



viu

**Universidad
Internacional
de Valencia**

Desarrollo de un sistema de diagnóstico de enfermedades en hojas de tomate mediante modelos de aprendizaje profundo

Titulación:
Máster en Big Data y
Ciencia de Datos
Curso Académico
2024-2025

Alumno/a: Marín Lucas,
Rubén
DNI: 07272889-J
Director/a del TFT:
Ricardo Lebrón Aguilar

Convocatoria:
SEGUNDA

Índice general

Índice de figuras	3
Índice de cuadros	3
1. Introducción	8
2. Objetivos	13
2.1. Objetivos específicos	13
3. Estado del arte	14
4. Implementación y desarrollo	16
4.1. Herramientas usadas	16
4.2. Procedencia y descripción de los datos	18
4.3. Preprocesado de los datos	18
4.4. Modelado	23
4.4.1. Modelo <i>MobileNetV2</i> 1	25
4.4.2. Modelo <i>MobileNetV2</i> 2	26
4.4.3. Modelo <i>MobileNetV2</i> 3	27
4.4.4. Modelo <i>EfficientNetB0</i> 1	27
4.4.5. Modelo <i>EfficientNetB0</i> 2	28
4.4.6. Modelo <i>EfficientNetB0</i> 3	29
4.4.7. Modelo <i>NASNetMobile</i> 1	30
4.4.8. Modelo <i>NASNetMobile</i> 2	30
4.5. Arquitecturas	31
4.5.1. <i>MobileNetV2</i>	31
4.5.2. <i>EfficientNetB0</i>	32
4.5.3. <i>NASNetMobile</i>	33
5. Evaluación y resultados	36
5.1. Modelo <i>MobileNetV2</i> 1	36
5.1.1. Entrenamiento 1	36
5.1.2. Entrenamiento 2	39
5.1.3. Entrenamiento 3	42
5.2. Modelo <i>MobileNetV2</i> 2	46
5.3. Modelo <i>MobileNetV2</i> 3	46

5.4. Modelo <i>EfficientNetB0</i> 1	46
5.5. Modelo <i>EfficientNetB0</i> 2	47
5.6. Modelo <i>EfficientNetB0</i> 3	48
5.7. Modelo <i>NASNetMobile</i> 1	49
5.8. Modelo <i>NASNetMobile</i> 2	51
6. Conclusiones	52
A. Anexo I: Ejemplo de anexo	53

Índice de figuras

1.1. Origen del tomate	8
1.2. Tizón tardío en una planta de tomate	10
1.3. Tizón precoz en una planta de tomate	10
1.4. Mancha bacteriana en una planta de tomate	11
1.5. Picadura de la araña roja de dos manchas en una planta de tomate	11
1.6. Virus de la hoja amarilla en una planta de tomate	12
4.1. Imagen aleatoria por clase	20
4.2. Resumen del modelo <i>MobileNetV2 1</i>	25
4.3. Resumen del modelo <i>MobileNetV2 2</i>	26
4.4. Resumen del modelo <i>MobileNetV2 3</i>	27
4.5. Resumen del modelo <i>EfficientNetB0 1</i>	28
4.6. Resumen del modelo <i>EfficientNetB0 2</i>	29
4.7. Resumen del modelo <i>EfficientNetB0 3</i>	29
4.8. Resumen del modelo <i>NASNetMobile 1</i>	30
4.9. Resumen del modelo <i>NASNetMobile 2</i>	31
5.1. Histórico de métricas del entrenamiento 1 del modelo <i>MobileNetV2 1</i>	37
5.2. Resumen de métricas del modelo <i>MobileNetV2 1</i> en el entrenamiento 1	38
5.3. Matriz de confusión del modelo <i>MobileNetV2 1</i> en el entrenamiento 1	39
5.4. Histórico de métricas del entrenamiento 2 del modelo <i>MobileNetV2 1</i>	40
5.5. Resumen de métricas del modelo <i>MobileNetV2 1</i> en el entrenamiento 2	41
5.6. Matriz de confusión del modelo <i>MobileNetV2 1</i> en el entrenamiento 2	42
5.7. Histórico de métricas del entrenamiento 3 del modelo <i>MobileNetV2 1</i>	43
5.8. Resumen de métricas del modelo <i>MobileNetV2 1</i> en el entrenamiento 3	44
5.9. Matriz de confusión del modelo <i>MobileNetV2 1</i> en el entrenamiento 3	45
5.10. Histórico de métricas del del modelo <i>EfficientNetB0 1</i>	46
5.11. Histórico de métricas del entrenamiento del modelo <i>EfficientNetB0 2</i>	47
5.12. Histórico de métricas del entrenamiento del modelo <i>EfficientNetB0 3</i>	49
5.13. Histórico de métricas del entrenamiento 1 del modelo <i>NASNetMobile</i>	50

