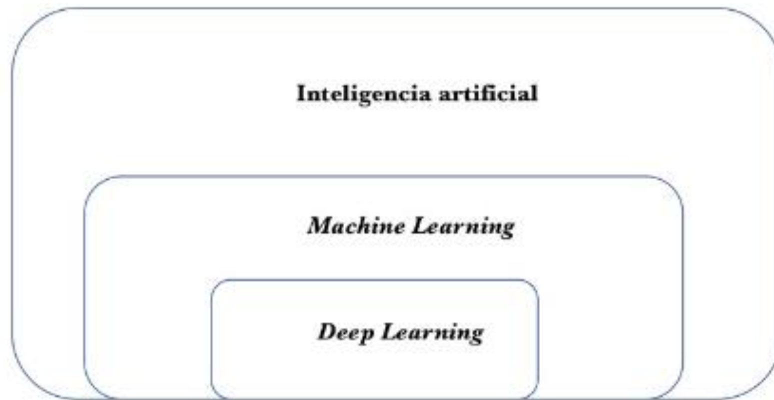


Figura 2: Campos de estudio que engloban al Deep Learning [5]

2.1.3 Deep Learning

Traducido al castellano como "aprendizaje profundo", es un subcampo específico del Machine Learning. La singularidad que lo distingue es que, a la hora de buscar que un algoritmo aprenda a reconocer patrones en los datos, el Machine Learning requiere la extracción manual de características relevantes de esos datos, mientras que el modelo de Deep Learning tiene la capacidad de aprender automáticamente a partir de los datos, lo que lo hace extremadamente poderoso en el procesamiento y análisis de información.

El fundamento del Deep Learning se basa en **la extracción progresiva de características, partiendo de las más generales a las más específicas**, que concede la capacidad al modelo de aprender en diferentes niveles de abstracción. Por ejemplo, en el reconocimiento de imágenes, las primeras capas de procesamiento de un modelo Deep Learning pueden aprender características básicas como bordes y texturas, mientras que las capas más profundas ofrecen múltiples niveles de abstracción que permiten aprender características más complejas como formas o partes de objetos. Un elemento fundamental en muchos modelos de deep learning es la **red neuronal** [8].

2.2 Redes neuronales artificiales

Las redes neuronales artificiales tratan de emular el comportamiento del cerebro humano, buscando alcanzar el aprendizaje y conocimiento a través de la experiencia y los datos. No obstante, en cuanto a la estructura, poco tienen que ver con la de las neuronas humanas.

2.3.1 Evolución histórica

Los primeros análisis del proceso de pensamiento nacen de los antiguos filósofos griegos, destacando teorías realizadas por Aristóteles (entre el 422 y el 384 a.C.) o Platón (entre el 427 y el 347 a.C.), que serían continuadas por Descartes en el siglo XVII y por los filósofos empiristas en el siglo XVIII.

La gestación de la Inteligencia Artificial comienza con el primer trabajo generalmente reconocido como IA, realizado por Warren McCulloch y Walter Pitts en 1943. Propusieron una teoría acerca de la forma de trabajar de las neuronas que pudieron modelar mediante circuitos eléctricos, y demostraron que cualquier función

podía ser computada por alguna red de neuronas conectadas, y que todas las operaciones lógicas (AND, OR, NOT) podían implementarse mediante estructuras de red simples.

Alan Turing, fue el primero en estudiar el cerebro desde el punto de vista computacional. En 1950 describió las reglas para determinar si una máquina era inteligente, a lo que se le conoce como "test de Turing". Hebb por su parte, expuso una ley explicativa del aprendizaje neuronal conocida como "regla de Hebb", que fue la antecesora de las técnicas modernas de Redes Neuronales.

Durante la década de los años 50-60, surgió el desarrollo de la red conocida como *Perceptrón,* el modelo actual más simple de red neuronal creado por Frank Rosenblatt, y que era capaz de aprender una serie de patrones y reconocerlos previamente. Esto llevó a que, en 1960, Widrow y Hoff desarrollaran el modelo ADALINE, la primera Red Neuronal Artificial que pudo ser aplicada a un problema real. Se dio a partir de una importante variación del algoritmo del Perceptrón.

A partir de la siguiente década, el interés por este ámbito se vino abajo debido a las limitaciones teóricas que presentaban los modelos neuronales, lo que hizo que los investigadores se centraran en el análisis de los sistemas simbólicos, mucho más en auge en el momento. Fue en la década de 1980 cuando resurgió ese interés con importantes eventos y congresos que dieron lugar a nuevos avances.

Fue en 1988 cuando surgió una solución muy poderosa a partir del desarrollo de Rumelhart *et al.* de un algoritmo de aprendizaje supervisado para Redes Neuronales conocido como *backpropagation*, que ofrecía una solución para la construcción de modelos más complejos.

Como consecuencia de estos avances, se ha llegado al punto de que los últimos años se vea más futuro en el paradigma conexionista que en el simbólico, y como prueba de ello tenemos las soluciones sorprendentes que nos ofrece Deep Learning en la actualidad, y que este trabajo va a tratar de abarcar en la medida de lo posible, a través del estudio teórico-práctico de las redes neuronales [9].

2.3.2 Terminología básica

Este punto es clave para dar un primer punto de vista de los conceptos básicos antes de meternos por completo en la metodología de las redes neuronales.

El Machine Learning consiste en la implementación de un modelo que sea capaz de asignar variables de entrada a etiquetas. Con etiquetas (también se denomina con el término en inglés ***labels***) nos referimos al resultado que estamos intentando que el modelo proporcione a partir de esa variable de entrada que se le introduce. Para conseguir que un modelo defina esa relación, necesitamos ajustar sus **pesos**, que son los parámetros necesarios que definen la relación de las variables de entrada con las *labels*. Para ello, el modelo pasa por dos etapas diferenciadas:

- **Etapa de entrenamiento**: se conoce como la fase de aprendizaje, donde se le muestra al modelo ejemplos de entrada que están etiquetados. El objetivo es que el modelo ajuste sus pesos a partir de las relaciones entre las variables de entrada y sus *labels*.
- **Etapa de inferencia**: se refiere al proceso de realizar **predicciones** con el modelo ya entrenado, ya sea para validar que el modelo ha entrenado correctamente, o para utilizarlo en una aplicación real. Por lo tanto, en esta fase los pesos ya están definidos por la etapa anterior y las entradas que se introducen son variables no etiquetadas.

Para el caso de Deep Learning, también se aplican estos fundamentos en las ***Redes Neuronales Artificiales***. El elemento central de las redes neuronales es la **capa** (*layer*), un módulo de procesamiento que puede

considerarse un filtro de datos. Más concretamente, las capas se encargan de procesar y transformar la información a medida que se propaga a través de la red. En la fase de entrenamiento, cada capa toma las señales de entrada, realiza cálculos y transformaciones (lineales y no lineales) en base a sus parámetros y reglas, y produce salidas que se transmiten a la siguiente capa. Las redes neuronales, al ser entrenadas mediante un aprendizaje supervisado, van a incluir en los datos de entrenamiento las respuestas objetivo que se buscan predecir. Finalmente, de la fase de aprendizaje obtendremos los parámetros (pesos) que van a conseguir que los resultados sean óptimos y que la salida sea muy parecida a la esperada [10].

Las capas se encadenan mediante **neuronas**, y cada capa contiene un número de neuronas que la conforman. Las neuronas son los elementos computacionales que comunican una capa con la siguiente interconectándose entre ellas de diferentes maneras.

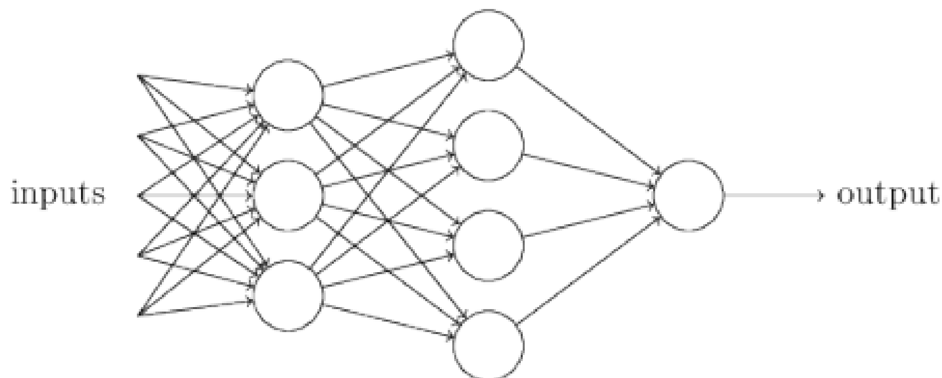


Figura 3: Representación visual de una estructura de red neuronal, donde las neuronas se representan con círculos y cada capa como una columna de neuronas [9]

Hoy en día se maneja una gran cantidad de capas que da mucha profundidad a la red, y cada capa contienen un número de neuronas, donde cada una calcula sus pesos que realizan una transformación simple, y en la unión de todas esas transformaciones que se pasan de unas a otras, el algoritmo obtiene la capacidad de reconocer patrones complejos [11].

2.3.3 El perceptrón

Como mencionamos anteriormente [*sección 2.3.1*], el modelo de Perceptrón simple fue desarrollado por Frank Rosenblatt en la década de los 50 (inspirado en trabajos anteriores de Warren McCulloch y Walter Pitts) y representa el modelo más simple de red neuronal. Representa la unidad principal de las redes neuronales artificiales, aunque en la actualidad moderna y en este trabajo el modelo de neurona más comunmente usado es la neurona *sigmoid* (se verá más adelante [*sección 2.3.5*]).

¿Cómo funcionan los perceptrones? Recibe como entrada un número de inputs binarios (x_1, x_2, ...) y produce un único output binario.

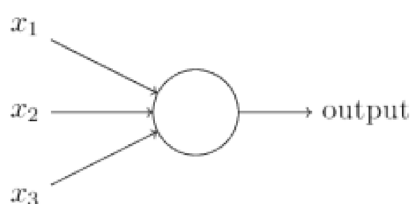


Figura 4: Perceptrón simple

Rosenblatt propuso una regla simple para calcular la salida: introdujo pesos (w_1, w_2, …) representados por números reales que expresaran la importancia de los *inputs* para el *output*. Por lo tanto, el resultado del *output* (0 ó 1) se obtiene de la influencia de los pesos en cada input mediante la suma ponderada de:

$$\sum_{i=1}^{N} w_i \, x_i$$

El resultado de esa suma ponderada se compara con un umbral predefinido, y dependiendo si es mayor o menor obtendrá una salida de 1 ó 0. Al igual que los pesos, ese umbral será también un número real:

$$output = \begin{cases} 0 \quad if \quad \sum_{i=1}^{N} w_i \, x_i \leq umbral \\ 1 \quad if \quad \sum_{i=1}^{N} w_i \, x_i > umbral \end{cases}$$

Para simplificar esta expresión, vamos a definir la salida del perceptrón como una función f que relaciona un vector X de variables independientes de *inputs* con la variable dependiente *y = f(x).* Parametrizamos la función con el vector de pesos W que ponderan la influencia de cada uno de los *inputs*, y nos queda un sumatorio que se convierte en un producto escalar:

$$W \cdot X = \sum_{i=1}^{N} w_i \, x_i$$

La segunda simplificación que hacemos es pasar el umbral al otro lado de la desigualdad, y este valor va a reconocerse como sesgo. Usando el sesgo en vez del umbral, la regla se reescribe de la siguiente manera:

$$output = \begin{cases} 0 \quad if \quad W \cdot X + b \leq 0 \\ 1 \quad if \quad W \cdot X + b > 0 \end{cases}$$

Para un perceptrón con un sesgo muy grande, es extremadamente fácil que el perceptrón emita un 1, y viceversa [12].

Con este modelo nos referimos a un modelo con una única capa de una neurona, pero hoy en día lo normal es que encontremos redes de numerosas capas, en las que cada capa contiene numerosas neuronas que se comunican con las de la capa anterior para recibir su información y con las neuronas de la siguiente capa para enviar su salida.

2.3.4 Redes multicapa

La estructura de capas de una red neuronal se compone de una capa de entrada (*input layer*) donde se introducen los datos, una de salida (*output layer*) que devuelve el valor esperado (en este caso contiene una única neurona, pero puede contener varias) y las capas ocultas (*hidden layers*) que son las que están situadas entre la capa de entrada y la de salida, y en cada modelo pueden variar en número y también en la composición de neuronas.

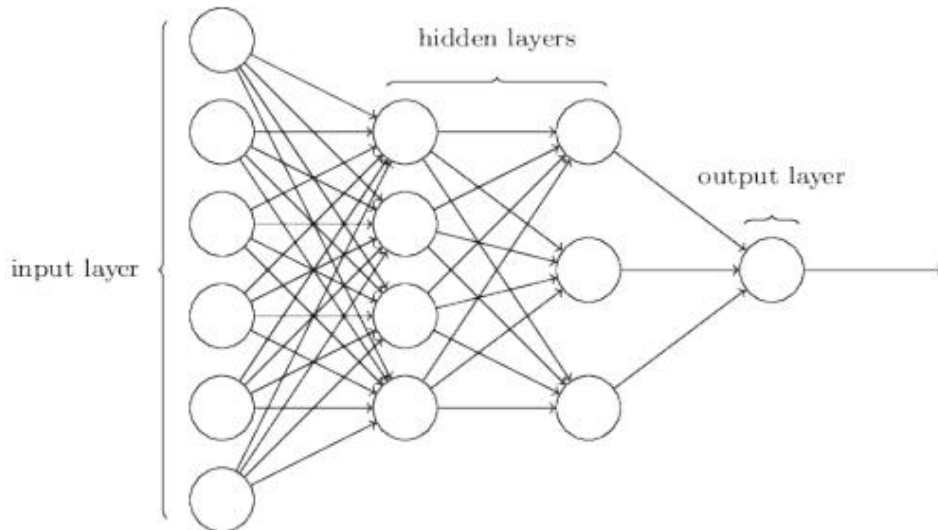


Figura 5: Tipos de capa de una red neuronal [9]

Estos diseños se usan para algoritmos de regresión y clasificación. Se podría decir que los algoritmos de regresión son métodos que tratan de establecer una relación matemática entre un conjunto de variables independientes (las entradas o variables predictoras) y la variable objetivo que se quiere predecir. Interesan dos tipos que son diferenciados por la salida del modelo:

- Regresión lineal: cuando la salida sea continua. Un ejemplo muy simple sería una suposición en la que se quiere predecir el precio de una vivienda basándose en los metros cuadrados, a partir de un conjunto de datos que establecen el tamaño de diferentes viviendas y el precio al que fueron vendidas. Utilizando estos datos en una gráfica de dispersión, podemos ajustar una línea recta que represente la relación tamaño-precio. Esta línea se ajusta a los puntos de manera que minimiza la distancia entre los puntos de los datos y los puntos predichos por el modelo. De esta manera, se hace fácil la obtención de una ecuación de predicción a partir de una línea recta.

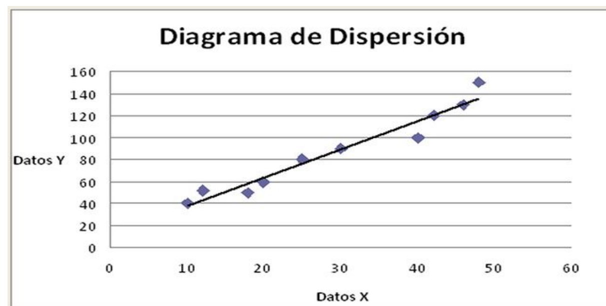


Figura 6: Problema de regresión lineal [13]

- Regresión logística (o clasificación): cuando la salida sea discreta. Orientado a la identificación de clases de las características de entrada, medido mediante probabilidad de pertenecer a una clase u otra. El problema más simple en el que se da es en un conjunto de puntos en un plano de dos dimensiones, y los puntos se clasifican en dos tipos. El algoritmo tiene que resolver como separar ambos tipos con la línea recta que mejor se ajuste. La adición de neuronas podría permitir que esa línea recta pueda ajustarse con curvas para poder obtener una mejor clasificación de los tipos.

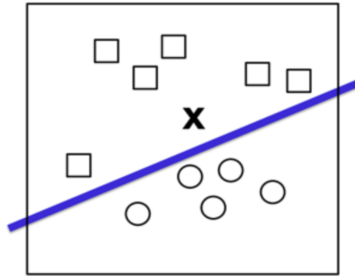


Figura 7: Problema de clasificación [14]

Al ser algoritmos de aprendizaje supervisado, utilizan una métrica de error conocida como **función de pérdida** o ***loss***, que cuantifica el nivel de fallo que existe entre los valores que el modelo predice y los resultados esperados. Es decir, en un contexto de entrenamiento de un modelo, la *loss* nos ayuda a identificar cómo de mala es la predicción y como evoluciona a lo largo del proceso de entrenamiento, sirviendo también de indicativo de evolución del modelo.

Volviendo a la arquitectura de la red neuronal, el diseño de la capa de entrada y la de salida suele ser sencillo. Por ejemplo, si queremos introducir imágenes de dimensión 64 x 64 (en escala de grises), tendríamos que meter en la capa de entrada 4096 = 64x64 neuronas, con el valor de la intensidad de gris de cada pixel (entre 0 y 1). Para la salida, si queremos afrontar un problema de clasificación binaria, con una neurona de salida bastará para que proporcione un 0 o un 1 como resultado.

A estos modelos de capas se les conoce históricamente como *multilayer perceptrons* o MLPs, pero algunos libros lo reconocen como una denominación errónea debido a que el tipo de neuronas que se usan en estos modelos no son perceptrones. Esta apreciación se explica en el hecho de que utilizamos neuronas que son capaces de hacer cáculos no lineales gracias a las **funciones de activación** aplicadas en la salida de cada neurona. Por tanto, **permite al modelo aprender relaciones no lineales** entre las variables de entrada y salida, y de esta manera, da una mayor eficiencia para la resolución de problemas más complejos [12] [14].

2.3.5 Funciones de activación

Las funciones de activación son elementos indispensables para garantizar la no linealidad en las redes neuronales. Esto es fundamental para poder modelar relaciones y patrones complejos en los datos, ya que las relaciones en muchos conjuntos de datos del mundo real son intrínsecamente no lineales. Algunos ejemplos de situaciones donde se administra este modelo no lineal son el reconocimiento de objetos en imágenes, el procesamiento del lenguaje natural o la detección de anomalías en datos.

Existen distintas funciones de activación y vamos a describir las más esenciales:

- **Función sigmoide**: retorna un valor real en el rango entre 0 y 1 para cualquier entrada. Su fórmula se representa como:

$$y = \frac{1}{1 + e^{-z}}$$

Que gráficamente se representa como:

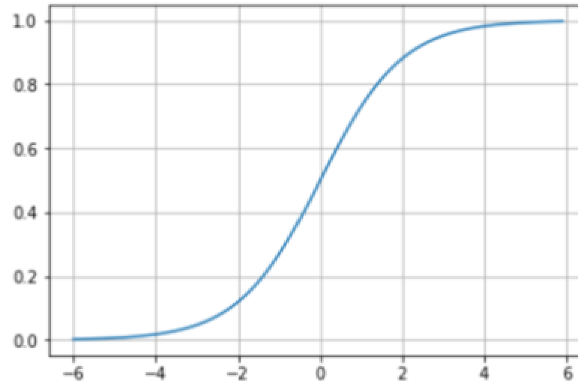


Figura 8: Representación gráfica de la función sigmoide [14]

Si observamos la gráfica, vemos que siempre tiende a dar valores cercanos al 0 y 1. Viendo la fórmula, se representa la entrada con el valor *z* y la salida que buscamos es la *y*. En el caso de que *z* tuviera un valor grande y positivo, el cálculo de la exponencial elevado a un valor grande negativo sería próximo a 0, y por tanto quedaría que la salida es igual a 1. Al contrario, si *z* << 0, $e^{z \gg 0}$ que sería igual a un número grande positivo, lo que daría como valor de salida un número próximo a 0.

$$output = \begin{cases} 0 & if \quad z \ll 0 \\ 1 & if \quad z \gg 0 \end{cases}$$

Este tipo de función es útil en problemas de clasificación binaria. Más adelante se verá la aplicación de esta función en nuestra red neuronal [14].

- **Función *softmax***: una ventaja de las redes neuronales es que permite el uso de más de una neurona de salida. Esto es útil para la abordar problemas de clasificación multiclase. Por ejemplo, una red neuronal con 10 neuronas en la capa de salida permitiría que cada neurona se encargara de detectar si la entrada pertenece a la clase que representa. Lo ideal sería que, para cada entrada, solamente se activara una neurona indicando claramente que pertenece a esa clase, pero nada garantiza que dos neuronas o más se activen a la vez. Para eso utilizamos la función de activación *softmax* en la capa de salida.