Índice de cuadros

1.1.	<u>Top 10 países productores de tomates 2022</u>	9
4.1.	<u>Clases del conjunto de datos</u>	18
4.2.	<u>Los 5 tamaños de imágenes más comunes</u>	19
4.3.	<u>Número de imágenes por clases en el conjunto de entrenamiento</u>	22
4.4.	<u>Pesos asignados a cada clase en el entrenamiento del modelo</u>	23
4.5.	<u>División funcional de MobileNetV2</u>	33
4.6.	<u>División funcional de EfficientNetB0</u>	34
4.7.	<u>División funcional de NASNetMobile</u>	35

Resumen

Palabras clave: primero, segundo, tercero

Agradecimientos

 Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

1. Introducción

Tomate o tomatera (*Solanum lycopersicum*) es una planta herbácea de la familia Solanaceae cultivada en todo el mundo para el cultivo de su fruto, el tomate o jitomate, uno de los ingredientes más universales de ensaladas y salsas en el mundo entero. (Wikipedia contributors, s.f.-b)

Según los últimos estudios filogenéticos, la planta silvestre de la cual surge el tomate doméstico actual tiene origen en la zona andina del norte de Perú y sur de Ecuador. Su domesticación y diversificación posterior se originó en México. Los pueblos aztecas y mayas lo usaban en su cocina y fue exportado al resto del mundo a partir de la llegada de los españoles que lo distribuyeron a lo largo de sus colonias en el Caribe y la península ibérica a partir de lo cual pudo llegar al resto de Europa. También lo llevaron a Filipinas y de allí pudo entrar al continente asiático. (Silva, s.f.-a)

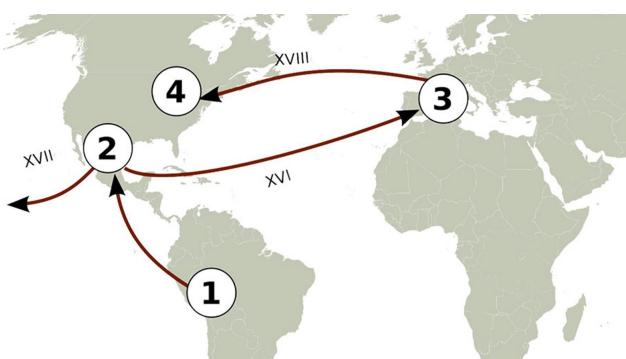


Figura 1.1.: Origen del tomate

La producción mundial de tomate ascendió a más de 186 millones de toneladas en 2022 según los datos de la Organización de las Naciones Unidas para la Alimentación y la Agricultura (FAO). Según esta misma organización la evolución de los 10 países que más han producido hasta 2022 es la mostrada en el Cuadro 1.1. (Wikipedia contributors, s.f.-a)