  Se podría decir que es una función extendida de la sigmoide, que convierte las salidas en un vector al que le aplica un valor exponencial a cada evidencia calculada y las normaliza de modo que sumen 1, formando una distribución de probabilidad.

$$softmax_i = \frac{e^{evidencia_i}}{\sum_{j=1}^{N} e^{evidencia_j}}$$

  El efecto que se consigue es que, mediante las exponenciales, se obtiene un efecto multiplicador cuando mayor sea la unidad, o el efecto inverso con una unidad menor. No obstante, en el caso de que haya una predicción débil porque los valores de probabilidad sean cercanos en varias clases, aún hace falta obtener la clasificación final. Esto se consigue utilizando la codificación *one-hot*, que asigna el valor 1 al componente del vector que tiene la mayor probabilidad, y al resto de elementos el valor 0.

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

Figura 9: Transformación softmax y codificación one-hot [15]

Un problema más complejo que se podría abordar con esta función de activación podría consistir en tomar una imagen en tonos grises, y que como salida se generase otra imagen que transformara la imagen de entrada en una imagen a color. La imagen se representa como un vector de píxeles, y en la capa de salida de la red neuronal se usarían tantas neuronas como píxeles se quiera que tenga la imagen en la salida. Si la entrada fuera una imagen en tonos grises de N píxeles, para colorear la imagen, haría falta una capa de salida con 3N neuronas para representar la imagen con 3 canales (rojo, verde, azul) y con el mismo tamaño de la imagen de entrada. Así, para cada diferente tono de grises, el algoritmo aplicaría un color concreto de la gama RGB [15].

Esta aplicación de algoritmos presenta ciertos problemas y limitaciones. Una que abordaremos más adelante es la necesidad de convertir los datos de entrada y salida en una representación vectorial, descomponiendo las dimensiones espaciales de los datos. Por ejemplo, una imagen 28x28, convertirla a una vector de dimensión 784, donde las primeras 28 entradas serían los 28 píxeles de la primera fila, y así sucesivamente.

- **Función *ReLU***: las siglas se refieren a Rectified Linear Unit, y consiste en una transformación que activa un solo nodo si la entrada está por encima de cierto umbral. En una gráfica se representa como:

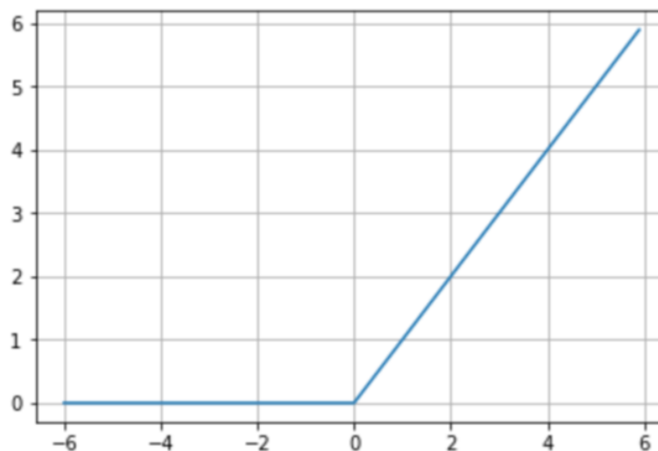


Figura 10: Representación gráfica de la función ReLU [16]

Como se puede apreciar, su comportamiento se basa en que mientras la entrada tenga un valor por debajo de 0, la salida no se activa, pero cuando la entrada es positiva, la salida obtiene una relación lineal con la entrada de $f(x) = x$, es decir, retorna el mismo valor de la entrada.

Se ha convertido en la función más empleada para la activación de las capas ocultas.

2.3.6 Entrenamiento de la red

En un nivel más práctico, nos centramos en cómo implementar la red neuronal profundizando en conceptos

como el conjunto de datos o *dataset* con el que se va a entrenar la red, la estructura de sus capas, las métricas que se deben ajustar para el entrenamiento y posibles errores en los resultados.

2.3.6.1 Conjunto de datos

El conjunto de datos, también conocido como *dataset*, es una colección estructurada de datos, ejemplos o registros que se utilizan para entrenar una red neuronal con el fin de que aprenda a resolver el problema de clasificación, predicción o reconocimiento que se plantea. El *dataset* proporciona al algoritmo las características e información necesarias para identificar patrones y relaciones en los datos. Por ello, es importante la buena selección de los datos en cuanto a tamaño, calidad y lo que representan para obtener unos resultados óptimos para el problema planteado.

Para la configuración y evaluación correcta del modelo, se suelen dividir los datos en tres conjuntos:

- *Training*: datos de entrenamiento, los que se usan para que el algoritmo "aprenda" mediante la obtención de los patrones y relaciones de características. Estos datos van a tratar de ajustarse al modelo para que las neuronas configures sus pesos.
- *Validation*: porción que se obtiene del *dataset* para controlar la evolución de la fase de entrenamiento y ajustar los hiperparámetros hasta obtener unos resultados de validación que consideremos correctos.
- *Testing*: es una porción de datos que se reserva para cuando hemos considerado que el modelo está ajustado y ya no modificaremos más sus hiperparámetros. Evalúa el rendimiento y resultado enfrentándose a nuevos datos [17].

2.3.6.2 Proceso de aprendizaje

En el proceso de entrenamiento de la red, donde busca que la red aprenda los valores de los pesos y sesgos de cada neurona, se genera un curso iterativo de ida y vuelta sobre la estructura de capas de la red. Al proceso de ida se le conoce como ***forwardpropagation*** y al de vuelta como ***backpropagation***.

- La fase de *forwardpropagation* sucede cuando se introducen los datos a la red y estos son procesados en cada capa hasta la capa de salida. Es el momento donde las neuronas hacen sus cálculos y pasan los datos a la capa siguiente.
- La fase de *backpropagation* se produce justo después de haber calculado la función de pérdidas (*loss)* tras finalizar la fase de *forwardpropagation.* El proceso consiste en ir propagando la información de la *loss* por todas las neuronas de la capa oculta partiendo de la capa de salida hacia atrás. Las neuronas reciben una fracción de la señal total de la *loss,* basándose en la aportación relativa que haya tenido cada neurona en los resultados de la fase anterior. Esto se repite por cada capa de manera que cada neurona tenga la información de *loss* que describe su contribución de la *loss* total.

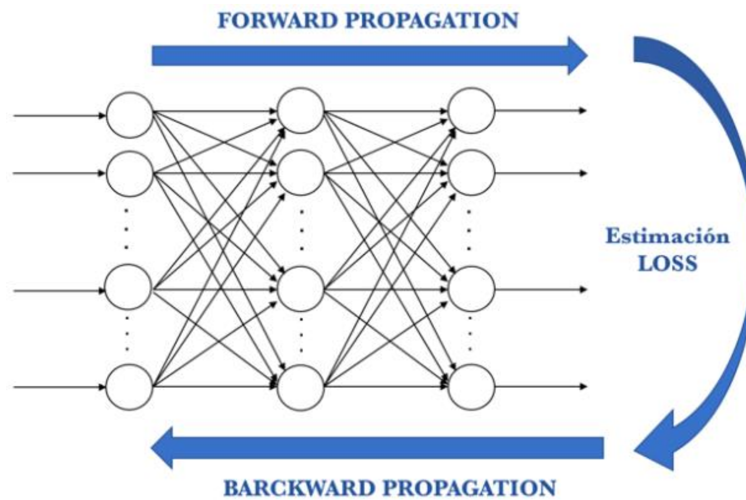


Figura 11: Proceso de aprendizaje de una red neuronal [16]

Una vez se cumple todo este proceso de ida y vuelta, se ajustan los pesos en las neuronas con el objetivo de reducir la *loss* lo máximo posible, y este ajuste de pesos se consigue con una técnica llamada **descenso del gradiente** y la abarcaremos a continuación [16].

2.3.6.3 Elementos del entrenamiento

Ya se explicó anteriormente [*sección 2.3.5*] un elemento importante como es la **función de activación**, y se vieron algunos tipos y su funcionamiento. Otro que también se ha mencionado anteriormente es la **función de pérdidas (*loss*)**, que es la que cuantifica la cercanía del modelo de los resultados que se esperan en el proceso de entrenamiento (recordemos que, al ser aprendizaje supervisado, los datos de entrada incorporan las etiquetas, que son los resultados que esperamos obtener, y con eso se calcula el grado de similitud con lo que el modelo predice).

Un tercer elemento importante es el **optimizador**. Si recordamos el proceso antes mencionado en el que las neuronas ajustan sus parámetros buscando reducir la *loss,* los optimizadores son los algoritmos encargados de ajustar esos pesos.

El tipo de optimizador más básico y que marca el punto de partida de muchos otros, es el antes mencionado **descenso del gradiente.** Este optimizador parte de una función de pérdida como la que se puede ver en la siguiente imagen. Partiendo de un punto inicial marcado por unos pesos inicializados, se busca ajustarlos con el objetivo minimizar esa *loss* (lo ideal sería llegar al punto mínimo de la gráfica, que es el más cercano a 0 que podemos obtener), y para ello se calcula el gradiente, que es el vector de derivadas parciales de la función de pérdida con respecto a un punto (parámetro) y nos indica la dirección donde la pendiente asciende en ese punto (en este caso, el punto inicial). Si la función de pérdida involucra múltiples parámetros, se calcularán las derivadas parciales con respecto a cada uno de ellos para obtener el gradiente completo. Como lo que se busca es descender, se utiliza la dirección contraria, de ahí el nombre de gradiente descendente. Utilizando la información proporcionada del gradiente, el algoritmo de este optimizador calcula nuevos pesos de manera que disminuyan la función de pérdida. La magnitud del cambio de peso (todo lo que recorre en la gráfica el cambio de un punto a otro) se ve reflejado en un hiperparámetro que luego abarcaremos, el *learning rate*. Este proceso se sigue repitiendo tras cada *epoch* hasta minimizar el error todo lo posible, como se ve representado en la imagen con cada nuevo punto que se actualiza en la gráfica en dirección del gradiente descendente [16].

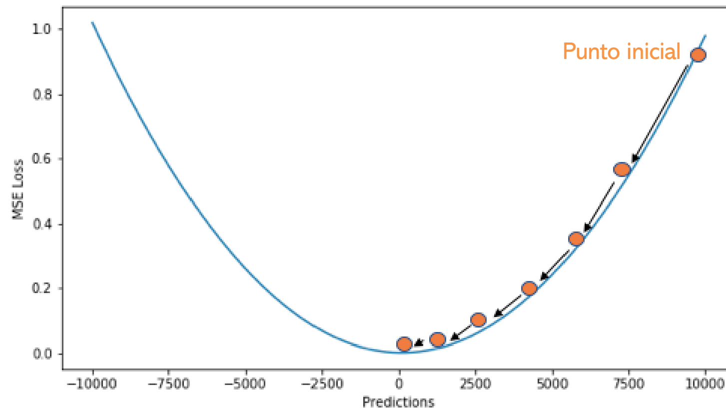


Figura 12: Representación gráfica del algoritmo de descenso de gradiente sobre la función de error [17]

Este es un ejemplo [*Figura 12*] en el que se tiene en cuenta un único peso para tener una idea clara de como funciona el algoritmo del optimizador, pero se vuelve mucho más complejo cuando se tiene en cuenta que la función *loss* es más compleja con distintos mínimos relativos y también se añaden todos los pesos de las neuronas en función de su contribución al resultado final. Por ejemplo, si tuviéramos en cuenta dos parámetros de una neurona, ya la función del error se representaría en una gráfica en 3D y el gradiente se calcularía a partir de los gradientes (primera derivada de la función) de cada parámetro.

$$\begin{bmatrix} \frac{\partial loss}{\partial \theta_1} \\ \frac{\partial loss}{\partial \theta_2} \end{bmatrix} = \vec{\nabla} f$$

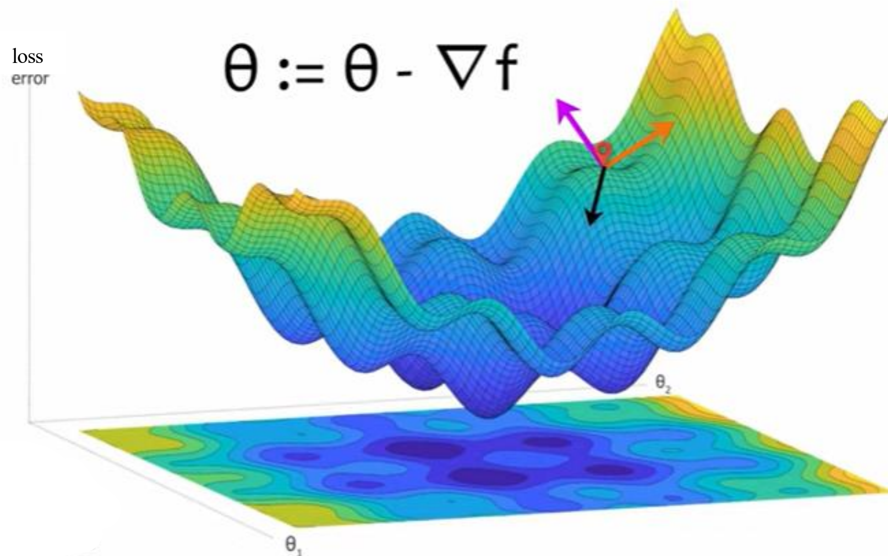


Figura 13: Representación gráfica de una función de coste tridimensional: se aplica el algoritmo del gradiente descendente. Primero, se evalúa la pendiente desde una posición inicial calculando sus derivadas parciales. Cada uno de esos valores indicará la pendiente en el eje de dicho parámetro. Todas las derivadas parciales conforman un vector ascendente que se denomina gradiente, el cual usamos para tomar el sentido opuesto, y actualizar los nuevos parámetros en esa dirección. [18]

En esta situación [*Figura 13*] ya visualizamos una función de error más compleja, con mínimos relativos que pueden inducir a una limitación del rendimiento del modelo, haciendo creer que estamos ante el mínimo

absoluto.

2.3.6.4 Parámetros e hiperparámetros

Nos referimos a parámetros del modelo cuando son variables de configuración interna que el modelo estima a partir de los datos que procesa, como por ejemplo los pesos y sesgos que las neuronas configuran. En cambio, los hiperparámetros son las variables externas que el programador especifica y que el modelo no puede ajustar a partir de los datos. Para llegar a los valores óptimos se requiere cierta experiencia e intuición del programador. Los hiperparámetros que debemos tener en cuenta para este proyecto son:

- ***Epoch* (época)**: entero que indica el número de veces en las que todos los datos del conjunto de *training* van a ser procesados por el modelo en el entrenamiento. Dependiendo del tamaño del conjunto de datos y de la complejidad del modelo, este hiperparámetro debe ajustarse para obtener un equilibrio entre el rendimiento del modelo y el tiempo de entrenamiento.

- ***Batch size***: es el tamaño de los lotes en los que se dividen los datos. Esto se hace para evitar alimentar a la red con todo el conjunto de datos a la vez y saturar su capacidad de memoria y computación. Hasta que no se procese el número de *batch* definido, no se cumple una *epoch*.

  Un tamaño de *batch* demasiado pequeño permite al modelo actualizar sus pesos con mayor frecuencia, lo que le lleva a un mejor rendimiento, aunque puede requerir más memoria (almacena los cáculos del proceso) y un mayor tiempo de entrenamiento. Por otro lado, un tamaño de *batch* demasiado grande, acelera el tiempo, pero puede hacer que su rendimiento sea malo por sobreajuste del modelo, un fenómeno conocido como ***overfitting*** (proceso que ocurre cuando los datos no son muy variables y el modelo se ajusta demasiado a los datos de entrenamiento, teniendo dificultades para generalizar).

- ***Learning rate:*** es un escalar que actúa en el optimizador, multiplicando al gradiente, y determina la magnitud del avance en la gráfica del error, es decir, cómo de rápido o lento se realizan los ajustes en los parámetros del modelo. Un *learning rate* demasiado grande puede llevar a actualizaciones muy bruscas que no consiguen ajustarse al mínimo, como se muestra en el siguiente ejemplo.

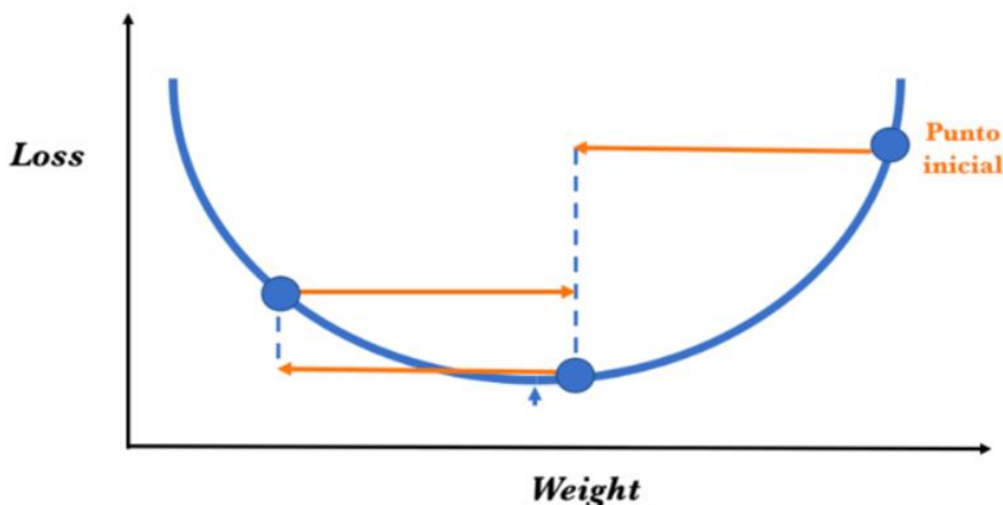


Figura 14: Funcionamiento de un learning rate grande sobre la función de coste [16]

En cambio, un *learning rate* más pequeño necesitará más *epochs* de entrenamiento, pero ajustará mejor sus pesos y da más seguridad a la hora de minimizar el error. Un ejemplo lo podemos ver en la siguiente gráfica [16].

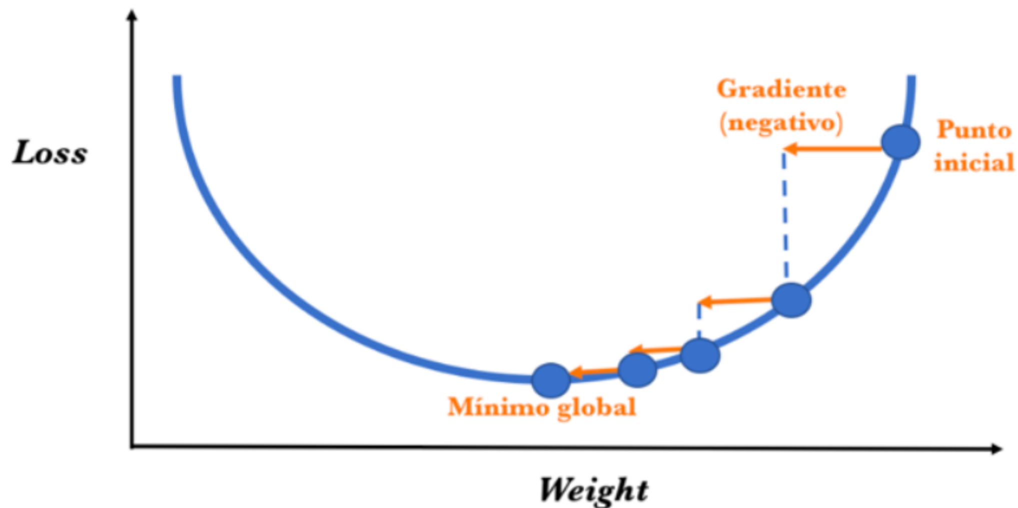


Figura 15: Funcionamiento de un learning rate pequeño sobre la función de coste [16]

2.3.7 Evaluación del modelo

Una vez el modelo ha sido entrenado, existen ciertas métricas que ayudan a evaluar cómo de bien o mal se comporta el modelo ante datos nuevos. Existe la llamada matriz de confusión que, básicamente, y sin profundizar mucho en el tema, contabiliza las predicciones en distintos grupos:

- Verdaderos positivos: número de predicciones clasificadas correctamente como positivas.
- Verdaderos negativos: número de predicciones clasificadas correctamente como negativas.
- Falsos positivos: número de predicciones clasificadas erróneamente como positivas.
- Falsos negativos: número de predicciones clasificadas erróneamente como negativas.

Cuando se habla de predicciones positivas, se refiere a esas predicciones que son asignadas a una categoría de interés para el problema que se plantea, mientras que, las predicciones negativas son aquellas que pertenecen a una clase que no es objetivo de lo que se evalúa.

La matriz de confusión se usa para evaluar tareas de clasificación y también en segmentación de imágenes. En función de sus valores, se pueden calcular algunas métricas de evaluación del entrenamiento del modelo. La primera, es la métrica de **exactitud (*accuracy*)**.

$$Accuracy = \frac{VP + VN}{VP + FP + VN + FN}$$

En principio, es una herramienta que indica la proporción de las predicciones correctas frente al total de predicciones, sin embargo, hay que tener en cuenta alguna desventaja. La *accuracy* no distingue entre el coste del falso positivo y falso negativo. Un falso positivo puede no tener importancia en el resultado final, (por ejemplo, una seta que sea comestible pero que se clasifique como venenosa), pero un falso negativo si (una seta que sea venenosa y se clasifique como comestible). Por ello, existe una métrica que indica cómo de bien evita los falsos negativos el modelo, y se conoce como ***Sensitivity***.

$$Sensitivity = \frac{Verdaderos\ positivos}{(Verdaderos\ positivos + Falsos\ negativos)}$$

También existe la métrica de **precisión**, que mide la tasa de clasificados como positivos que realmente lo son :

$$Precision = \frac{Verdaderos\ positivos}{(Verdaderos\ positivos + Falsos\ positivos)}$$

A partir de la matriz de confusión del modelo, se pueden obtener diversas métricas que focalicen en situaciones concretas. Un ejemplo básico es la evaluación de la calidad del modelo, donde la matriz muestra en un modelo de clasificación donde hay 10 tipos de clases, el número predicciones en las que el modelo coincide con el valor de la etiqueta, que se muestra en la diagonal [*Figura 16*].

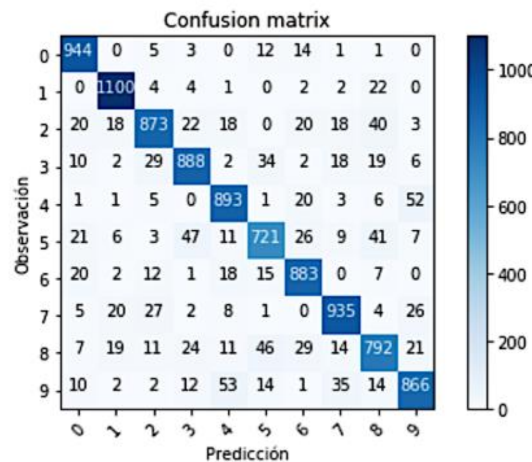


Figura 16: Ejemplo de matriz de confusión [17]

Los otros valores muestran el número de veces que ha sido clasificado incorrectamente con esa clase habiendo sido otra. Por ejemplo, el modelo ha clasificado 5 veces en la clase 2 cuando realmente el dato pertenecía a la clase 0.

2.4 Redes neuronales convolucionales

Las redes neuronales convolucionales, también conocidas como CNN (Convolutional Neural Networks), son el tipo de red neuronal que se implementa en este trabajo. Son redes especializadas en tareas de visión por computador para el reconocimiento y segmentación de imágenes. Lo que las hace especiales es el aprendizaje de características, ya que cada capa va aprendiendo diferentes niveles de abstracción, lo que se traduce a que las primeras capas ocultas se centran en características de los datos más generales y a medida que el procesamiento de datos pasa a las siguientes capas, se van enfocando en la identificación de estructuras cada vez más complejas y específicas.

Existen dos tipos de capas ocultas que son los componentes clave de su estructura, y estas capas se componen de un grupo de neuronas especializadas en dos operaciones: convolución y *pooling*.

2.3.1 Convolución

La operación de convolución ofrece una ventaja muy interesante al algoritmo, que consiste en que una vez la capa aprende una característica de en un punto concreto de la imagen, esa característica pasa a ser reconocida en cualquier parte de la misma. Esto es la diferencia clave entre las redes convolucionales y las redes

densamente conectadas, las primeras son capaces de detectar patrones en diferentes ubicaciones de la imagen, sin necesidad de volver a aprenderlos en cada ubicación, mientras que las segundas, si un patrón se aprende en una zona de la imagen, el patrón solo se va a reconocer en esa zona y no en otra ubicación, al menos hasta que se vuelva a aprender ese patrón en esa ubicación diferente. Esta ventaja de las redes convolucionales se obtiene a partir de la aplicación de pequeños filtros en cada capa, que son los encargados de detectar las características recorriendo todas las regiones de la imagen. Estas características se combinan y reagrupan a medida que se propagan a las siguientes capas que, con la aplicación de nuevos filtros, capturarán características cada vez más abstractas y de mayor nivel. Este último proceso hace referencia a la segunda ventaja de las redes convolucionales, que posibilita el aprendizaje de patrones de manera jerárquica, reconociendo primero elementos más simples y, a medida que se pasa a las siguientes capas, va aprendiendo elementos más complejos y abstractos. Esto proporciona un aprendizaje eficiente de la red.

Para que una red neuronal trabaje con una imagen, primero se separa la imagen en cada uno de sus píxeles y cada píxel contendrá un valor. Si se trabaja con imágenes en blanco y negro, el valor irá desde 0 para negro hasta 255 para blanco. Para que la red pueda reconocer características en la imagen, no es suficiente revisando un único píxel, pero si se observan los pixeles de alrededor sí que se puede detectar cambios bruscos en los valores de píxeles adyacentes, lo que marca una característica de un borde o un eje de algún objeto, como se aprecia en la imagen [*Figura 17*].

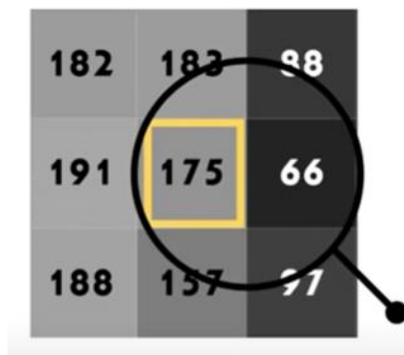


Figura 17: Representación de un píxel de una imagen en blanco y negro y de los píxeles de su alrededor, que muestra el cambio drástico entre la segunda y la tercera columna dando lugar a un posible borde [20]

La operación de convolución toma importancia en esta tarea, ya que va recorriendo los píxeles de una imagen, y revisando los de alrededor, por lo que acaba centrándose en pequeñas zonas localizadas de la imagen que no son más que matrices de píxeles. Si vamos recorriendo cada píxel y los de su alrededor, podemos compararlo con una ventana de cierta dimensión fija (en este caso 3x3, 9 píxeles) que va recorriendo toda la imagen de entrada.

Visualmente, la ventana parte de la esquina superior izquierda de la imagen y se va deslizando una posición de un píxel a la derecha. Esta ventana se conoce como el filtro (o *kernel*), y es una matriz que lleva asignados unos valores en cada casilla [*Figura 18*].

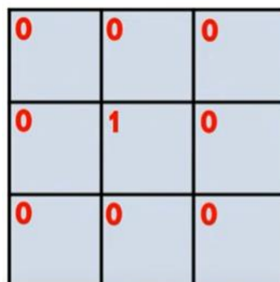


Figura 18: Ventana o filtro aplicable en una operación de convolución [20]

Cada vez que se aplica el filtro a una zona de la imagen, se calcula el producto escalar, es decir, se multiplica

cada casilla por el valor del píxel y se suma todo, y el resultado constituye una zona de la nueva imagen. Por ejemplo, si aplicamos el filtro a la primera posición de la imagen, el resultado se añadirá a la primera posición de la nueva imagen [*Figura 19*].

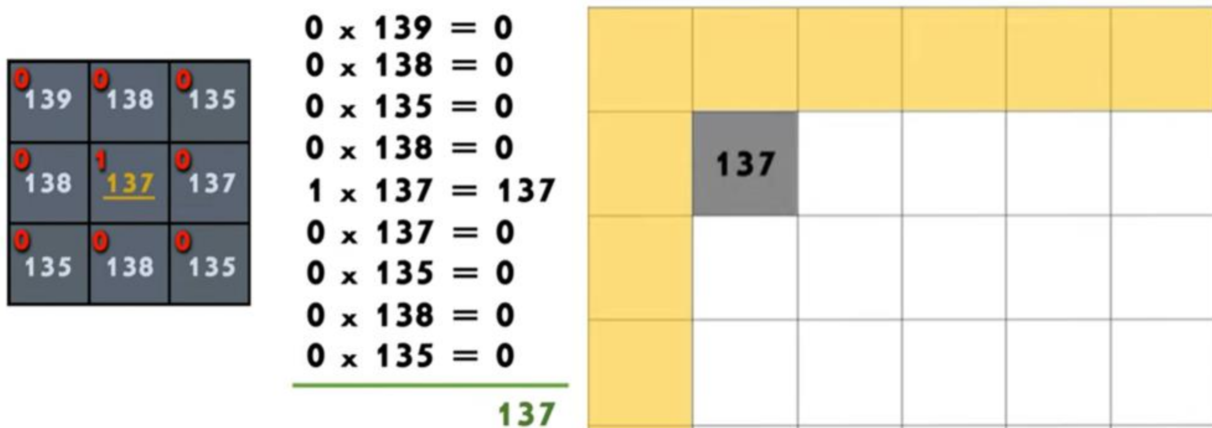


Figura 19: Operación de convolución, que aplica el filtro (valores en rojo) a la primera zona de la imagen, y el resultado del producto escalar se posiciona en la zona correspondiente (primer píxel) de la nueva imagen [20]

Al avanzar una posición, se vuelve a aplicar el filtro y se vuelve a guardar el resultado en la nueva imagen, y así sucesivamente[*Figura 20*].

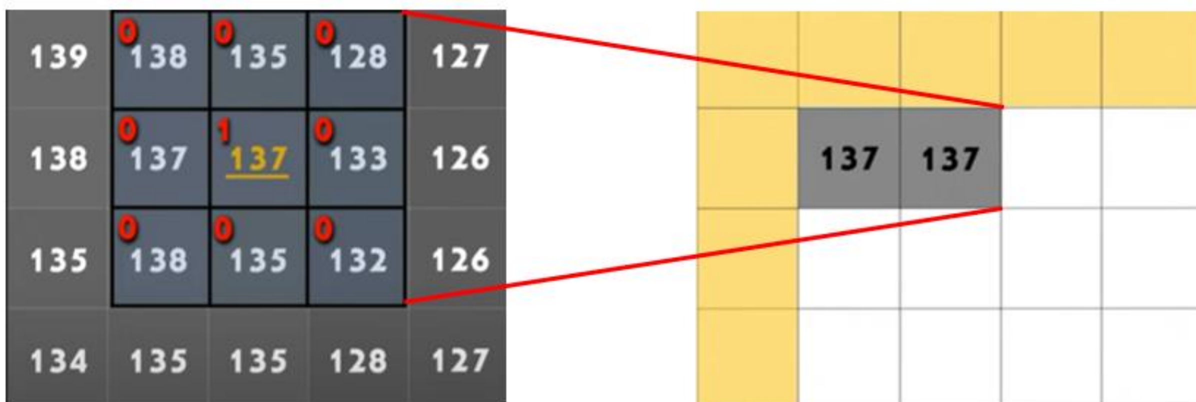


Figura 20: Operación de convolución, que aplica el filtro (valores en rojo) a la segunda zona de la imagen, y el resultado del producto escalar se posiciona en la zona correspondiente (segundo píxel) de la nueva imagen [20]

Es importante aclarar que, como se muestra en el ejemplo, cada paso de avance de 1 píxel del filtro se solapa con el anterior tanto de manera horizontal como vertical cuando empieza una nueva fila.

Según el filtro que tengamos, se aplicarán ciertos cambios en la imagen resultante. En el caso de este filtro, el resultado será el mismo ya que al recorrer píxel por píxel y multiplicando por 1 solo el valor del píxel central del filtro no cambia nada.

La aplicación de este proceso a una red neuronal es factible gracias a las capas convolucionales, que se emplean de la siguiente manera:

- La capa de entrada de la red se compondría de una neurona por cada píxel, y recibe como entrada el valor de ese píxel (recordemos que, para pasar imágenes a una red neuronal, necesitamos traducir a un vector de valores los datos de la imagen). Si la imagen de entrada es de 28x28, habrá 784 neuronas en la capa de entrada.
- En la siguiente capa oculta se aplica una capa convolucional, y fijándonos en el caso de estudio que estamos tratando, cada neurona de la capa oculta se conectaría a 9 neuronas de la capa de entrada,

y aplicaría los 9 pesos del filtro a los 9 valores que recibe de esas 9 neuronas, ejecutando la operación de convolución. Su salida será el producto escalar que resulta.

- Un filtro definido por una matriz de pesos y un sesgo solo permite detectar una característica concreta. Por ejemplo, un filtro concreto puede servir para localizar bordes, como se puede ver en la imagen [*Figura 21*]. Para el reconocimiento de imágenes, las redes neuronales convolucionales aplican varios filtros a la vez, uno para cada característica que queramos detectar [20]. Es por ello por lo que **las capas convolucionales incluyen varios filtros**.

Figura 21: Ejemplo de filtro utilizado para localizar bordes en una imagen [20]

- Por cada filtro que se aplica en una capa, se genera una salida del tamaño de la nueva imagen. Si se aplican 32 filtros a una imagen de entrada de 28x28, tendremos 32 nuevas imágenes de dimensión 24x24, donde en cada una se resalta una característica.

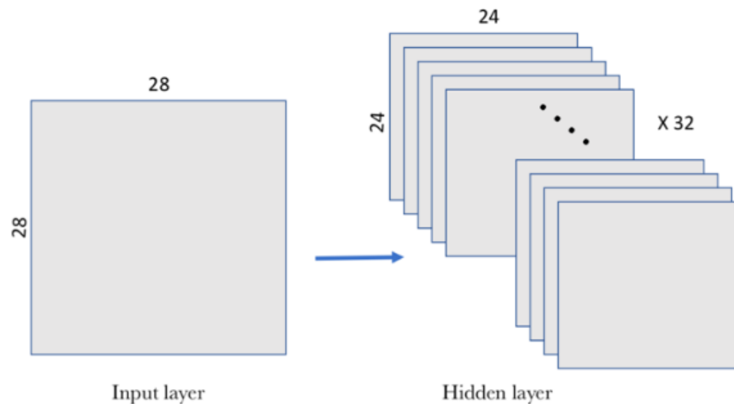


Figura 22: Operación de convolución: una capa de entrada que introduce una imagen de 28x28 píxeles, a la que se le aplica una operación de convolución con 32 filtros en la capa oculta, dando como resultado 32 imágenes de 24x24 píxeles cada una [14]

Si nos fijamos en la última imagen, se puede ver que las nuevas imágenes que se generan disminuyen su tamaño un poco, exactamente dos píxeles en cada dimensión: esto es consecuencia del filtro que se va deslizando por la imagen, que al tener una dimensión 3x3 como es el caso anterior [*Figura 18*], el primer filtro que se aplica en la esquina superior izquierda tiene como píxel central el de la posición (2,2), porque los de la primera columna/fila no tienen valores en todo su alrededor para poder aplicar el cálculo, y lo mismo pasará con la última fila y la última columna.

Hay ocasiones que se quiere obtener una imagen de salida de las mismas dimensiones que la entrada. Existe el hiperparámetro ***padding,*** con el que se pueden agregar ceros alrededor de las imágenes de entrada antes de deslizar los filtros sobre ellas. De esta manera, agregando una columna arriba, otra debajo, una fila arriba y otra debajo con ceros, la salida del filtro dará como resultado una imagen de las mismas dimensiones que la entrada.

El ***stride*** es otro hiperparámetro que indica el número de pasos en el que se mueve la ventana de filtro (anteriormente se ha explicado el ejemplo con *stride* a 1). Si se pusiera un *stride* a 2, el salto del filtro sería cada dos píxeles, y esto provocaría un decrecimiento en los tamaños de los resultados. Simplifica la información. El *stride* tiene por defecto el valor *(1,1)* que indica por separado en avance por ambas dimensiones [14].

2.3.2 Pooling

Además de las capas convolucionales, las CNNs se acompañan de las capas *pooling*, que son aplicadas inmediatamente después de las de convolución. Esta operación lo que pretende es simplificar la información recogida de la capa convolucional, generando una versión más concentrada.

La técnica que usaremos y la más habitual es la *max-pooling*. Básicamente, a partir de una ventana de cierta dimensión, recorre las imágenes resultantes de la convolución, y de los valores que tiene seleccionados se queda con el valor máximo. Si, como ejemplo, usamos una ventana de 2x2 a las imágenes de 24x24 que se sacaban de la convolución anterior, la dimensión resultante de la imagen se divide por 4, quedando una imagen de 12x12.

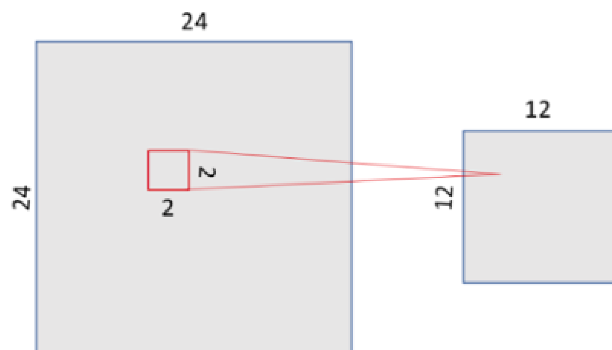


Figura 23: Operación de pooling sobre una imagen de 24x24 con una ventana de 2x2, disminuye la dimensión de la nueva imagen a la mitad [14]

Es importante destacar que el tipo de salto en esta operación es diferente a la operación de convolución y no es configurable. En ningún momento se solapa una ventana con la siguiente, sino que se escoge el cuadrado que sigue.

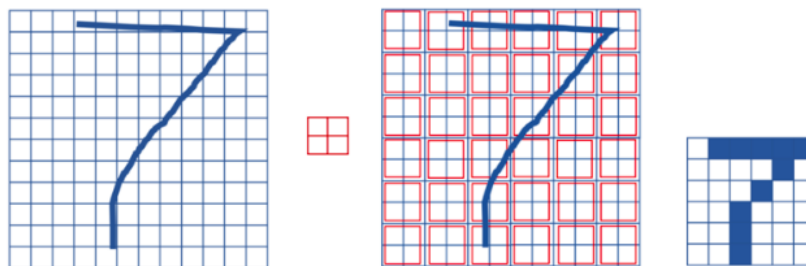


Figura 24: Operación de pooling: las ventanas de la operación de pooling no se solapan entre ellas [14]

También está la opción de *average-pooling* que, en vez de escoger el valor máximo de la ventana, transforma la salida en el promedio de los valores [14].

2.4 Técnicas avanzadas de entrenamiento

2.5.1 Data Augmentation

Es una técnica muy usada para reducir la probabilidad de sobreajuste (*overfitting*) en el aprendizaje de los modelos neuronales. Ofrece una solución al problema de *datasets* pequeños, ya que genera más datos de entrenamiento a partir de los datos disponibles. Esto se consigue aplicando una serie de transformaciones aleatorias a los datos ya existentes.

Aplicado a este proyecto, la ejecución del *Data Augmentation* a los conjuntos de datos consiste en la aplicación de transformaciones sencillas a sus imágenes, que produzca una nueva imagen para el entrenamiento. Estas transformaciones, pueden ser tan simples como voltear, hacer zoom, cambiar el brillo, rotar, etcétera. El programador puede elegir y combinar varias transformaciones, eligiendo los intervalos entre los que cada transformación puede realizarse, y el programa es el que aplica aleatoriamente un nivel de transformación a partir de ese intervalo (por ejemplo, la imagen original puede rotar entre -20 y 20 grados para generar una nueva imagen).

Es importante destacar que las transformaciones se aplican de manera automática durante el entrenamiento, sin necesidad de modificar y aumentar los datos en disco. El modelo ve una imagen generada aleatoriamente una única vez. La única desventaja que presenta es que el preprocesado es más lento.

En las dos redes neuronales que se implementan en este trabajo se utiliza esta técnica, con el módulo *ImageDataGenerator* de la librería Keras (*apartado 3*) [21].

2.5.2 Dropout

Técnica más usada para mitigar el *overfitting*, que se produce en los conjuntos de neuronas, y consiste en "apagar" cierto porcentaje de neuronas durante la fase de entrenamiento, es decir, no tener en cuenta estas neuronas durante una iteración en el proceso de aprendizaje.

El objetivo que tiene es mejorar la generalización del modelo, forzando a las neuronas a aprender características útiles y reduciendo la dependencia entre las unidades [21].

2.5.3 Transfer Learning

Otra técnica muy útil y de las más importantes de *Deep Learning*. Se basa en que, en lugar de tratar de entrenar una red desde cero, que implica tener un *dataset* bastante amplio y requiere una gran cantidad de tiempo de computación para el proceso de entrenamiento, se usa de una red preentrenada, es decir, permite la descarga de un modelo *open-source* que haya sido entrenado previamente con un amplio *dataset* y con los pesos ya configurados, y una vez elegido el modelo, se continúa entrenando con un conjunto de datos específico para resolver el problema.

Existe un amplio elenco de modelos preentrenados de descarga. Muchos de ellos vienen entrenados de *ImageNet*, un conjunto de datos muy popular en el campo de las redes neuronales [21].