Como ya se ha mencionado, el cultivo de tomate es uno de los cultivos hortícolas

Cuadro 1.1.: Top 10 países productores de tomates 2022

Top 10 países productores de tomates 2022				
País	2000	2010	2020	2022
China	22 200	46 760	64 680	68 242
India	7430	12 433	20 550	20 694
Turquía	8890	10 052	13 204	13 000
Estados Unidos	12 622	14 053	10 939	10 200
Egipto	6786	8545	6494	6275
Italia	7538	6025	6248	6136
México	2666	2998	4137	4208
Brasil	3005	4107	3757	3810
Nigeria	1261	1800	3390	3685
España	3766	4313	4313	3652

más importantes a nivel mundial. Sin embargo, su producción se ve amenazada por una amplia variedad de enfermedades causadas por hongos, bacterias, virus y nematodos. Estas enfermedades pueden provocar una bajada de rendimiento que van desde reducciones parciales hasta la pérdida completa de la cosecha.

Entre las enfermedades más comunes se encuentran:

- Tizón tardío (*Late blight*): El agente causal es el oomiceto *Phytophthora infestans* que afecta a las plantas solanáceas. Se desarrolla mejor en condiciones de humedad alta, lluvias, rocío persistente o días nublados, y temperaturas frías (entre 10° y 20°). Su ciclo incluye esporangios, zoosporas (en condiciones húmedas) y puede sobrevivir mediante estructuras resistentes (oosporas) o restos vegetales infectados (Editors of Encyclopedia Britannica, 2024). Las primeras lesiones aparecen típicamente en hojas o tallos donde aparecen manchas de aspecto mojado que luego se tornan a manchas oscuras e irregulares. Bajo condiciones muy húmedas se observa una capa blanca o algodonosa de esporulación en el envés de las hojas u otras partes (Gevens & Wilbur, 2014) como se puede observar en la Figura 1.2.
- Tizón precoz (*Early blight*): El agente causal principal es el hongo *Alternaria solani* (y en algunos casos *Alternaria tomatophila*) que puede sobrevivir en residuos de plantas enfermas, en el suelo, sobre plantas de la familia de las solanáceas y también transmitirse por semillas o plántulas infectadas (McGrath, Margaret, s.f.). Las condiciones favorables para su desarrollo incluyen: temperaturas cálidas (por encima de 25°), humedad elevada o períodos de hojas mojadas (Pacific Northwest contributors, 2025b). Los primeros síntomas sue-



Figura 1.2.: Tizón tardío en una planta de tomate

len aparecer en las hojas más viejas, situadas en la base de la planta, aparecen manchas oscuras que a menudo presentan anillos concéntricos. Alrededor de esas manchas la hoja puede ponerse amarilla (clorosis) y luego necrosarse, lo que deriva en caída de hojas o defoliación parcial del vegetal (Baysal-Gurel, Fulya and Miller, Sally, 2010) como puede observarse en la Figura 1.3.



Figura 1.3.: Tizón precoz en una planta de tomate

- Mancha bacteriana (*Bacterial spot*): Es causada por bacterias del género *Xanthomonas* que atacan al tomate. El patógeno puede sobrevivir en residuos vegetales, en semillas contaminadas, en plantas voluntarias o en malas hierbas de la familia de las solanáceas (Pacific Northwest contributors, 2025a). Las condiciones favorables incluyen alta humedad y temperaturas templadas o calientes (20 °C o más). En las hojas aparecen manchas húmedas, luego se tornan oscuras, ásperas y con margen amarillo en ocasiones. Las manchas suelen aparecer primero en hojas viejas o en el dosel inferior de la planta, desde donde pueden subir (Davis et al., s.f.) como muestra la Figura 1.4.
- Picadura de la araña roja de dos manchas (*Twospotted spider mite*): El agente



Figura 1.4.: Mancha bacteriana en una planta de tomate

es la araña roja de dos manchas (*Tetranychus urticae*) un ácaro (ni hongo, ni bacteria) que ataca muchas hortalizas incluyendo el tomate. Se alimenta mediante piezas bucales que perforan las células de las hojas (generalmente en el envés) y succionan el contenido, provocando daño celular, pérdida de clorofila y debilitamiento de la planta. Su reproducción es rápida, especialmente bajo condiciones de calor y sequedad, prefiere hojas viejas, la parte baja de la planta, y condiciones de ambiente seco, polvoso, con buena radiación y temperaturas elevadas. En las hojas aparecen pequeñas manchas puntiformes, amarillas o blanquecinas, que resultan del daño celular tipo "picoteado"(*stippling*) como se muestra en la Figura 1.5. (Hazzard, 2022)