3 METODOLOGÍA

En este capítulo, presentamos el procedimiento utilizado en este trabajo para abordar el desafío de la segmentación de imágenes mediante el empleo de redes neuronales convolucionales. Nuestro objetivo principal es desarrollar y comparar dos enfoques basados en estas redes, con el fin de lograr una segmentación precisa y efectiva de imágenes en un contexto específico.

3.1 Contexto

El propósito principal de este trabajo se basa en realizar un estudio del campo del *Deep Learning* a través de las redes neuronales, tanto en su aspecto teórico como en su aplicación práctica. Una vez se ha estructurado la parte teórica y se han asentado los conceptos, enfocamos la parte práctica en las redes neuronales convolucionales para adquirir un conocimiento más profundo sobre su funcionamiento y la evaluación de su rendimiento ante situaciones reales.

Para la parte práctica, se presenta un problema de segmentación de imágenes que tratamos de abordar de varias maneras. Partimos de una serie de imágenes que en ningún momento son sacadas de una base de datos, sino que han sido producidas en un ámbito personal. Constituyen un *dataset* más pequeño de lo habitual, que suele ser de miles de imágenes, y la temática se enfoca en plantas. El objetivo va a consisitir en la segmentación de las hojas en planta, independientemente del tipo de hoja o planta.

La primera parte del trabajo consitirá en la segmentación de las hojas a través de una biblioteca de *Python* de código abierto, *OpenCV*, que se utiliza para aplicaciones de visión por computadora y procesamiento de imágenes. Con esta librería vamos a poder obtener la segmentación de imágenes que buscamos, teniendo que manejar ciertos parámetros que explicaremos en esta sección. Una vez esté el *dataset* organizado, se pasará a la implementación de dos modelos de red neuronal, uno de clasificación y otro de segmentación, y ambos se centrarán en el mismo objetivo de segmentar las hojas con sus distintos planteamientos que iremos desgranando.

3.2 Recursos empleados

Para llevar a cabo todo el desarrollo del proyecto, ha sido necesario estudiar los entornos disponibles que mejor se adaptaban a las necesidades del mismo, tanto a nivel software como hardware. Se procede a detallar los productos escogidos que han permitido la implementación, diseño y el correcto funcionamiento de las redes neuronales.

El lenguaje de programación empleado en este proyecto es *Python*, un lenguaje ampliamente utilizado en el ámbito de la inteligencia artificial. La implementación y ejecución del código se realiza en la plataforma de *Google Colab*, que está basada en la nube y ofrece un entorno de notebook interactivo (similar al entorno de *Jupyter Notebook*). Una ventaja de usar *Google Colab* tiene que ver con el acceso fácil que proporciona a recursos computacionales, y esto nos va a ayudar a acelerar mucho el proceso de entrenamiento y a mejorar

el rendimiento de los modelos de red neuronal gracias al acceso a la GPU que tenemos disponible gratuitamente de manera limitada. La **GPU (Unidad de Procesamiento Gráfico)** se trata de un componente hardware diseñado para realizar cálculos intensivos y paralelos, particularmente eficaces para acelerar el entrenamiento de redes neuronales profundas por su gran capacidad de cálculo de matrices multidimensionales. Se componen de miles de núcleos de procesamiento que trabajan de manera simultánea para realizar estos cálculos masivos. Los resultados en aceleración y rendimiento frente a una CPU de cualquier portátil personal son muy visibles. Con el uso gratuito del recurso, no se usa un tipo de GPU concreto, sino que va variando [21].

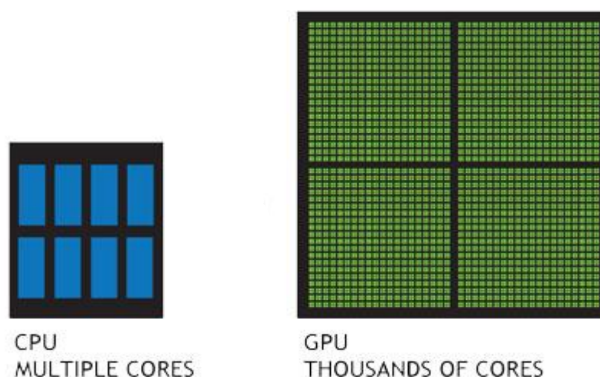


Figura 25: Comparación visual del número de cores de una CPU y una GPU [22]

A nivel de software, para el diseño de redes neuronales nos apoyamos en la librería de código abierto ***Keras***, escrita en lenguaje *Python*, que proporciona de manera sencilla la creación de modelos usando como *backend* librerías como *TensorFlow.* Fue desarrollado y es mantenido por François Chollet, ingeniero de Google. Esta librería nos va a aportar bastante simplicidad y legibilidad en el desarrollo de modelos *Deep Learning*.

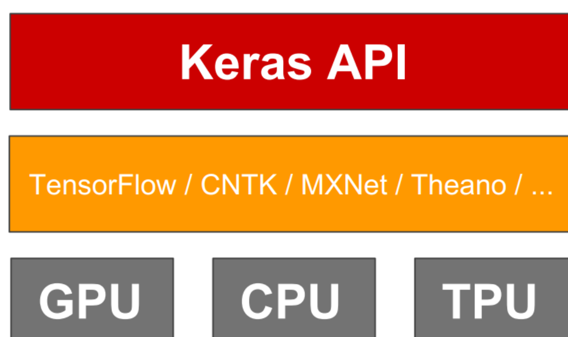


Figura 26: Torre de integración de Keras, impulsado por hardware CPU, GPU y TPU [23]

Otra librería de Python importante va a ser la de OpenCV (Open Source Computer Vision Library), también de código abierto y ampliamente utilizada para procesamiento de imágenes y visión por computador. Proporciona gran cantidad de funciones para tareas como la manipulación de imágenes, reconocimiento facial o la detección de objetos, entre otras. Su papel va a ser fundamental para el preprocesamiento de imágenes y la preparación de los labels en el conjunto de datos que vamos a utilizar para entrenar los modelos de redes neuronales.

Figura 27: Logo OpenCV [24]

En cuanto al *dataset* utilizado, para el primer modelo se planteó el reto de resolver el problema de segmentación utilizando un *dataset* muy pequeño, compuesto por 12 imágenes de almendros, y para el segundo modelo hizo falta ampliarlo para su correcto funcionamiento (pues con un *dataset* tan pequeño no era capaz de aprender), con lo que aumentó a 162 imágenes con fotos de distintas plantas de jardín.

Para resolver el problema de segmentación de hojas en planta, en primer lugar, será necesario construir el conjunto de datos para entrenar la red mediante aprendizaje supervisado, con lo que hará falta obtener las máscaras de las imágenes con el apoyo de la librería *OpenCV*. Una vez tengamos las imágenes y sus correspondientes máscaras, lo siguiente será preparar el *dataset* en cada modelo. Para ello, se utilizará la clase *ImageDataGenerator* de la librería Keras, que preprocesará estos datos y los organizará en lotes para entrenar la red. A continuación, se diseñará el modelo de capas que se va a entrenar, y se ajustarán los hiperparámetros previamente definidos, también mediante la librería Keras. Finalmente, se evaluarán los resultados obtenidos, que en ambos casos consistirá en una máscara que resuelva la segmentación de las hojas de cada imagen.

3.3 Preparación de los datos

La primera fase de este proyecto va a consistir en sacar las máscaras de lo que va a conformar el *dataset*. Con máscara, nos referimos a una matriz binaria (como las de este trabajo, aunque puede ser de más valores numéricos) que identifica y delimita las regiones de interés de una imagen. En nuestro caso más concreto, se busca que la máscara identifique las regiones específicas donde se encuentran las hojas en las imágenes, e identifique esos píxeles con el valor 1 (color blanco), mientras que los píxeles con valor 0 se consideran parte del fondo sin interés para el estudio. Para sacar las máscaras, se dispone de 12 imágenes de almendro.

Para desarrollar esta tarea, vamos a apoyarnos en la librería de Python antes mencionada, *OpenCV*. Los siguientes puntos pasan a explicar, de manera ordenada, las fases que se han ido desarrollando con las funcionalidades de esta biblioteca.

3.3.1 Cambio del espacio de color

En primer lugar, es importante montar una estructura de directorios que como mínimo permita cargar las imágenes de un directorio y guardar las máscaras resultantes en otro. Una vez se carga la imagen en el programa, el primer proceso por el que pasa es el cambio de espacio de color. Un **espacio de color** consiste en un sistema que se utiliza para representar los colores de manera numérica, y cada espacio de color tiene su propio método.

El espacio más común y del que parten las imágenes cargadas es el RGB, que se basa en la combinación de tres colores primarios: azul, verde, rojo. Cada color primario representa un canal que puede tener asignado un

valor del 0 al 255, que permite un total de 256 niveles de intensidad para cada componente. El valor muestra el nivel de intensidad de cada color primario, y su totalidad de combinaciones abarca toda la gama de colores.

Vamos a cambiar el espacio de las imágenes del RGB al HSV. Este modelo se caracteriza por trabajar con 3 componentes básicos:

- Matiz (*Hue*) se refiere a la tonalidad o tipo de color, representada por el ángulo en un círculo de colores primarios donde los primarios se encuentran en posiciones específicas (rojo esta en 0°, verde 120°, etc).
- Saturación (*Saturation*) se refiere a la intensidad o pureza del color.
- Luminancia (Value) representa el nivel de brillo o luminosidad de color.

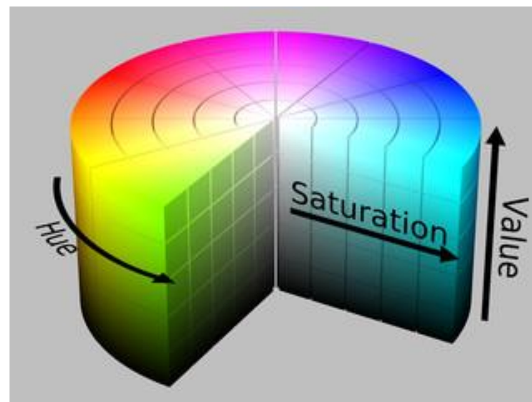


Figura 28: Cilindro de representación del espacio HSV [25]

El método usado de la librería OpenCV para esta tarea es el siguiente:

```
# Convertir de BGR a HSV
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
```

Código 1: Función para cambiar el espacio de color de RGB a HSV

Este método convierte la imagen en formato RGB guardado en la variable *img* al formato HSV (indicado con el código de conversión *COLOR_BGR2HSV*), y guarda la imagen convertida en la variable *hsv*.

Este espacio nos va a permitir un filtrado más fácil de colores específicos para la segmentación de las hojas que el modelo RGB. Lo explicamos en la siguiente fase.

3.3.2 Definición de umbrales

Se va a realizar la segmentación basada en el color. Como se busca segmentar las hojas de fotos de almendros, tenemos que ajustar los umbrales de color en el espacio HSV para que abarque todas las tonalidades del color verde (al menos las que cubren las hojas), sin que se solapen con los colores y tonos de otros elementos de la imagen.

Los umbrales que marcamos para el color verde en el espacio de color HSV, tras varios intentos de prueba y error, son los siguientes:

```
# Definicion umbrales color verde
lower_green=np.array([29,40,40])
upper_green=np.array([100,255,255])
```

Código 2: Umbrales del espacio HSV para el color verde

Se define el máximo y mínimo en dos variables para que luego la función *inRange*, de *OpenCV*, tenga como referencia ese intervalo.

```
mask = cv2.inRange(hsv, lower_green, upper_green)
```

Código 3: Función para verificar los píxelesque están dentro del umbral

Esta función básicamente va a devolver una matriz de unos y ceros, donde los unos serán aquellas zonas de la imagen (píxeles) que estén dentro del intervalo que se impone, y cero serán las zonas que se encuentren fuera. Con lo cual, máscara será la imagen resultante definida por esa matriz, traduciendo el 1 al color blanco (zona verde de hojas) y el 0 con negro (zona de fondo). Un ejemplo de la máscara para una imagen 3x3 [*Figura 29*].

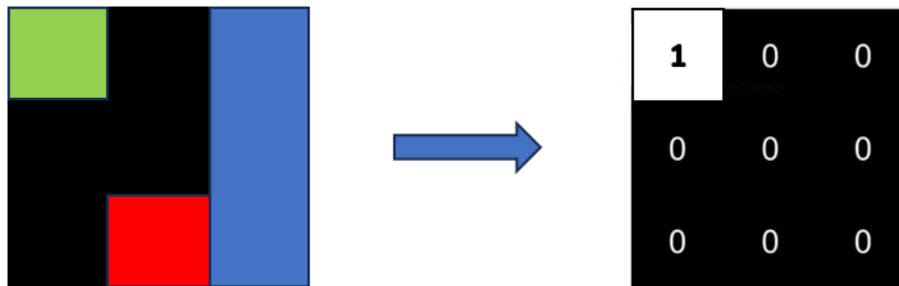


Figura 29: Máscara que detecta los píxeles de color verde

3.3.3 Transformaciones morfológicas

Las tranformaciones morfológicas son operaciones simples basadas en imágenes normalmente binarias. Para realizarlas, se utilizan como *inputs* la máscara y el filtro de la operación que recorrerá la imagen (como en la operación de convolución del punto 2.2.8). Para máscaras seleccionadas del *dataset*, ha sido necesario utilizar la operación de *Opening* para mejorar su precisión y eliminar pequeñas zonas de ruido, como las que se muestran en la ilustración [*Figura 30*].

Figura 30: Operación de Opening antes y después [26]

La operación de *Opening* se aplica con la combinación de dos operaciones básicas: *Erosión* seguido de *Dilatación*. En la primera, se desliza el filtro a través de la máscara, y cambia el píxel principal a 1 solo si todos los píxeles de la posición del que se encuentra el filtro son 1, de lo contrario se hace 0. En la segunda operación vuelve a pasarse el filtro, pero esta vez es justo lo contrario, un elemento de píxel será 1 si al menos un píxel de dentro del filtro es 1. El resultado de la combinación, con una dimensión de filtro considerable, limpia el

ruido de la imagen [26].

```
if file_name [-6] == '-' and file_name[-5] == 'O':
  kernel = np.ones((2,2),np.uint8)
  mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
```

Código 4: Aplicación de operación Opening en las máscaras

Las imágenes con las máscaras que queremos aplicar la operación van a renombrarse con un "-O" final, después se define un filtro de dimensión 2x2 y se llama a la función *morphologyEx* para la operación de *Opening*. Se modifica la misma variable *mask* que metemos como *input* junto al filtro.

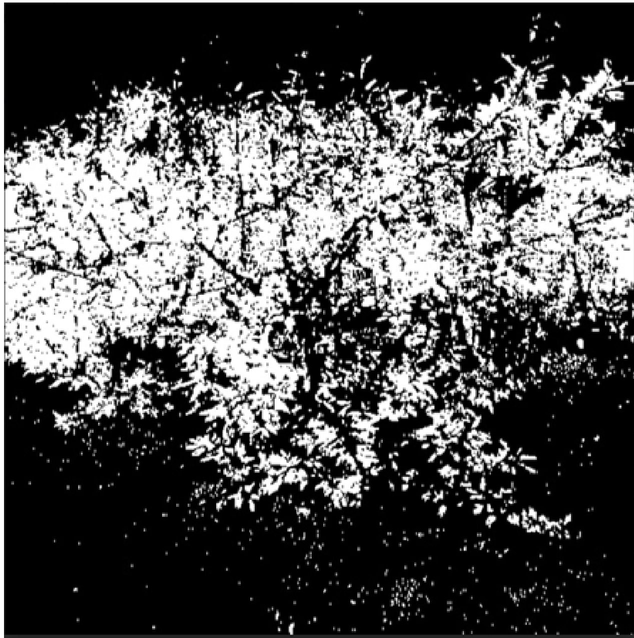
3.3.4 Evaluación

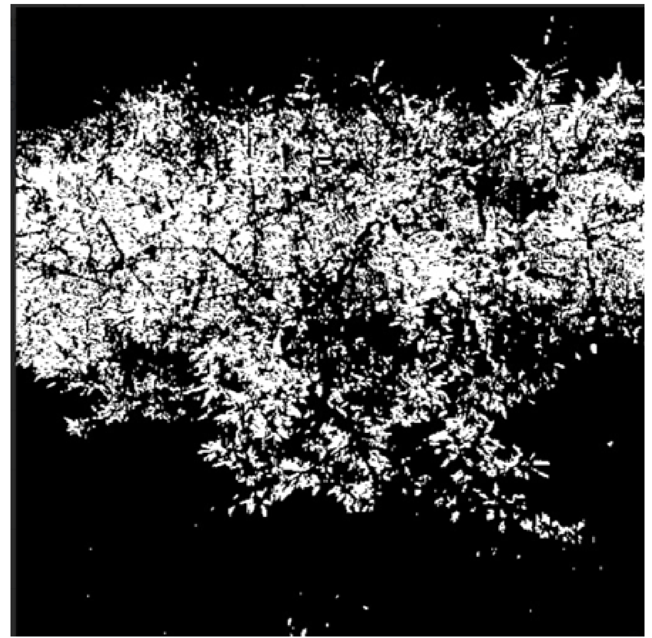
Ya solo queda comprobar que se han obtenido unos resultados acordes a los previstos, y si es así, almacenarlos en el directorio que hayamos asignado con la función *imwrite* de *OpenCV*. Para comprobar el efecto que genera en la imagen la transformación morfológica, se procede a mostrar uno de los resultados que necesitaban la reducción de ruido [*Figura 32*].

- La imagen original es la siguiente:

Figura 31: Imagen planta (1)

- La máscara resultada, antes y después de la operación de *Opening*:





ANTES **DESPUÉS**

Figura 32: Máscara antes y después de aplicar la transformación morfológica Opening

Con todas las máscaras obtenidas, ya tenemos el conjunto de datos preparados con sus imágenes y *labels* correspondientes para el aprendizaje supervisado de las redes neuronales que vamos a implementar.

3.4 Implementación de las redes neuronales:

La parte más práctica de este trabajo realiza la implementación de dos redes neuronales distintas que son entrenadas para el mismo fín, con un conjunto de datos parecido, que se amplia por necesidad para la segunda red neuronal. Se va a dividir este punto en dos secciones que profundicen en el desarrollo de cada red neuronal: el modelo, el procesamiento del conjunto de datos, las técnicas de resolución del problema, etc.

3.4.1 Primer modelo: red neuronal de clasificación

La red neuronal convolucional que se va a tratar se define como un sistema de clasificación. Su objetivo principal consiste en categorizar cada imagen en una de las dos clases disponibles, que se definen como la clase *leaf_images* y la clase *background_images*. Mediante el aprendizaje del algoritmo, este será capaz de asignar un valor de 1 o 0 a cada imagen, dependiendo de la clase a la que pertenezca.

Esto en un principio podría no tener sentido si el fín que se busca es el de segmentar las hojas de las imágenes y sacar su máscara, y no el de clasificar las imágenes. Aparte, todas ellas contienen hojas, con lo cual tendría poco sentido a la hora de realizar una clasificación. El caso es que sí lo tiene, y esto se debe a una estrategia que aplicamos, que se basa en que antes de construir el conjunto de datos, subdividimos las imágenes de almendros en cuadrados de 8x8 píxeles, y cada cuadrado es una imagen perteneciente al conjunto de datos con el que vamos a entrenar la red. Seguidamente, la red va a tratar de clasificar estas imágenes dependiendo de si contiene parte de hojas (identificado por el color verde) o, por el contrario, es una parte de *background* (no contiene hoja y no interesa). Una vez esté la clasificación de todos los cuadrados de una imagen resuelta,

se genera una matriz de dimensión similar a la imagen original, y se ponen las casillas a 1 o 0 dependiendo de la clase a la que pertenece cada cuadrado. De ahí se sacará la máscara.

A continuación, vamos a explorar en mayor profundidad el proceso de desarrollo de este modelo y las diferentes etapas necesarias para construir una red neuronal.