Figura 1.5.: Picadura de la araña roja de dos manchas en una planta de tomate

- Virus de la hoja amarilla (*Yellow leaf curl virus*): El agente causal es un virus los begomovirus, que afecta principalmente al tomate y ocasionalmente otras solanáceas. El vector principal es la mosca blanca, *Bemisia tabaci*, que adquiere el virus al alimentarse de plantas infectadas y luego lo transmite a plantas sanas. Las hojas muestran claramente: amarilleamiento entre las venas (clorosis interveinal), márgenes de la hoja amarillos, hojas enrolladas o curvadas hacia arriba y hacia dentro, y foliolos más pequeños de lo normal como muestra la Figura 1.6. (Díaz-Pendón et al., 2010)



Figura 1.6.: Virus de la hoja amarilla en una planta de tomate

La manifestación simultánea o sucesiva de estas enfermedades es una de las principales causas en la disminución en la productividad del cultivo a escala global. Además, muchas de estas enfermedades no solo viven en la planta sino que persisten en el suelo, semillas o herramientas que hayan interactuado con la planta, lo que dificulta su erradicación y aumenta los costos del tratamiento. (Silva, s.f.-b)

Dada la magnitud del impacto de estas enfermedades, la detección temprana y precisa de las mismas es crucial. Permite una correcta intervención que minimiza las pérdidas, permitiendo la reducción del uso innecesario de los agroquímicos y mejorando la sostenibilidad. En este contexto, las tecnologías basadas en visión por computadora, sensores remotos e inteligencia artificial ofrecen soluciones eficaces para mejorar el seguimiento y el control sanitario de este cultivo clave.

2. Objetivos

El objetivo general de este proyecto consiste en desarrollar y evaluar un sistema inteligente basado en técnicas de aprendizaje profundo capaz de identificar el estado de salud y las principales enfermedades en hojas de tomate a partir de imágenes, asegurando un alto grado de precisión y capacidad de generalización.

2.1. Objetivos específicos

1. Revisar el estado del arte relacionado con la detección de enfermedades en plantas, con especial énfasis en el cultivo del tomate.
2. Analizar y preparar el conjunto de datos aplicando las técnicas necesarias de preprocesamiento de imágenes para garantizar la calidad y homogeneidad del material de entrenamiento.
3. Implementar y entrenar distintos modelos utilizando arquitecturas preentrenadas con alguna base de datos para adaptarlas a la tarea de clasificación de enfermedades en hojas de tomate.
4. Evaluar los modelos desarrollados comparando su rendimiento mediante métricas como precisión, sensibilidad o capacidad de generalización.

3. Estado del arte

En los últimos años la aplicación de técnicas de inteligencia artificial en la agricultura ha cobrado un papel relevante, especialmente en tareas de diagnóstico temprano de enfermedades en cultivos. El uso de aprendizaje profundo permite automatizar la detección de patrones en imágenes, lo cual puede ayudar a los agricultores a tomar decisiones más rápidas y eficientes.

Inicialmente, el enfoque tradicional para las tareas de clasificación de imágenes se basaba en características diseñadas manualmente como SIFT (Lowe, 2004), HoG (Dalal & Triggs, 2005), SURF (Bay et al., 2008), etc, para luego usar algún algoritmo de aprendizaje supervisado como máquinas de vectores de soporte (SVM), kvecinos más cercanos (KNN) y redes bayesianas. Sin embargo, estos enfoques dependían en gran medida de las características predefinidas. La ingeniería de características en sí misma es un proceso complejo y laborioso que debe revisarse cada vez que el problema en cuestión o el conjunto de datos asociado cambia considerablemente. Este problema se da en todos los intentos tradicionales de detectar enfermedades de las plantas , ya que se basabann en gran medida en características diseñadas manualmente, técnicas de mejora de imágenes y otras metodologías complejas.