3.4.1.1 Dataset

En primer lugar, se diseña la estructura de directorios del conjunto de datos. La división del dataset se va a efectuar siguiendo el punto *2.2.6.1*: de las 12 imágenes iniciales de las que se parte, 8 van a pertenecer al grupo de *training*, 2 al grupo de *validation* y 2 al grupo de *testing*. Sabiendo esto, se sigue la siguiente estructura:

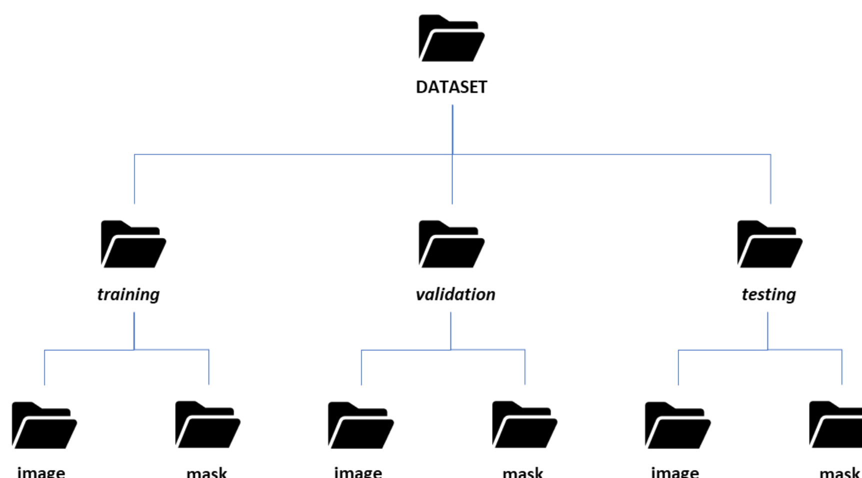


Figura 33: Estructura del conjunto de datos. Modelo 1

Se va a aplicar a las imágenes de almendros la subdivisión en cuadrados de 8x8 píxeles. En ningún momento se solapa una división con la siguiente, y de cada división se va a sacar una imagen 8x8 que pertenecerá al conjunto de datos de entrenamiento. Este proceso se cumple gracias a la función *división_image*, que realiza lo siguiente:

1. La función recibe la ruta del directorio donde se encuentran las imágenes que vamos a subdividir. Este directorio va a contener un directorio de imágenes y otro de máscaras, y cada máscara se llamará igual que la imagen original a la que corresponde.

2. De esa ruta de entrada, carga la primera imagen y la máscara con el mismo nombre de cada carpeta, con la intención de recorrerlas simultáneamente. Cada recorte que haga en una zona de la imagen original, lo va a hacer también en la máscara. Para la división en cuadrados de la imagen nos apoyamos en el módulo *Image* de la librería *PIL* de Python, con el método *crop* al que hay que pasarle la dimensión.

3. Por cada subimagen, se ejecuta un proceso de clasificación. Se comprueba en la submáscara que le corresponde si contiene zonas blancas que indiquen que esa subimagen cubre una zona de hojas (la matriz 8x8 de la submáscara tiene algún valor a 1), o

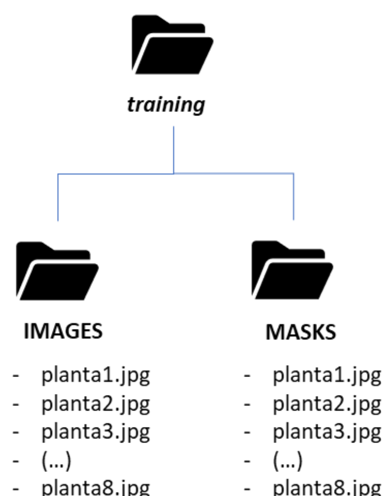


Figura 34: Estructura conjunto training. Modelo 1

en cambio la submáscara es completamente negra y por tanto no contiene hojas (la matriz es nula). Esta disyuntiva va a determinar si guardamos la subimagen en la lista de imágenes que contienen hojas (*leaf_images*) o la lista de imágenes sin hojas (*background_images*).

4. Se sigue este mismo procedimiento para el resto de las imágenes contenidas en la ruta de entrada. Finalmente, la función va a delvolver las listas *leaf_images* y *background_images*.

La función se va a aplicar para el directorio de *training* y para el de *validation*. En los datos de *testing* veremos que no es necesario. Se acaban obteniendo 4 listas:

```
# Sacamos las listas de las subimagenes
train_leaf_images, train_background_images = division_imagen(train_dir)
valid_leaf_images, valid_background_images = division_imagen(valid_dir)
```

Código 5: División de imágenes en subimágenes 8x8 que se guardan en listas según contengan hojas o no.

Vamos a referirnos a las imágenes por sus clases: si contienen hojas, su clase será *leaf,* y si no contienen, pertenecerán a la clase *background*. Para hacernos una idea de la dimensión del *dataset* inicial:

- Las listas de *training* contienen 1.398.749 imágenes de la clase *leaf* y 591.907 de *background*.
- Las listas de *validation* contienen 370.567 imágenes de la clase *leaf* y 127.097 de *background*.

De estas listas vamos a coger una pequeña muestra de subimágenes de cada una con el objetivo de igualar el número de datos con el que se entrena la red, lo que va a aportar un aprendizaje más equilibrado y preciso. De las listas de *training* se sacan 50.000 subimágenes, y de las listas de validación, 10.000. Esto se hace con la siguiente función de la librería *random* aplicado a las cuatro listas:

```
valid_leaf_samples = random.SystemRandom().sample(valid_leaf_images, 10000)
```

Código 6: Se coge una muestra aleatoria de 10.000 datos de validación que contengan hojas

Seguidamente, se van a coger las dos listas de muestras para *training* y se van a unir en una. El resultado va a ser una nueva lista *train_samples* de 100.000 subimágenes donde las primeras 50.000 pertenecen a la clase *leaf* y las últimas 50.000 a la clase *background*. Repitiendo el proceso para la lista de *validation*, se obtiene una lista con 20.000 datos con los primeros 10.000 pertenecen a la clase *leaf* y los últimos 10.000 a la clase *background*. Es importante tener claro este proceso, porque va a facilitar la asignación de las *labels*.

```
train_samples = train_leaf_samples + train_background_samples
valid_samples = valid_leaf_samples + valid_background_samples
```

Código 7: Se combinan las listas de entrenamiento y de validación

Ahora, sabiendo que la primera mitad de subimágenes de las dos listas pertenecen a la clase *leaf*, y la segunda mitad a la clase *background*, toca generar las listas de labels. Las dos clases se van a representar con un número identificativo, siendo la clase *leaf* el número 1 y la clase *background* el 0, y se van a generar dos listas binarias:

- Lista *train_labels*, de dimensión 100.000, que va a contener los primeros 50.000 valores con unos, y los últimos 50.000 con ceros.
- Lista *valid_labels*, de dimensión 20.000, que va a contener los primeros 10.000 valores con unos, y los últimos 10.000 con ceros.

De esta manera conseguimos que la posición de un dato de la lista de subimágenes coincida con la posición de su *label* correspondiente de la lista de *labels*, tanto en los datos de *training* como en los de *validation.*

3.4.1.2 Preprocesado de datos

El siguiente paso es ver cómo cargar las imágenes de las listas a la red neuronal, junto con sus respectivas etiquetas. Al estar trabajando con la librería Keras, esta contiene una clase generadora que va a servir de apoyo para generar los lotes de datos de imágenes durante el entrenamiento del modelo CNN. También nos va a permitir preprocesar las imágenes en tareas como la normalización de los píxeles y el redimensionamiento de imágenes. Este objeto es ***ImageDataGenerator***.

Además, proporciona una amplia gama de transformaciones y aumentos de datos de imágenes (como rotación, traslación, cambio de zoom o de brillo, recorte aleatorio, volteo horizontal, entre otros), pero esta ventaja la de variabilidad de datos será más útil para la segunda red neuronal, ya que el número de datos será muy inferior a esta y las imágene más complejas.

Volviendo a la práctica, usamos esta clase 'generador' de la siguiente manera. En primer lugar, se configuran dos objetos de la clase *ImageDataGenerator*, una para *training* y otra para *validation*. Estos objetos van a generar lotes de datos durante el entrenamiento y la validación del modelo de red neuronal convolucional. Para el *Data Augmentation*, dentro del constructor del objeto *train_datagen*, se definen varios argumentos que van a aplicar transformaciones aleatorias a las imágenes en cada lote.

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    vertical_flip=True)

valid_datagen = ImageDataGenerator(rescale=1./255)
```

Código 8: Se instancian dos objetos ImageDataGenerator con sus argumentos de transformación de imágenes. Modelo 1

Las transformaciones que se van a aplicar son (en el orden que se listan, excluyendo *rescale*):

- Rotación aleatoria en un rango de -40 y 40 grados.
- Desplazamiento horizontal aleatorio en un rango de -20% a 20% del ancho de la imagen.
- Desplazamiento vertical aleatorio en un rango de -20% a 20% del alto de la imagen.
- Zoom aleatorio en un rango de 20% de la escala de la imagen.
- Volteo horizontal de algunas imágenes.
- Volteo vertical de algunas imágenes.

El argumento *rescale* se refiere al proceso de normalización de imágenes, y se aplica en ambas instancias debido a que es un paso esencial en el preprocesamiento de los datos. Es necesario un ajuste de los valores de los pixeles de una imagen para que estén en el rango entre 0 y 1 antes de cargarlos en la red. Esto va a permitir una mejora del rendimiento del entrenamiento, evitando riesgos de valores altos y de rangos grandes, y una mejor generalización y adaptabilidad de diferentes conjuntos de datos.

Otro apunte importante es que para los datos de validación no es necesario aplicarle *Data Augmentation*, ya

que el propósito de estos datos no es mejorar el modelo del entrenamiento, sino evaluar como evoluciona el rendimiento y la capacidad de generalización del modelo tras cada *epoch* de entrenamiento. Los datos de validación deben ser representativos del conjunto de datos real.

A continuación, se aplica el método *flow* de la clase *ImageDataGenerator* para generar el flujo de datos de las imágenes y sus etiquetas a partir de las listas creadas anteriormente.

```
batch_size = 64
train_generator = train_datagen.flow(
    np.array(train_samples), train_labels,
    batch_size=batch_size,
    shuffle=True
)
validation_generator = valid_datagen.flow(
    np.array(valid_samples), valid_labels,
    batch_size=batch_size,
    shuffle=False
)
```

Código 9: Implementación de los generadores de lotes de datos. Modelo 1

El primer y segundo argumento que le pasamos son la lista de los datos de entrada y la lista de las etiquetas. Usamos el *np.array* porque, para representar los datos en Keras, se usan *arrays* multidimensionales conocidos como **tensores**, y necesitamos convertir esos datos a un formato de *array* 4D que cumpla las dimensiones ***(numero de imágenes, altura de imágenes, anchura de imágenes y canales)***. Los tensores que contiene la lista son de dimensión (8, 8, 3), y la dimensión adicional comprende el número de datos contenidos en esa lista.

Los demás argumentos indican el tamaño de los lotes que se van a generar en cada iteración del entrenamiento y que se mezclen los datos aleatoriamente antes de cada *epoch (shuffle)*, ofreciendo variabilidad de datos, que al igual que con el *Data Augmentation*, tampoco es necesaria para los datos de validación.

3.4.1.3 Modelo

El modelo de red neuronal utilizado para este conjunto de datos es el siguiente:

```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (2,2), input_shape=(DIM_SQUARE,DIM_SQUARE,3),
                           activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(64, (2,2), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Flatten(),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(units=64, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

Código 10: Estructura de capas. Modelo 1

Este modelo de red neuronal convolucional comienza con una capa convolucional (*sección 2.3.1*) de entrada con 32 filtros diferentes y un tamaño de filtro (2, 2). Acepta tensores de entrada de dimensión (8, 8, 3) para imágenes de altura y anchura 8x8 y con 3 canales de color (espacio de color RGB). Se aplica la función de activación *relu* (*sección 2.2.5*) que es la función que se suele aplicar alas capas convolucionales. Seguidamente, tenemos una capa de *Pooling* que aplica la operación de *MaxPooling* (*sección 2.3.2*). Se realiza otro nivel de abstracción con la aplicación de una nueva capa convolucional seguida de una *MaxPooling,* esta vez con 64 filtros.

La elección de 32 y 64 filtros en las capas convolucionales ha sido obtenida a partir de la práctica común de libros y ejemplos donde implementan redes convolucionales, y se suelen seleccionar esos valores. Normalmente, se comienza en la primera capa con un número de filtros bajo, y a medida que se profundiza en las capas posteriores, se suele aumentar de manera gradual, por ejemplo, multiplicando por dos el valor previo.

Ahora se aplica la capa Flatten, que "aplana" la salida de la capa anterior para convertir los datos en un array unidimensional (vector) para que las siguientes capas densamente conectadas puedan operar con esos datos. Las capas Dense van a ser las encargadas de clasificar, con lo que la última capa debe reducirse a una única neurona para que defina la clasificación binaria con un 1 o un 0 en su salida. Así que se usan dos capas Dense, la primera con 64 neuronas con función de activación *relu*, responsable de tomar las carácterísticas extraídas de las capas convolucionales para clasificar los datos, y luego la capa de salida de una única neurona con función de activación *sigmoid* (*sección 2.2.5*), muy usada para la clasificación binaria.

Aparte del diseño del modelo, seleccionamos los hiperparámetros que se van a aplicar en el entrenamiento, mediante la función *compile* de la librería Keras:

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Código 11: Hiperparámetros. Modelo 1

Como función de pérdida se usa la *binary_crossentropy*, muy utilizada para tareas de clasificación binaria, el optimizador que se establece es el *Adam*, que es ampliamente utilizado y algo más avanzado que el de *grandient dencent* (*sección 2.2.6.3*), que tiene la ventaja de adaptación del *learning rate* de manera dinámica en función de los gradientes acumulados durante el entrenamiento y esto ofrece un ajuste más eficiente en los pesos. Por último, se va a monitorizar la evolución del entrenamiento con la métrica *accuracy*, que se verá como funciona en la sección de resultados del modelo.

Finalmente, se inicia el proceso de entrenamiento con el método *fit* de la librería Keras, al cual se le pasa los generadores creados anteriormente, el número de *epochs* que se van a aplicar y el número de pasos por época en los datos de entrenamiento y validación, que simplemente se define dividiendo el número de datos de entrenamiento y validación entre el tamaño de los lotes.

```
history = model.fit(train_generator,
                    steps_per_epoch=len(train_samples)//batch_size,
                    epochs=10,
                    validation_data = validation_generator,
                    validation_steps = len(valid_samples)//batch_size)
```

Código 12: Inicio del entrenamiento. Modelo 1

El historial del entrenamiento se guarda en la variable *history*, que va a permitir acceder a los registros de las métricas obtenidos durante el proceso de entrenamiento.

3.4.1.4 Testing

Una vez el modelo termina el entrenamiento, toca comprobar su funcionamiento ante unos datos que nunca ha visto anteriormente. Es necesario recordar que para esta etapa se asignaron dos imágenes originales. Para este modelo, antes de comprobar si funciona la estrategia de dividir la imagen en subimágenes, debemos comprobar si funciona la clasificación de las subimágenes, y ya luego formar la máscara final.

Se procede a realizar la misma tarea de subdividir las imágenes en pequeños cuadrados 8x8, y una vez hecho, se aplica la función *predict* de la librería Keras a varias subimágenes para que el modelo trate de determinar a que clase cree que pertenecen:

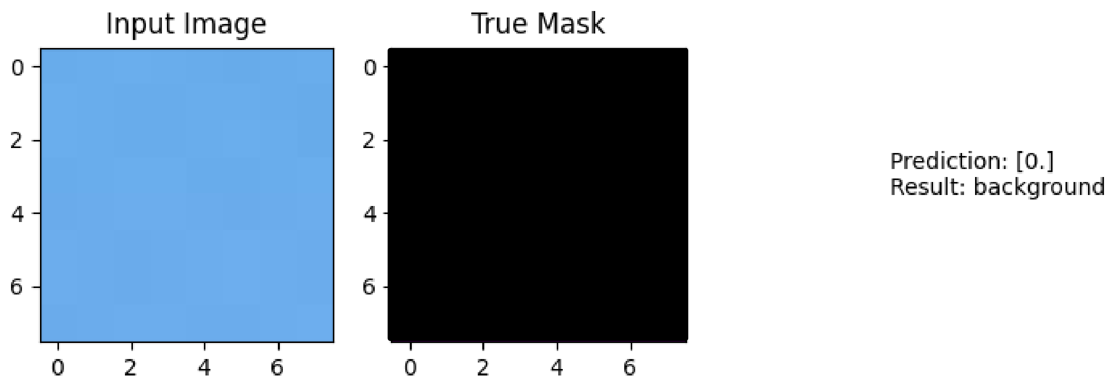


Figura 35: Resultados en subimágenes 8x8 píxeles. Modelo 1

En el ejemplo de la ilustración superior se puede comprobar la imagen de entrada que se introduce al modelo, la máscara que le pertenece y la predicción que se obtiene: clase 0 (*background*). Al comprobar varias predicciones similares, se puede dar por hecho que el modelo funciona correctamente.

La última parte consistirá en construir la máscara completa de las imágenes originales. Para ello, después de haber descompuesto la imagen en cuadrados, recorriendola como un filtro de *pooling*, esas subimágenes se guardan en una lista por orden y se van pasando una a una al modelo para que prediga su clase. Paralelamente, en una matriz vacía de la misma dimensión que la imagen original, se asigna el valor de la clase en la misma posición donde se encuentra la subimagen en la imagen original. Por ejemplo, si la primera subimagen de clase *background* es el cuadrado 8x8 en la esquina superior izquierda, se asignará valor 0 a todo el cuadrado 8x8 en la misma posición de la matriz.

De esta manera, el resultado obtenido es el siguiente (primera imagen es la imagen original, la segunda imagen es la máscara real y la tercera imagen es la máscara que la red predice):

Figura 36: Resultado imagen original. Modelo 1

3.4.2 Segundo modelo: red neuronal de segmentación

Se sube un nivel la complejidad de modelos de red neuronal con uno de segmentación. En este caso, se busca que el propio modelo sea el que prediga la máscara de la imagen de entrada y sea capaz de construirla en su salida. Se requiere utilizar una arquitectura de red especial que permita generar imágenes como salida, y esta ventaja la va a ofrecer el modelo **U-Net**.

U-NET es un modelo de red neuronal dedicado a tareas de visión artificial y, más concretamente, a problemas de segmentación semántica. Consta de dos partes, de las cuales la primera es la de la contracción, conocida también como **codificador,** conformada por un conjunto de capas de convolución y *maxpooling* que se encargan de crear un mapa de características de una imagen y reducir su tamaño para optimizar el número de parámetros en la red. La segunda parte es la vía de expansión, también **decodificador**, formada esta vez por capas deconvolucionales, que ejecutan la operación de convolución traspuesta, y capas de concatenación, que permiten combinar características extraídas del codificador con las del decodificador, permitiendo aprovechar tanto la información local (detalles específicos) como la información global (características de alto nivel) para realizar tareas como segmentación de imágenes. Se va a estudiar con más detalle el funcionamiento de este modelo a continuación [*Sección 3.4.2.3*].

3.4.2.1 Dataset

Al disponer de un conjunto de datos pequeño, como en el caso de las 12 imágenes del primer modelo, existe un mayor riesgo de sobreajuste, fenómeno que ocurre cuando el modelo se ajusta demasiado a los datos de entrenamiento específicos y no logra capturar patrones generales que se puedan aplicar a datos nuevos. El modelo memorizaría cada imagen individualmente y el rendimiento de predicción del modelo sería deficiente. Como solución, se escoge aumentar el *dataset* y aplicar técnicas de aumento de datos y *dropout*.

Para este aumento de datos, se seleccionaron imágenes de ámbito personal y se trató se ajustar los umbrales de color con la librería *OpenCV* para sacar las máscaras. Se disponía de unas 160 imágenes aproximadamente de las que luego hubo que descartar varias por la imprecisión de los resultados en sus máscaras, con lo que finalmente se trabajó con un *dataset* de 116 imágenes.

El *dataset* se vuelve a distribuir de manera similar al primer modelo, en porciones de datos para entrenamiento, validación y testeo. Para la fase *training*, se asignan 95 imágenes, para el grupo de *validation* se asignan 10 imágenes y para *testing,* las 11 restantes. La estructura de directorios se organiza de manera parecida al primer modelo, con una pequeña variación, ya que añadimos un nuevo directorio, llamado *aug,* donde vamos a guardar las transformaciones de las imágenes a las que el generador de *ImageDataGenerator* aplique *Data Augmentation*, para visualizar el correcto funcionamiento de la técnica.

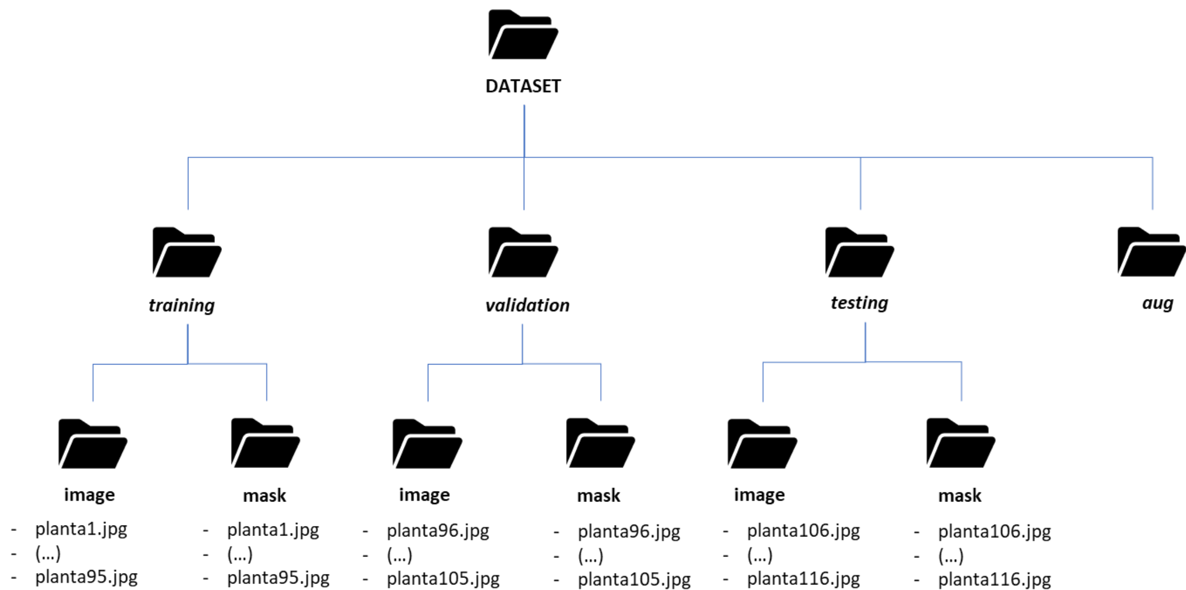


Figura 37: Estructura del conjunto de datos. Modelo 2

3.4.2.2 Preprocesado de datos

Para el preprocesamiento de datos se utiliza también la clase *ImageDataGenerator* para instanciar generadores que regulen la entrada de datos en la red. Se construye un generador para las imágenes y otro para las máscaras tanto para los datos de *training* como los de *validation*, que hacen un total de 4 generadores. Se muestra la construcción de los generadores para *training*, primero definiendo las transformaciones que se van a aplicar en la construcción de los objetos:

```
data_aug_args = dict(rotation_range=0.2,
                     width_shift_range=0.2,
                     height_shift_range=0.2,
                     shear_range=0.2,
                     zoom_range=0.2,
                     horizontal_flip=True,
                     vertical_flip=True)

batch_size = 2

image_datagen = ImageDataGenerator(rescale=1./255, **data_aug_args)
mask_datagen = ImageDataGenerator(rescale=1./255,
                                  **data_aug_args)
```

Código 13: Se instancian dos objetos ImageDataGenerator con sus argumentos de transformación de imágenes. Modelo 2

Al aplicar las mismas transformaciones en las imágenes y las máscaras, garantizamos que ambas se modifican de la misma manera y se mantienen alineadas. Esto evita situaciones en las que la máscara podría definir una segmentación errónea de la imagen a la que corresponde.

Ahora, se utiliza la función *flow_from_directory* de la clase *ImageDataGenerator* que permite generar automáticamente lotes de datos de imágenes y sus etiquetas asociadas a partir de un directorio de archivos de imágenes organizado en subdirectorios. El directorio de archivos es el *training* y los subdirectorios son el de image y el de mask.

```python
image_generator = image_datagen.flow_from_directory(
        #train_dir,
        "/content/drive/MyDrive/Colab-Notebooks/dataset3/training",
        classes = ['image'],
        class_mode = None,
        color_mode = 'rgb',
        target_size = (256,256),
        batch_size = batch_size,
        save_to_dir = "/content/drive/MyDrive/Colab-Notebooks/dataset3/aug",
        save_prefix  = "img",
        seed = 1)


mask_generator = mask_datagen.flow_from_directory(
        #train_dir,
        "/content/drive/MyDrive/Colab-Notebooks/dataset3/training",
        classes = ['mask'],
        class_mode = None,
        color_mode = 'grayscale',
        target_size = (256,256),
        batch_size = batch_size,
        save_to_dir = "/content/drive/MyDrive/Colab-Notebooks/dataset3/aug",
        save_prefix  = "mask",
        seed = 1)
```

Código 14: Implementación de los generadores de lotes de datos. Modelo 2

La razón por la que se usa dos generadores en lugar de uno solo para las imágenes y las máscaras es porque manejan distintos tipos de datos. En este caso, el espacio de color cambia la dimensión de los tensores: para las imágenes son tensores de *(256, 256, 3)* y para las máscaras son de *(256, 256, 1)*.

Algunos puntos para destacar son el del tamaño de las imágenes, que van a configurarse automáticamente a una dimensión de 256x256 (en las primeras pruebas se hacía esto de manera manual con ayuda de *OpenCV*). Esto supone una reducción de la dimensión de las imágenes que, a pesar de la reducción de detalles, va a proporcionar mayor eficiencia computacional, ahorro de espacio de almacenamiento y aumento de la velocidad de entrenamiento. También se define el espacio de color de cada tipo de datos, la ruta donde se van a almacenar las imágenes de entrada tras aplicarle el Data Augmentation, y los lotes van a ser de 2 imágenes.

Finalmente, se combinan ambos generadores mediante la función *zip* incorporada en Python, y devuelve un nuevo generador. Este generador va a generar tuplas, donde cada tupla contiene un lote de imágenes y su correspondiente lote de máscaras, que van a asegurar que cada imagen en el lote generado por *image_generator* esté asociada con su respectiva máscara en el lote generado por *mask_generator*.

```python
train_generator = zip(image_generator, mask_generator)
```

Código 15: Combinación generadores. Modelo 2

3.4.2.3 Modelo

Para la implementación de este modelo, nos hemos apoyado en el diseño de un proyecto de GitHub de segmentación de membranas celulares, del usuario zhixuhao, que se puede acceder desde la URL: https://github.com/zhixuhao/unet.

Este diseño se apoya en un tipo de arquitectura de red neuronal convolucional conocido como U-Net, muy utilizada para tareas de segmentación semántica de imágenes. Fue propuesta por Olaf Ronneberger, Philipp

Fischer y Thomas Brox en 2015. Su nombre deriva de la forma característica de esta arquitectura, en forma de "U", que se puede ver a continuación.

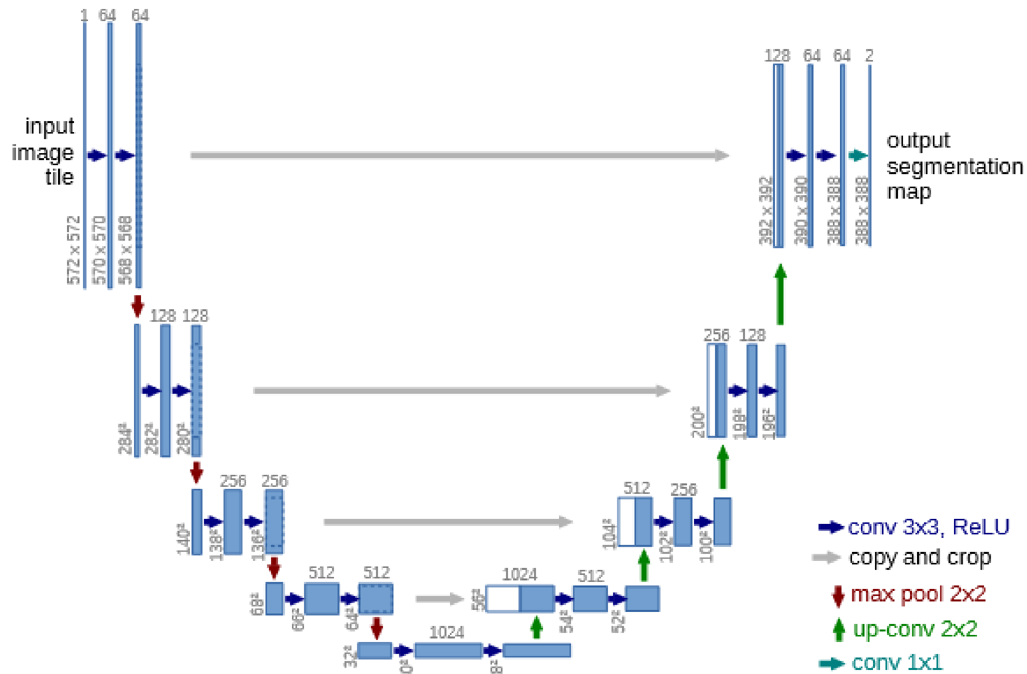


Figura 38: Arquitectura U-Net [27]

Esta red se creó para segmentar y etiquetar imágenes automáticamente. A diferencia de otras arquitecturas de red, la de U-Net no tiene capas densamente conectadas. Utiliza:

- **Capas convolucionales**, sin padding, que reduce, en la salida de cada capa, la dimensión de las imágenes en dos filas y dos columnas. A cada nivel se aplica 2 filtros convolucionales seguidos de tamaño 3x3 y con activación *relu.*

- **Capas de convolución traspuesta**, que se diferencian de las primeras en que aumentan el tamaño de los datos, y se implementan mediante *upsampling*, que dobla el tamaño del mapa de características y reduce el tamaño de canales (filtros) a la mitad.

La arquitectura de la red utiliza una estructura *encoder-decoder*, donde el *encoder* sigue la arquitectura de una red convolucional, encargandose de las operaciones de convolución (parte izquierda) para la captura de características con diferentes niveles de abstracción, y el *decoder* realiza las operaciones de deconvolución, que define la ruta de expansión para la segmentación de las características.

El ***encoder*** es la ruta de contracción, que consiste en la aplicación repetida de dos convoluciones 3x3 (convoluciones sin padding) seguidas de la función de activación relu, y una operación de MaxPooling 2x2 con *stride* igual a 2. En cada paso, se dobla el número de filtros. En cuanto al ***decoder,*** esta ruta expansiva consiste en una convolución "ascendente" que va reduciendo a la mitad el número de filtros y aumentando la resolución de las características, y la **capa de concatenación**, que toma las características del mismo nivel del *encoder* y las fusiona con las obtenidas en el nivel correspondiente del *decoder*. La operación de concatenación es esencial en la U-Net porque permite combinar información contextual de niveles anteriores con detalles de alta resolución en niveles posteriores.

La última capa de la red neuronal utiliza una convolución 1x1. Esta capa se utiliza para transformar el vector de características de 64 componentes en un vector de salida con el número deseado de clases, utilizando para cada píxel una codificación ***one-hot***. A diferencia de las convoluciones típicas que se utilizan para extraer características, la convolución 1x1 se utiliza principalmente para cambiar la dimensión del canal de los datos.

En total, el modelo usa 23 capas de convolución [27].

La implementación del diseño con Keras es el siguiente:

```
inputs = Input((256,256,3))
conv1 = Conv2D(64, 3, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(inputs)
conv1 = Conv2D(64, 3, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(conv1)
pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
conv2 = Conv2D(128, 3, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(pool1)
conv2 = Conv2D(128, 3, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(conv2)
pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
conv3 = Conv2D(256, 3, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(pool2)
conv3 = Conv2D(256, 3, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(conv3)
pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)
conv4 = Conv2D(512, 3, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(pool3)
conv4 = Conv2D(512, 3, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(conv4)
drop4 = Dropout(0.5)(conv4)
pool4 = MaxPooling2D(pool_size=(2, 2))(drop4)

conv5 = Conv2D(1024, 3, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(pool4)
conv5 = Conv2D(1024, 3, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(conv5)
drop5 = Dropout(0.5)(conv5)

up6 = Conv2D(512, 2, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(drop5))
merge6 = concatenate([drop4,up6], axis = 3)
conv6 = Conv2D(512, 3, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(merge6)
conv6 = Conv2D(512, 3, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(conv6)

up7 = Conv2D(256, 2, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(conv6))
merge7 = concatenate([conv3,up7], axis = 3)
conv7 = Conv2D(256, 3, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(merge7)
conv7 = Conv2D(256, 3, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(conv7)

up8 = Conv2D(128, 2, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(conv7))
merge8 = concatenate([conv2,up8], axis = 3)
conv8 = Conv2D(128, 3, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(merge8)
conv8 = Conv2D(128, 3, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(conv8)
up9 = Conv2D(64, 2, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(conv8))
merge9 = concatenate([conv1,up9], axis = 3)
conv9 = Conv2D(64, 3, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(merge9)
conv9 = Conv2D(64, 3, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(conv9)
conv9 = Conv2D(2, 3, activation = 'relu', padding = 'same', kernel_initializer = 'he_normal')(conv9)
conv10 = Conv2D(1, 1, activation = 'sigmoid')(conv9)

model = Model(inputs = inputs, outputs = conv10)
```

Código 16: Estructura de capas. Modelo 2

También se aplican dos capas *Dropout* como técnica de regularización para prevenir posible sobreajuste, con una tasa de *0.5* cada una.

Se aplican como hiperparámetros los mismos que en el primer modelo: un optimizador Adam con un *learning rate* de *1e-4*, una función de pérdida *binary_crossentropy* y como métrica la *accuracy*. Por último, el entrenamiento se ejecuta en 5 *epochs*.

3.4.2.4 Testing

Esta vez, para los datos de *testing* se utiliza también un generador construido desde la librería *ImageDataGenerator,* que va a facilitar mucho a nivel de código la preparación de los datos, ya que ejecuta el rescale de las imágenes, las redimensiona, las transforma en tensores y las asigna en lotes de datos. Luego, simplemente con la función *predict_generator* de Keras, recibe como argumento el generador y devuelve todas las predicciones en la matriz *predictions*, que representa la máscara de cada imagen.

```
test_generator = test_datagen.flow_from_directory(
        "/content/drive/MyDrive/Colab-Notebooks/dataset3/test",
        classes = ['images'],
        target_size=(256, 256),
        batch_size=2,
        color_mode='rgb',
        class_mode=None,
        shuffle=False)

predictions = model.predict_generator(test_generator, steps=len(test_generator), verbose=1)
```

Código 17: Implementación de generador de lotes de datos para testing. Modelo 2

Finalmente, del testing se obtienen resultados como este, donde la imagen de la izquierda es la original, la del centro es la máscara obtenida de *OpenCV* que le corresponde y a su derecha está la máscara que el modelo predice:

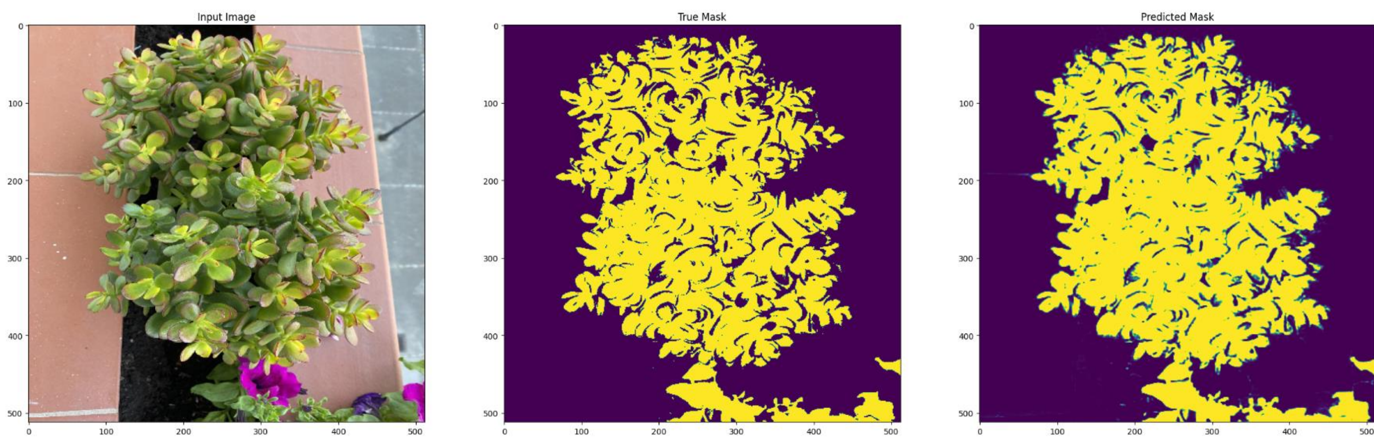


Figura 39: Resultado (1). Modelo 2

4 EVALUACIÓN

Tras haber realizado un profundo análisis de los procedimientos implementados para abordar el desafío de la segmentación, es hora de evaluar el rendimiento de ambos modelos y verificar su eficacia. Para ello, se va a llevar a cabo una serie de evaluaciones que determinen su buen funcionamiento.

- De cada modelo se va a analizar visualmente los resultados obtenidos y compararlos con las máscaras con las que se entrenan la red para evaluar la calidad y precisión de las predicciones, y buscar posibles mejoras.
- Se van a presentar las curvas de aprendizaje del entrenamiento de cada modelo, para comprobar si aprende correctamente, si disminuye la función de pérdida y si aumenta el accuracy, o si por el contrario hubiera sobreajuste.
- Vamos a hacer una evaluación de ciertos parámetros de predicción: *precisión, recall, f1-score, support* (*sección 2.2.7*).
- Por último, también analizaremos la matriz de confusión de cada modelo (*sección 2.2.7*)

4.1 Primer modelo

4.1.1 Parámetros iniciales

En la siguiente table se recogen los parámetros utilizados para el entrenamiento de la red:

Tabla 1: Hiperparámetros configurados del modelo 1

Parámetro	*Valor*
Dimensión de entrada	8x8x3
Tamaño del dataset	120.000 (64 px. cada imagen)
Tamaño del batch	64
Epochs	20
Optimizador	Adam
Learning Rate	1×10^{-3} (por defecto)
Steps per epoch	*1562*

El modelo al que se le aplican estos parámetros es el descrito en la sección *3.4.1*.

4.1.2 Resultados

Se va a presentar en primer lugar los resultados obtenidos del entrenamiento del modelo. Se recuerda que el modelo maneja este tipo de dimensiones debido a que se divide la imagen principal en pequeñas subimágenes

de 8x8, y estas son las que el modelo analiza y clasifica para determinar si contienen hojas o no. La clasificación de subimágenes genera una nueva máscara en la cual cada zona de tamaño 8x8 se etiqueta como blanca si contiene una hoja y como negra si no contiene una hoja, generando la máscara de la imagen original.

El modelo lo entrenamos con 12 imágenes del mismo almendro, con lo cual el entrenamiento es bastante específico. Por ejemplo, si introducimos esta nueva imagen del almendro:

Figura 40: Imagen planta (2)

Y comparamos la máscara que generamos con la librería *OpenCV,* y la máscara que sacamos de las subimágenes que predice el primer modelo:

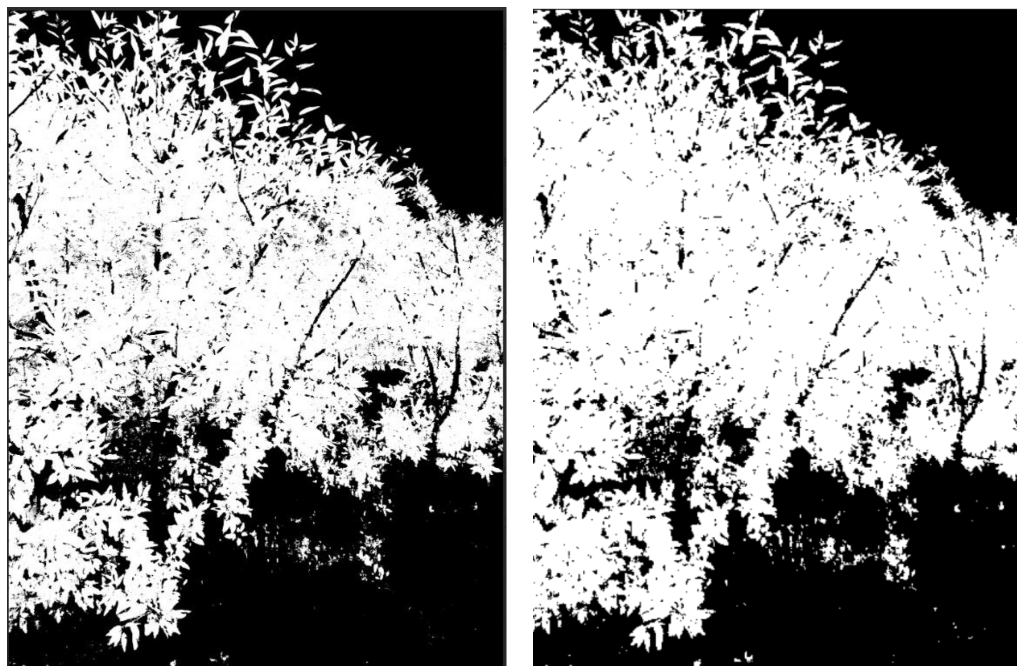


Figura 41: Comparación máscara original (izquierda) y máscara modelo 1 (derecha)

Se puede comprobar que la técnica funciona, aunque es menos precisa que la máscara de *OpenCV.* Se podría esperar esto ya que a la mínima que un cuadrado 8x8 contiene parte de una hoja, aunque sea un píxel, ya se va a pintar en cuadrado entero, y eso va a llevar a que el blanco este más presente en la máscara del modelo, y aparente un efecto más borroso al no ser tan detallista. Pero se acerca mucho, y si se aplicara para subimágenes 4x4 la precisión sería visualmente mayor.