Además, los enfoques tradicionales para la clasificación de enfermedades mediante ML suelen centrarse en un número reducido de clases, normalmente dentro de un mismo cultivo. Por ejemplo, se han usado imágenes térmicas y estéreo para detectar el moho polvoriento (*powdery mildew*) en hojas de tomate (Raza et al., 2015), imágenes RGBD para detección del tizón del manzano (*apple scab*) (Chéné et al., 2012) y sensores basados en aeronaves para la detección del virus del rizado amarillo del tomate (*tomato yellow leaf curl*) mediante el uso de un conjunto de pasos clásicos de extracción de características (Garcia-Ruiz et al., 2012).

Posteriormente han aparecido estudios que usan entrenamiento supervisado de extremo a extremo utilizando una arquitectura de red neuronal convolucional (CNN) como (Krizhevsky et al., 2012) que demostraron que este tipo de arquitecturas tienen un buen desempeño incluos en problemas de clasificación de imágenes con un número de clases, superando con creces a los enfoques tradicionales. El hecho de

no depender de la laboriosa fase de la ingeniería de características, además de la generalización que alcanza convierte este enfoque en un candidato prometedor para la detección computacional de enfermedades en plantas. (Mohanty et al., 2016)

Para aplicaciones de diagnóstico de enfermedades en plantas es habitual seleccionar arquitecturas que equilibren precisión y coste computacional. Arquitecturas ligeras como MobileNetV2 y NASNetMobile han demostrado ser adecuadas para sistemas móviles por su bajo número de parámetros y alta velocidad de inferencia, manteniendo precisiones comparables a modelos más pesados tras *fine tuning* (Lu et al., 2023). Por su parte, EfficientNet-B0, diseñado mediante *compound scaling*, ofrece una eficiencia de parámetros superior y suele vencer a tradicionales en relación precisión y tamaño (Atila et al., 2021). Estas características hacen que MobileNetV2, EfficientNet-B0 y NASNetMobile sean opciones sólidas para aplicaciones prácticas de diagnóstico en tomate, especialmente cuando el despliegue en dispositivos con recursos limitados.

El uso de modelos preentrenados en bases de datos extensas como *ImageNet* constituye una práctica consolidada en tareas de clasificación de enfermedades foliares, dado que permiten aprovechar representaciones visuales previamente aprendidas (bordes, texturas, patrones) para iniciar el entrenamiento en un dominio específico con menor cantidad de datos. Estudios recientes demuestran que el uso de *transfer learning* a partir de arquitecturas entrenadas sobre *ImageNet* acelera la convergencia, reduciendo el número de épocas (*epochs*), cantidad de datos y riesgo de sobreajuste (*overfitting*) (Al Sahili & Awad, 2022). No obstante, también se advierte que la diferencia de dominio entre *Imagenet* y las imágenes reales de hojas puede afectar a la generalización, lo cual resalta la importancia de una augmentación adecuada, ajustes de hiperparámetros y validación con imágenes de campo (Dong et al., 2023).

En este trabajo se emplean modelos basados en las arquitecturas MobileNetV2, EfficientNetB0 y NASNetMobile preentrenados con ImageNet como punto de partida, lo que permite centrar el esfuerzo en la adaptación al problema de enfermedades de tomate mediante fine-tuning y validación rigurosa.

4. Implementación y desarrollo

En este capítulo se presenta tanto el *hardware* como el *software* usados en este proyecto. Además se explica la procedencia y estructura del conjunto de datos que serán usados para el estudio. Finalmente, se desarrolla el preprocesamiento que se realiza a este conjunto de datos junto con los modelos entrenados para conseguir un clasificador.