Se hablaba antes de la poca generalización de los patrones que tomaría este modelo debido a que su entrenamiento es muy específico. Se va a comprobar el resultado que se obtiene si introducimos imágenes de planta que se salen del patrón. Un ejemplo:

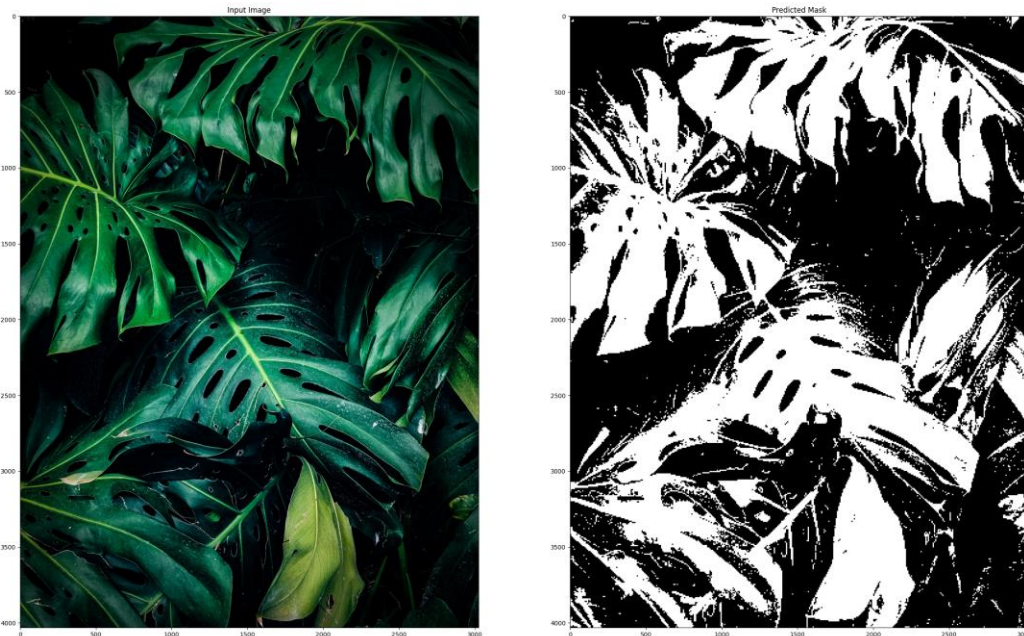


Figura 42: Imagen original y resultado máscara modelo 1 (1)

Se puede comprobar en esta figura, como el modelo tiene dificultad para detectar los tonos de verde más oscuro, notándose más la imprecisión, aunque con más acercamiento del esperado en esta última prueba. Cuando el modelo se entrenó con menos *epochs*, el resultado apenas reflejaba partes blancas. Segundo ejemplo:

Figura 43: Imagen original y resultado máscara modelo 1 (2)

En esta figura se puede comprobar la imprecisión que nos da la técnica que usamos en este modelo, y es que, al sacar las máscaras de subimágenes tan pequeñas, aquello que segmenta se localiza por su color, no por su forma, lo que va a originar que la red neuronal cuando entrene con estas etiquetas aprenda a localizar el color, y cuando se genere la máscara total, las zonas de interés se elijan por el color. Esta desventaja se puede comprobar con esta imagen con fondo verde, donde el resultado es una imprecisión visual alta por el umbral de color que maneja el modelo.

4.1.3 Entrenamiento

A continuación, se presentan los resultados del entrenamiento de la red. Para comenzar, se muestra como ha evolucionado la tasa de acierto (*accuracy*) a lo largo del entrenamiento, y también se va a comprobar la función de coste, que muestra la evolución de la *loss*, tanto para el conjunto de datos de *training* como para el de *validation*.

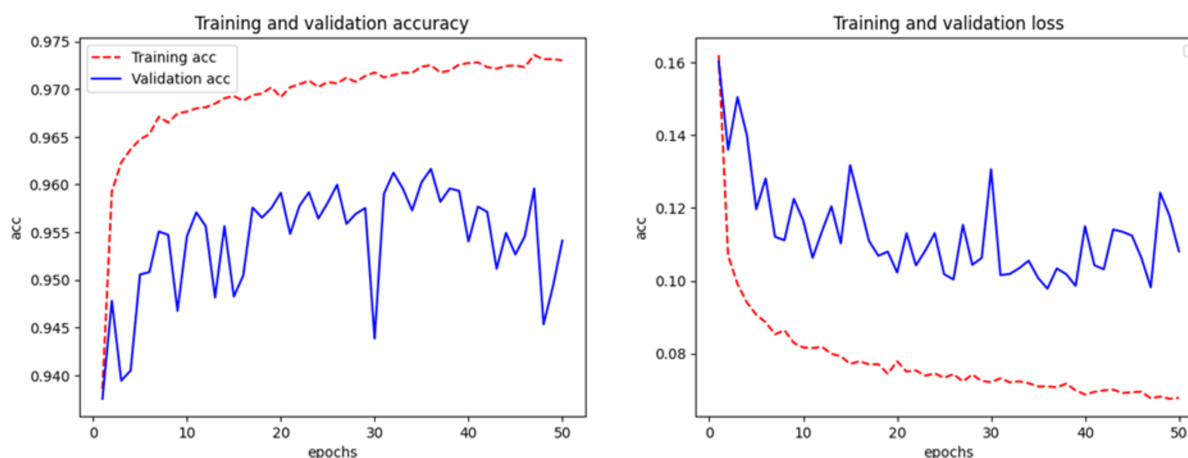


Figura 44: Tasa de acierto (izquierda) y función de coste (derecha) modelo 1

De estas gráficas se pueden sacar conclusiones como:

1. En primer lugar, la tasa de acierto tanto en el entrenamiento como en la validación empieza en un valor muy alto, rondando el 94%, y luego en ambos tiene tendencia decreciente, en el entrenamiento de manera más constante que en la curva de validación. Los resultados del *accuracy* en ambos casos es muy alto, llegando a estabilizarse la curva de aprendizaje en el 97% y la curva de validación en el 95%.

2. Para la función de pérdida, similar, pero de manera contraria. Valores muy bajos en ambos casos, rondando el 16% nada más terminar la primera *epoch*, y luego en tendencia decreciente, más variable en el caso de la validación. En la curva de entrenamiento se llega a estar por debajo del 1% desde muy pronto, la época 3 aproximadamente, y la de validación acaba llegando y estabilizándose alrededor del 1%.

3. En general, las gráficas muestran resultados muy conseguidos y eficientes. El modelo aparenta estar aprendiendo con bastante precisión.

Como se habló al inicio del punto 4, también se va a profundizar un poco en las métricas de predicción, que como es obvio, cuanto más cerca estén del 1, será una indicación de que mejor es el modelo.

Tabla 2: Métricas del entrenamiento del modelo 1

Métrica	*Valor*
Accuracy	0. 9197234793455022
Precisión	0. 90076027826721305
Recall	0. 9622782012185593
F1-score	0. 93038319172673765

Por lo tanto, tenemos que hay aproximadamente un 92% de datos clasificados correctamente (ya sea por verdadero positivo o negativo), un 90% de predicciones positivas correctas respecto al total de predicciones positivas y un 96% de positivas correctas en relación con el total de positivas reales.

La métrica F1-score es la única de la que no se ha hablado, y consiste en una media armónica de la precisión y el recall, y se define de la siguiente manera: F1-score = 2 * (precision * recall) / (precision + recall)

$$F1_{score} = \frac{2 \times (precision \times recall)}{(precision + recall)}$$

Finalmente, se obtiene la matriz de confusión:

$$\begin{array}{cc} 12.419.809 & 1.890.512 \\ 666.335 & 16.873.840 \end{array}$$

La suma de los valores es igual al número de pixeles de las dos imágenes que componen el conjunto de datos de *testing*. De la matriz de confusión nos interesa que los valores en diagonal sean los más altos posibles, para que nos muestren los Verdaderos positivos o los verdaderos negativos, es decir, el número de datos que ha sido clasificado correctamente. En la siguiente imagen se puede comprobar lo que representa cada zona de la matriz, como ya hablamos en la *sección 2.2.7* [*Figura 45*].

		Predicción	
		Positivos	**Negativos**
Observación	**Positivos**	Verdaderos Positivos (VP)	Falsos Negativos (FN)
	Negativos	Falsos Positivos (FP)	Verdaderos Negativos (VN)

Figura 45: Representación de las zonas de la matriz de confusión [17]

4.2 Segundo modelo

4.2.1 Parámetros iniciales

En la siguiente table se recogen los parámetros utilizados para el entrenamiento de la red:

Tabla 3: Hiperparámetros configurados del modelo 2

Parámetro	*Valor*
Dimensión de entrada	256x256x3
Tamaño del dataset	116
Tamaño del batch	2
Epochs	100
Optimizador	Adam
Learning Rate	1×10^{-4}

Steps per epoch	*47*

El modelo al que se le aplican estos parámetros es el descrito en la sección *3.4.2*.

4.2.2 Resultados

Este modelo de red neuronal de segmentación se basa en la arquitectura U-Net, y es lo que va a permitir que la red pueda predecir directamente la máscara de segmentación de una imagen, en lugar de predecir etiquetas de clase individuales. La red genera una matriz binaria como salida que se traduce en una máscara que define la presencia o ausencia de una región de interés en cada píxel de la imagen de entrada. Tras recordar su funcionamiento, se procede a analizar algunos resultados obtenidos.

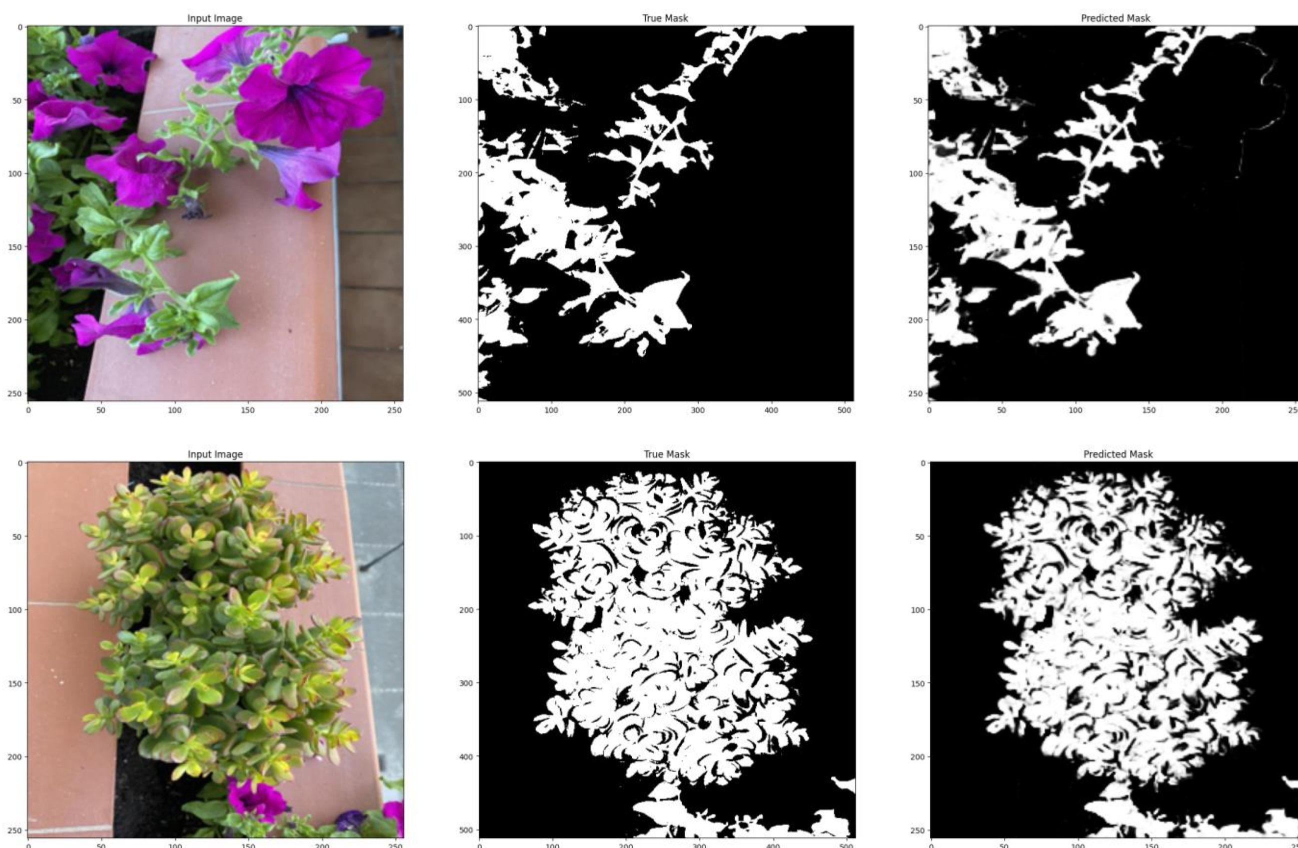


Figura 46: Imagen original (izquierda), máscara original (centro), máscara resultado modelo 2 (1)

Las máscaras centrales son las de *OpenCV* y las de la derecha son las máscaras que la red neuronal resuelve. Se puede comprobar que la máscara se acerca bastante a la original de *OpenCV*. A medida que se ha ido probando a aumentar el número de *epochs*, más precisión ha tenido. De la misma manera ha pasado al aumentar el numero de *steps per epoch*. Al visualizar más casos, como la figura de abajo, se puede notar incluso una mejora de la máscara en comparación con la original, al menos en ese caso concreto, donde el umbral de *OpenCV* no ha llegado a reconocer ese color en ningún momento en ningún dato de entrenamiento, y aún así la red lo destaca en el resultado.

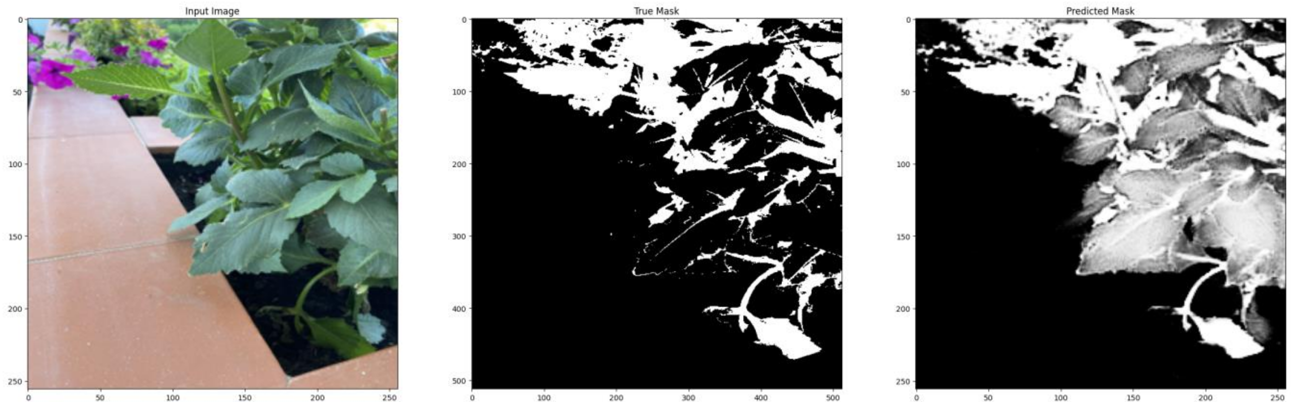


Figura 47: Imagen original (izquierda), máscara original (centro), máscara resultado modelo 2 (2)

4.2.3 Entrenamiento

Analizando las mismas gráficas que en el caso anterior:

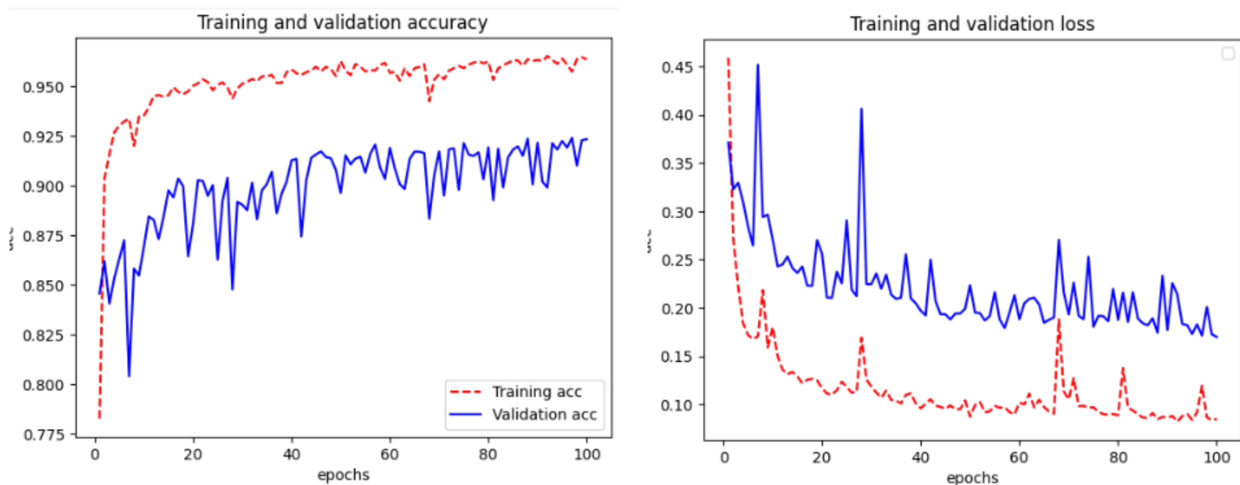


Figura 48: Tasa de acierto (izquierda) y función de coste (derecha) modelo 2

1. La tasa de acierto en la primera *epoch* ronda casi el 80% para los datos de entrenamiento, y ya en la tercera supera el 90%. Da la sensación de un resultado muy bueno, a pesar de no haber sido apenas entrenado.

2. En cuanto a la validación, comienza más fuerte que los resultados de entrenamiento, aunque pronto es superada, pero se mantiene a partir de la época 18 aproximadamente en el valor de 90%, más o menos estable y con picos menos diferenciados a medida que pasan las épocas.

3. En cuanto a la función *loss,* el comportamiento es bastante similar en términos opuestos. Para el entrenamiento decrece muy rápido en las primeras *epochs,* y se llega a mantener en un 10% a partir de la *epoch* 30 aproximadamente. La función de validación da el mismo pico de empeoramiento en la *epoch* 5, coincidiendo con su pico más bajo en la métrica *accuracy*, y luego se estabiliza entre un 25 – 20% durante el resto del proceso, y siempre con tendencia decreciente.

El modelo puede ser mejorable si se le aumentaran más las épocas. En general se ven resultados eficientes. En cuanto a las métricas de predicción:

Tabla 4: Métricas del entrenamiento del modelo 2

Métrica	*Valor*
Accuracy	0.8981623649597168
Precisión	0.999148624839987
Recall	0.8377301243891189
F1-score	0.9102654570721229

Comparando los valores con el modelo anterior, impacta el alto nivel de precisión, con un 99% de acierto en la clasificación de muestras positivas sobre el total de muestras clasificadas como positivas. En cuanto al f1-score, muestra un buen equilibrio entre precisión y sensibilidad, con un 91%.

Finalmente, analizando la matriz de confusión:

$$\begin{matrix} 336036 & 1163 \\ 59198 & 324499 \end{matrix}$$

Nos muestra que:

- Ha clasificado 336.036 muestras que fueron clasificadas correctamente como positivas.
- Ha clasificado 1.163 muestras que deberían haber sido clasificadas como positivas, pero fueron clasificadas de manera errónea como negativas.
- Ha clasificado 59.198 muestras que deberían haber sido clasificadas como negativas, pero fueron clasificadas de manera errónea como positivas.
- Y, por último, 324.499 muestras que fueron clasificadas correctamente como negativas.

5 CONCLUSIÓN

Como conclusión de este trabajo, se puede afirmar que se han cumplido los objetivos que se establecieron, y lo que empezó siendo un trabajo de implementar una red neuronal convolucional de clasificación, acabó ampliándose a un trabajo que implementa dos modelos de red con el objetivo de segmentar unas imágenes de plantas.

Como se mencionó al comienzo del documento, este trabajo no tiene otro fin que el de profundizar todo lo posible en las técnicas de Deep Learning. Se busca que este trabajo proporcione una oportunidad para iniciarse en el campo del aprendizaje profundo para quién le pueda despertar cierto interés esta área.

En cuanto a las tareas planteadas, se ha conseguido que ambos modelos aprendan el cometido que se les ha propuesto, tanto en la clasificación de imágenes como en la segmentación, para obtener de los dos resultados similares y con un buen porcentaje de precisión (al menos así lo demuestra la visualización de los resultados y sus métricas).