4.1. Herramientas usadas

Para llevar a cabo este proyecto, se ha usado Google Colab (abreviatura de Google Colaboratory) que se accedia desde el ordenador portátil del autor del documento. Este ordenador es un ASUS TUF Gaming FX505GT que cuenta con las siguientes características:

- 16 GB de RAM con formato DDR4.
- Almacenamiento compuesto por un disco duro con tecnología SSD de 512GB.
- Procesador Intel Core i7-9750H CPU a 2.60 GHz, con 6 procesadores principales y 6 procesadores lógicos.
- Tarjeta gráfica NVIDIA GeForce GTX 1650 con 4GB de RAM.

Google Colab es un servicio gratuito de Google que permite escribir y ejecutar código en la nube sin necesidad de instalar nada en tu equipo. Los recursos que ofrece de forma gratuita varían con el tiempo, pero las características que suele ofrecer son las siguientes:

- GPU NVIDIA Tesla T4 con 16 GB de VRAM ó CPU Intel Xeon con alrededor de 13 GB de RAM.
- Almacenamiento temporal se corresponde con unos 100 GB de espacio en disco.

- La duración de la sesión puede ser de hasta 12 horas, aunque en la práctica podrían terminarse antes según uso y carga del sistema.

Por otra parte el lenguaje de programación usado ha sido Python, un lenguaje que es ampliamente utilizado por científicos de datos. En las últimas décadas Python se ha enriquecido con numerosas librerías relacionadas con técnicas de ML que facilitan el uso de las mismas. En concreto para este proyecto se ha utilizado la versión 3.12.11 de Python.

En cuanto a las librerías de Python usadas para la implementación, se presentan a continuación:

- NumPy: es una librería que ofrece la posibilidad de crear matrices y vectores multidimensionales y provee además un gran número de operaciones matemáticas de alto nivel.
- Pandas: es una librería que ofrece la estructura de datos llamada DataFrame que facilita la manipulación y el análisis de datos. Es una extensión de la librería NumPy. Ha sido usada para tratar y transformar los datos.
- Plantcv: es una librería de Python de código abierto diseñada específicamente para el análisis de imágenes de plantas. Se ha usado para lectura de las imágenes.
- Tensorflow: es una librería de software de código abierto creada por Google para desarrollar y entrenar modelos de machine learning (ML) y deep learning (DL). Recibe este nombre porque trabaja con tensores, estructuras de datos multidimensionales, como matrices o vectores que fluyen a través de un grafo computacional de operaciones. Se ha usado para crear y entrenar los modelos descritos en este proyecto.
- Seaborn: esta librería permite la visualización de los datos a través de distintos tipos de gráficas. Ha sido usada para realizar los distintos gráficos como las matrices de confusión para evaluar los modelos.
- Sklearn: es una librería que contiene un gran número funciones relacionadas con modelos de machine learning (ML) y deep learning (DL). Se ha usado para extraer las principales métricas de los modelos tras su entrenamiento.

4.2. Procedencia y descripción de los datos

Los datos provienen de la plataforma online de ciencia de datos de Google, Kaggle, que funciona como una mezcla de red social, repositorio de datasets y espacio de competición. En concreto, el conjunto de datos usado es el llamado "Tomato Leaves Dataset". Según su descripción en la misma plataforma se trata de un conjunto de datos de más de 20.000 imágenes de hojas de tomate con 11 clases, 10 enfermedades y una clase sana. Estas imágenes se han recopilado tanto en entornos de laboratorio como en entornos naturales. (Motwani & Khan, 2025)

En concreto se pueden extraer dos directorios que servirán como conjunto de datos para el entrenamiento y conjunto de datos para validación, ambos cuentan con 11 directorios con imágenes dentro. Cada uno de estos subdirectorios representa una de las clases que serán brevemente expuestas a continuación:

Cuadro 4.1.: Clases del conjunto de datos

Conjunto de datos	
Clase	Traducción Clase
Healthy	Saludable
Bacterial_spot	Manchas bacterianas
Early_blight	Tizón precoz
Late_blight	Tizón tardío
Leaf_Mold	Hojas con moho
Powdery_mildew	Moho polvoriento
Septoria_leaf_spot	Hojas manchadas de septoriosis
Spidermite_Two-spotted_spider_mite	Picadura de araña roja de dos manchas
Target_Spot	Punto blanco
Tomato_mosaic_virus	Virus mosaico
Tomato_Yellow_Leaf_Curl_Virus	Virus de la hoja amarilla

4.3. Preprocesado de los datos

Al realizar la carga de datos se tiene un directorio con dos subdirectorios, cada uno de ellos representará un conjunto de datos, uno de datos de entrenamiento (*train*) y otro de validación (*valid*). Cada uno de estos directorios contienen a su vez 11 directorios, representando cada una de las clases.

A continuación, se llevan a cabo las primeras tareas de exploración de las imáge-

nes.

En primer lugar, seleccionando el directorio que contiene los datos de entrenamiento se realiza una función para obtener el número de imágenes existentes por cada tipo de tamaño. Gracias a esta función se sabe que se cuentan con imágenes de variables tamaños, concretamente existen 760 tipos de tamaños distintos. En la siguiente tabla se exponen los 5 tipos de tamaño que más se repiten:

Cuadro 4.2.: **Los 5 tamaños de imágenes más comunes**

Top 5 tamaños de imágenes	
Tamaño (píxeles)	Nº imágenes
256x256	18942
227x227	4120
640x640	1207
533x800	151
800x600	10

De cara a construir un modelo todas las imágenes tienen que tener el mismo tamaño y debida a esta primera toma de contacto se toma la decisión de transformar todas las imágenes a tamaño de 256x256 píxeles. Sin embargo, debido a los modelos usados se termina convirtiendo a imágenes de tamaño 224x224, ya que es un tamaño estándar que entiende cualquier modelo y es el más cercano a los dos tipos de tamaños de imágenes más usuales en nuestros datos.

En segundo lugar, se realiza una muestra aleatoria de una imagen por clase, para visualizar el tipo de imágenes que se van a tratar en la Figura 4.1.

En la Figura 4.1 se puede observar la gran variedad de imágenes que existen, con distintos brillos, distintos fondos o incluso giradas:

- En cuanto al brillo se puede observar que la imagen de Target_Spot tiene mucho más brillo que la de Tomato_Yellow_Leaf_Curl_Virus.
- Con respecto al fondo podemos ver distinciones entre la imagen Bacterial_spot con un fondo grisáceo plano, la imagen Late_blight con fondo completamente negro y la foto Early_blight en la que se ve el resto de la planta de tomate, no solo se ve una hoja.
- También se puede destacar la diferencia entre las posiciones de las hojas, algunas como Bacterial_spot tienen el tallo abajo, otras como Leaf_Mold tienen el tallo arriba y otras como Septoria_leaf_spot tienen el tallo horizontal.