Por resumir un poco lo realizado:

- Se han aplicado unos umbrales de color adecuados a lo que se necesitaba segmentar, a partir de la librería de *OpenCV*. Esto ha permitido que las *labels* de los modelos hayan ayudado a ajustar bien los pesos de las redes en su proceso de entrenamiento.
- Para la generación de los conjuntos de datos, la clase *ImageDataGenerator* de la librería Keras ha sido crucial para facilitar la tarea de preprocesamiento y generación de los lotes de datos. Además, ha agilizado considerablemente la implementación de la técnica de aumento de datos, lo cual era necesario debido al tamaño tan reducido del conjunto de datos disponible.
- También, se ha podido aplicar la técnica de Dropout para evitar *overfitting* y una mayor generalización en el aprendizaje de las dos redes.
- Se ha sabido, con el paso del tiempo y de la implementación de modelos fallidos, los hiperparámetros que usar, para qué se aplica su uso y cómo funcionan. Se ha sabido elegir un optimizador y una función *loss* acordes al modelo y el problema, y variar el número de *batches* y *epochs* para optimizar el entrenamiento.
- Se ha conseguido una evaluación correcta de los resultados, profundizando con el uso de las gráficas de tasa de acierto y tasa de error, y las matrices de confusión y los parámetros que se calculan a partir de esta.

También hay ciertas cosas que mejorar, que podrían ser un *dataset* más amplio y que abarque más tipos de planta y árbol. Esto podría llevar a que un modelo fuera capaz de reconocer el tipo de planta, para líneas futuras. Otra cosa que mejorar podría serla segmentación de las *labels*, que al segmentarse por color daba ciertos problemas con otros objetos que fueran verdes y no fueran hojas, y su precisión, aunque es buena, puede mejorar.

REFERENCIAS

[1] J. A. Sanchez Somolinos, «Introducción a la visión por computador,» de *Avances en robótica y visión por computador*, Cuenca, Ediciones de la Universidad de Castilla-La Mancha, 2002.

[2] R. Rizo Aldeguer, P. Arques Corrales y M. Pujol López, «Modelo de segmentación y caracterización (Tesis Doctoral),» Universidad de Alicante, Alicante, 2007.

[3] S. J. Rusell y P. Norvig, «Artificial Intelligence: A Modern Approach,» Prentice Hall, 1995, pp. 1 - 29.

[4] I. Pérez Borrero y M. E. Gegúndez Arias, «Redes Neuronales: Introducción,» de *Deep learning*, Servicio de Publicaciones de la Universidad de Huelva, 2021, pp. 25 - 28.

[5] R. Valbuena, Inteligencia Artificial: Investigación Científica Avanzada Centrada en Datos, vol. 2, Cencal Press, 2021.

[6] F. Chollet, «Artificial intelligence, machine learning, deep learning,» de *Deep Learning with Python*, Shelter Island, NY, Manning Publications Co., 2018.

[7] J. Torres, «Introducción: Machine Learning,» de *Deep Learning: Introducción práctica con Keras (PRIMERA PARTE)*, Barcelona, Colección WATCH THIS SPACE, 2018.

[8] Y. Xin, L. Kong, Z. Liu, C. Yuling, Y. Li y H. Hou, «Machine Learning and Deep Learning Methods for Cybersecurity,» *IEEE,* vol. 6, nº 17905844, pp. 35365 - 35381, 15 May 2018.

[9] R. Flores López y J. M. Fernández Fernández, de *Las Redes Neuronales Artificiales: Fundamentos teóricos y aplicaciones prácticas*, Universidad de León, Netbiblo SL, 2008, pp. 16 - 21.

[10] F. Chollet, «A first look at a neural network,» de *Deep Learning with Python*, Shelter Island, NY, Manning Publications Co., 2018.

[11] J. Torres, «Introducción: Terminología básica de Machine Learning,» de *Deep Learning: Introducción práctica con Keras (PRIMERA PARTE)*, Barcelona, Colección WATCH THIS SPACE, 2018.

[12] M. Nielsen, «Using neural nets to recognise handwritten digits,» de *Neural Networks and Deep Learning*, 2006, pp. 1 - 12.

[13] SPC Group, «SPC Consulting Group,» 17 Julio 2015. [En línea]. Available: https://spcgroup.com.mx/diagrama-de-dispersion/.

[14] J. Torres, «Redes neuronales convolucionales,» de *Deep Learning: Introducción práctica con Keras (PRIMERA PARTE)*, Barcelona, Colección "Watch this space", 2018.

[15] I. Pérez Borrero y M. E. Gegúndez Arias, «Redes Neuronales: Aplicaciones,» de *Deep learning*, Servicio de Publicaciones de la Universidad de Huelva, 2021, pp. 76 - 79.

[16] J. Torres, «Cómo se entrena una red neuronal,» de *Deep Learning: Introducción práctica con Keras*

(PRIMERA PARTE), Barcelona, Colección WATCH THIS SPACE, 2018.

[17] J. Torres, «Redes neuronales densamente conectadas,» de *Deep Learning: Introducción a prácticas en Keras (PRIMERA PARTE)*, Barcelona, Colección "Watch this space", 2018.

[18] Longongas, «Entrenamiento de redes neuronales,» 4 Octubre 2023. [En línea]. Available: https://logongas.es/doku.php?id=clase:iabd:pia:2eval:tema07.

[19] DotCSV, «¿Qué es el Descenso del Gradiente? Algoritmo de Inteligencia Artificial,» Youtube, 4 Febrero 2018. [En línea]. Available: https://www.youtube.com/watch?v=A6FiCDoz8_4&list=LL&index=1&t=2s&ab_channel=DotCSV.

[20] R. Tech, «Redes Neuronales Convolucionales - Clasificación avanzada de imágenes con IA / ML (CNN),» Youtube, 16 Agosto 2021. [En línea]. Available: https://www.youtube.com/watch?v=4sWhhQwHqug&t=1038s&ab_channel=RingaTech.

[21] J. Torres, «Data Augmentation y Transfer Learning,» de *Deep Learning, Introducción práctica con Keras (SEGUNDA PARTE)*, Barcelona, Watch this space, 2019.

[22] Universidad de Málaga, «Deep Learning con GPUs,» Master en Big Data e IA, 23 Febrero 2023. [En línea]. Available: https://www.bigdata.uma.es/modulo-7-deep-learning-con-gpus/.

[23] NVIDIA, «Computación acelerada,» [En línea]. Available: https://www.nvidia.com/es-la/drivers/what-is-gpu-computing/.

[24] G. Sharma, «Top 3 ways to write your Tensorflow Code,» 22 Julio 2021. [En línea]. Available: https://www.analyticsvidhya.com/blog/2021/07/top-3-ways-to-write-your-tensorflow-code/.

[25] doxygen, «OpenCV-Python Is Now An Official OpenCV Project,» 14 Febrero 2021. [En línea]. Available: https://opencv.org/blog/2021/02/14/opencv-python-is-now-an-official-opencv-project/.

[26] Proyectos Wikimedia, «Modelo de color HSV,» Wikipedia, 18 Diciembre 2022. [En línea]. Available: https://es.wikipedia.org/wiki/Modelo_de_color_HSV.

[27] doxygen, «Open Source Computer Vision,» Google, 9 Julio 2023. [En línea]. Available: https://docs.opencv.org/4.x/d9/d61/tutorial_py_morphological_ops.html.

[28] O. Ronneberger, P. Fisher y B. Thomas, «U-Net: Convolutional Networks for Biomedical,» Computer Science Department and BIOSS Centre for Biological Signalling Studies,, University of Freiburg, Germany, 2015.

ANEXO

En esta sección se recoge el código de todas las funciones adicionales que han sido necesarias para el correcto funcionamiento de ambos modelos de segmentación. El código que hace referencia a los modelos, la preparación del *dataset* y las respectivas explicaciones del proceso, se puede encontrar en la sección 3 de este trabajo.

Función que obtiene las máscaras para el entrenamiento con la librería OpenCV

```
import os
import cv2
import numpy as np

def detect_leaf(file_name, images_directory, norm_directory, masks_directory,
                res_directory, i):

      # Obtenemos la imagen a partir del directorio de imagenes y el nombre del archivo
      img_ruta = os.path.join(images_directory, file_name)
      img = cv2.imread(img_ruta)

      # Redimensionar las imágenes
      img = cv2.resize(img,(256,256)) #IMG_SIZE
      cv2.imwrite(os.path.join(norm_directory,"planta"+str(i)+".jpg"), img)

      # Convertir de BGR a HSV
      hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
      # Definicion umbrales color verde
      lower_green=np.array([29,40,40])
      upper_green=np.array([100,255,255])

      # Contruir una máscara basada en el umbral para construir una imagen en blanco y negro.
      mask = cv2.inRange(hsv, lower_green, upper_green)

      # Transformaciones morfológicas
      # Si filename acaba en '-O', se ejecuta erosion seguido de dilatacion en la mascara
      if file_name [-6] == '-' and file_name[-5] == 'O':
        kernel = np.ones((2,2),np.uint8)
        mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
      # Si filename acaba en '-E', se ejecuta la operacion erosion en la mascara
      if file_name [-6] == '-' and file_name[-5] == 'E':
        kernel = np.ones((3,3),np.uint8)
        mask = cv2.erode(mask,kernel,iterations = 1)
```

```
# Operación AND bit a bit entre la imagen y la máscara generada para segmentar color verde
res = cv2.bitwise_and(img, img, mask=mask)
#Guardar mask en la carpeta de mascaras con el mismo nombre de la imagen original
cv2.imwrite(os.path.join(masks_directory,file_name), mask)
#Guardar el resultado en la carpeta de imagenes segmentadas con el mismo nombre de la imagen original
cv2.imwrite(os.path.join(res_directory,"planta"+str(i)+".jpg"), res)
```

Estructura de directorios y organización del *dataset* para entrenar la red

```
import os, shutil

root_dir = "/content/drive/MyDrive/Colab-Notebooks/"

base_dir = os.path.join(root_dir,'almendros_dataset')
shutil.rmtree(base_dir)
os.mkdir(base_dir)

train_dir = os.path.join(base_dir,'train')
os.mkdir(train_dir)
valid_dir = os.path.join(base_dir,'validation')
os.mkdir(valid_dir)
test_dir = os.path.join(base_dir,'test')
os.mkdir(test_dir)

train_image_dir = os.path.join(train_dir,'images')
os.mkdir(train_image_dir)
train_mask_dir = os.path.join(train_dir,'masks')
os.mkdir(train_mask_dir)

valid_image_dir = os.path.join(valid_dir,'images')
os.mkdir(valid_image_dir)
valid_mask_dir = os.path.join(valid_dir,'masks')
os.mkdir(valid_mask_dir)

test_image_dir = os.path.join(test_dir,'images')
os.mkdir(test_image_dir)
test_mask_dir = os.path.join(test_dir,'masks')
os.mkdir(test_mask_dir)

fnames = ['planta{}.jpg'.format(i+1) for i in range(8)]
for fname in fnames:
    src1 = os.path.join(os.path.join(root_dir,'fotos_almendros'),fname)
    dst1 = os.path.join(train_image_dir, fname)
    src2 = os.path.join(os.path.join(root_dir,'masks2'),fname)
    dst2 = os.path.join(train_mask_dir, fname)
    shutil.copyfile(src1, dst1)
    shutil.copyfile(src2, dst2)
```

```
fnames = ['planta{}.jpg'.format(i+1) for i in range(8, 10)]
for fname in fnames:
    src1 = os.path.join(os.path.join(root_dir,'fotos_almendros'),fname)
    dst1 = os.path.join(valid_image_dir, fname)
    src2 = os.path.join(os.path.join(root_dir,'masks2'),fname)
    dst2 = os.path.join(valid_mask_dir, fname)
    shutil.copyfile(src1, dst1)
    shutil.copyfile(src2, dst2)

fnames = ['planta{}.jpg'.format(i+1) for i in range(10, 12)]
for fname in fnames:
    src1 = os.path.join(os.path.join(root_dir,'fotos_almendros'),fname)
    dst1 = os.path.join(test_image_dir, fname)
    src2 = os.path.join(os.path.join(root_dir,'masks2'),fname)
    dst2 = os.path.join(test_mask_dir, fname)
    shutil.copyfile(src1, dst1)
    shutil.copyfile(src2, dst2)

print('Total training images: ', len(os.listdir(train_image_dir)))
print('Total validation images: ', len(os.listdir(valid_image_dir)))
print('Total test images: ', len(os.listdir(test_image_dir)))
```

Modelo 1. Función que divide las imágenes en subimágenes de 8x8 píxeles

```
from PIL import Image
import numpy as np

def division_imagen(root_directory):

    '''Coge las imagenes originales y las máscaras de la carpeta que se envía
    como argumento, las divide en subimágenes de h x w, comprueba con la ayuda
    de la mascara si las imagenes contienen hojas o no, y se guarda en su
    respectiva lista dependiendo de esa condición (background_images si son
    imagenes que no contienen hojas y leaf_images si viceversa)'''

    images_directory = os.path.join(root_directory, "images")
    masks_directory = os.path.join(root_directory, "masks")
    images_filenames = os.listdir(images_directory)

    leaf_images = []
    background_images = []

    for filename in images_filenames:
        img = Image.open(os.path.join(images_directory, filename))
        mask = Image.open(os.path.join(masks_directory, filename))
        imgwidth, imgheight = img.size
        h = DIM_SQUARE
        w = DIM_SQUARE
```

```
        for i in range(0,imgheight,h):
            for j in range(0,imgwidth,w):
                box = (j, i, j+w, i+h)
                img_cut = img.crop(box)
                mask_cut = mask.crop(box)

                ''' Si la mascara contiene alguna parte de hoja (1), se
                introduce la imagen en la lista leaf_images, si la mascara
                no contiene hojas (todo en negro) se añade la imagen a la
                lista de background_images.'''

                # Vamos a comprobar si la máscara contiene partes de hojas
                matriz_px = np.array(mask_cut)

                aux = 0
                for px in np.nditer(matriz_px):
                    if px/255 == 1:
                        # Contiene hojas
                        leaf_images.append(np.array(img_cut))
                        aux = 1
                        break

                    else:
                        continue
                    break
                if aux == 0:
                    # NO contiene hojas
                    background_images.append(np.array(img_cut))

    return leaf_images, background_images
```

Modelo 1. Preparación del *dataset* [*sección 3.4.1.1*]

```
import random

# Sacamos las listas de las subimagenes
train_leaf_images, train_background_images = division_imagen(train_dir)
valid_leaf_images, valid_background_images = division_imagen(valid_dir)

print("En la parte de entrenamiento, hay "+ str(len(train_leaf_images)) +
      " ejemplos de hojas y " + str(len(train_background_images)) +
      " de background")

print("En la parte de validacion, hay "+ str(len(valid_leaf_images)) +
      " ejemplos de hojas y " + str(len(valid_background_images)) +
      " de background")
```

```
# Vamos a coger un número de imágenes más pequeño para el entrenamiento

''' Cogemos 50k ejemplos con hojas y 50k ejemplos de background
de manera aleatoria para el entrenamiento de la red '''

train_leaf_samples = random.SystemRandom().sample(train_leaf_images, 50000)
train_background_samples = random.SystemRandom().sample(train_background_images, 50000)
valid_leaf_samples = random.SystemRandom().sample(valid_leaf_images, 10000)
valid_background_samples = random.SystemRandom().sample(valid_background_images, 10000)

# Cargamos las subimágenes con hojas y las de background en una sola lista
train_samples = train_leaf_samples + train_background_samples
valid_samples = valid_leaf_samples + valid_background_samples

# Creamos las etiquetas para cada imagen
train_leaf_labels = np.ones(len(train_leaf_samples))
train_background_labels = np.zeros(len(train_background_samples))
train_labels = np.concatenate([train_leaf_labels, train_background_labels])

valid_leaf_labels = np.ones(len(valid_leaf_samples))
valid_background_labels = np.zeros(len(valid_background_samples))
valid_labels = np.concatenate([valid_leaf_labels, valid_background_labels])
```

Modelo 1. *Testing*: subdivisión de imágenes en cuadrados de 8x8 píxeles, el modelo clasifica cada imagen y se genera la mascara

```
def divide_into_squares(img, square_size):
    squares = []
    imgwidth, imgheight = img.size
    h = square_size
    w = square_size

    for i in range(0,imgheight,h):
        for j in range(0,imgwidth,w):
            box = (j, i, j+w, i+h)
            img_cut = img.crop(box)
            squares.append(img_cut)

    return squares
```

```
# Metemos en una lista todas las imagenes de testeo del modelo
test_filenames = os.listdir(test_image_dir)
test_img_list = [Image.open(os.path.join(test_image_dir, filename)) for filename in test_filenames]
#test_mask_list = [Image.open(os.path.join(test_mask_dir, filename)) for filename in test_filenames]
guarda_masks_predict = []
for j, img in enumerate(test_img_list):

    # Definimos una imagen vacia para la nueva mascara
    imgwidth, imgheight = img.size
    mask = np.zeros((imgheight, imgwidth))

    # División de imagen en subimágenes 8x8
    squares = divide_into_squares(img, square_size=DIM_SQUARE)

    # El modelo las clasifica
    square_array = np.stack([np.array(square)/255.0 for square in squares])
    predictions = model.predict(square_array)
    num_cols = imgwidth/DIM_SQUARE

    # Según clasificación, cuadrado 8x8 a 1 o 0 en la nueva mascara
    for i, prediction in enumerate(predictions):
        if prediction >= 0.5:
            classes = np.ones((DIM_SQUARE,DIM_SQUARE))
        else:
            classes = np.zeros((DIM_SQUARE,DIM_SQUARE))
        row = i // int(num_cols)
        col = i % int(num_cols)

        # Ordenar las subimágenes para la nueva mascara
        mask[row*DIM_SQUARE:(row+1)*DIM_SQUARE, col*DIM_SQUARE:(col+1)*DIM_SQUARE] += (classes * 255).astype(int)

    guarda_masks_predict.append(mask)
    #fig, (ax1, ax2, ax3) = plt.subplots(1,3, figsize = (30, 30))
    fig, (ax1, ax2) = plt.subplots(1,2, figsize = (30, 30))

    ax1.set_title("Input Image")
    ax1.imshow(test_img_list[np.array(j)])

    '''ax2.set_title("True Mask")
    ax2.imshow(test_mask_list[np.array(j)])'''

    ax2.set_title("Predicted Mask")
    plt.imshow(np.uint8(mask), cmap='gray')
    plt.show()
```

Obtención de métricas y matriz de confusión

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix

# Cargar las mascaras en listas
true_mask = [Image.open(os.path.join(test_mask_dir, filename)) for filename in test_filenames]
predicted_mask = guarda_masks_predict

for i in range(2):
      # Convertir las imágenes de máscara en array NumPy
      true_mask_array = np.array(true_mask[i])
      predicted_mask_array = predicted_mask[i]

      # Binarizar las máscaras con umbral 0.5
      true_mask_binary = (true_mask_array >= 0.5).astype(int)
      predicted_mask_binary = (predicted_mask_array >= 0.5).astype(int)

      # Aplanar las matrices de máscara
      true_mask_flat = true_mask_binary.flatten()
      predicted_mask_flat = predicted_mask_binary.flatten()
      print(true_mask_flat)
      print(predicted_mask_flat)

      # Calcular las métricas de clasificación binaria
      accuracy = accuracy_score(true_mask_flat, predicted_mask_flat)
      precision = precision_score(true_mask_flat, predicted_mask_flat)
      recall = recall_score(true_mask_flat, predicted_mask_flat)
      f1 = f1_score(true_mask_flat, predicted_mask_flat)


      # Calcular la matriz de confusión
      confusion = confusion_matrix(true_mask_flat, predicted_mask_flat)

      # Imprimir las métricas y la matriz de confusión
      print("Accuracy:", accuracy)
      print("Precision:", precision)
      print("Recall:", recall)
      print("F1-score:", f1)
      print("Confusion matrix:")
      print(confusion)
```

Modelo 2. *Testing* con un generador de *ImageDataGenerator*

```
from matplotlib import pyplot as plt

test_filenames = os.listdir(test_image_dir)
test_img_list = [Image.open(os.path.join(test_image_dir, filename)).resize((256,256)) for filename in test_filenames]
test_mask_list = [Image.open(os.path.join(test_mask_dir, filename)) for filename in test_filenames]

test_datagen = ImageDataGenerator(rescale=1./255)
test_generator = test_datagen.flow_from_directory(
        "/content/drive/MyDrive/Colab-Notebooks/dataset3/test",
        classes = ['images'],
        target_size=(256, 256),
        batch_size=2,
        color_mode='rgb',
        class_mode=None,
        shuffle=False)

predictions = model.predict_generator(test_generator, steps=len(test_generator), verbose=1)

guarda_masks_predict = []

for j, prediction in enumerate(predictions):

    guarda_masks_predict.append(prediction)

    fig, (ax1, ax2, ax3) = plt.subplots(1,3, figsize = (30, 30))

    ax1.set_title("Input Image")
    ax1.imshow(test_img_list[np.array(j)])

    ax2.set_title("True Mask")
    ax2.imshow(test_mask_list[np.array(j)], cmap='gray')

    ax3.set_title("Predicted Mask")
    plt.imshow(prediction, cmap='gray')
    plt.show()
```