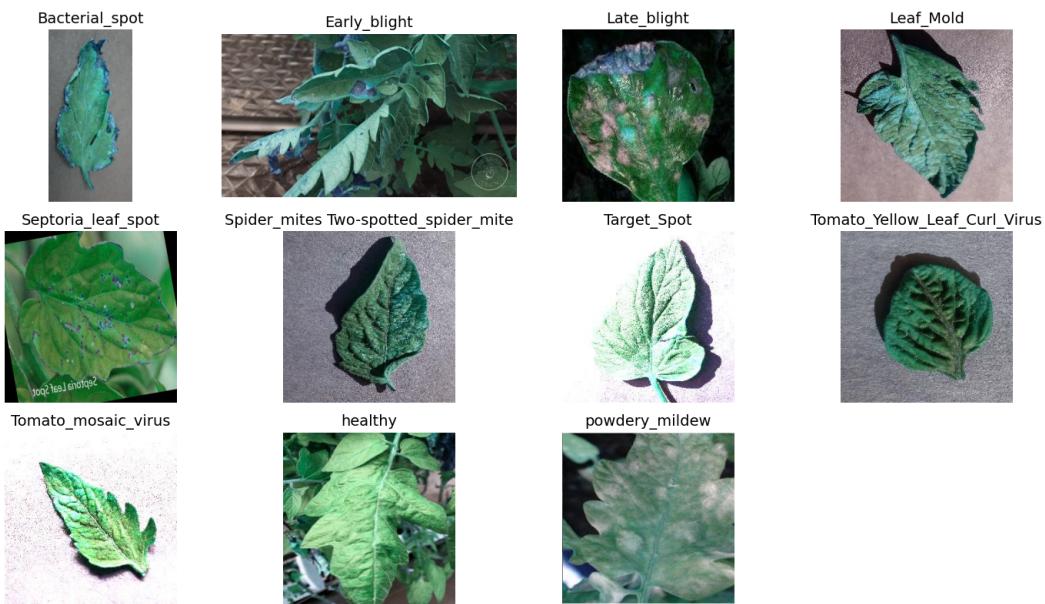


Figura 4.1.: Imagen aleatoria por clase

- Además se tiene un ejemplo de imagen rotada, concretamente la `Septoria_leaf_spot`

Debido a esta gran variedad de imágenes y de enfermedades se llegó a la conclusión de que no tiene mucho sentido usar funciones especiales de la librería PlantCV, ya que esta está muy orientada al fenotipado clásico (área, forma, color, índices), y esas características a veces no capturan la complejidad de patrones de enfermedades, que suelen ser más sutiles y no lineales. La estrategia que se ha seguido es pasar a un pipeline de *deep learning* con imágenes preprocesadas de forma estándar. De esta manera el modelo aprenderá por si mismo las características discriminantes en lugar de imponer un conjunto de *features* manuales.

A continuación se dispuso a formar los conjuntos de datos que usará el modelo. Hasta ahora se tienen datos para el entrenamiento del modelo y para la validación del mismo, sin embargo, no se tienen datos para realizar pruebas sobre el modelo resultante. Por lo tanto, se formará un nuevo conjunto de datos de pruebas a partir del conjunto de entrenamiento, concretamente, seleccionando un 20 % de sus datos.

Para realizar esta tarea se recorre el directorio de datos de entrenamiento y aleatoriamente se seleccionan imágenes de cada subdirectorio (clase) y se añaden a un nuevo directorio que será el de datos de prueba.

Otra tarea importante en cuanto al procesado de los datos es normalizar los mis-

mos. Para ello, se hace uso de *ImageDataagenerator* de Tensorflow. Se usa esta función porque también sirve para aplicar la técnica de aumento de datos (*data augmentation*) (Dong et al., 2023) en los datos de entrenamiento. Esta técnica consiste en generar datos adicionales a partir de los ya existentes aplicando transformaciones que mantienen la esencia de la información original, pero la presentan de manera distinta. Su objetivo principal es enriquecer el conjunto de entrenamiento para mejorar la capacidad de generalización del modelo resultante.

Es decir, a los tres conjuntos de datos se le aplaza normalización que consiste en transformar los pixeles de cada imagen de valores que van de 0 a 255 (escala RGB) a valores entre 0 y 1. Además a las imágenes del conjunto de entrenamiento se le aplica la técnica de *data augmentation* que se ha mencionado anteriormente, concretamente se le aplican las siguientes transformaciones:

- Rota aleatoriamente la imagen hasta más menos 20 grados.
- Aplica un zoom aleatorio entre 80 % y 120 % del tamaño original.
- Desplaza la imagen horizontalmente hasta un 20 % de su ancho.
- Desplaza la imagen verticalmente hasta un 20 % de su alto.
- Aplica una transformación de cizallamiento (*shear*), como si se deformara la imagen en diagonal.
- Gira horizontalmente las imágenes aleatoriamente.
- Cambia el brillo aleatoriamente entre 80 % y 120 %.
- Cuando una transformación (como una rotación o desplazamiento) deja espacios vacíos en la imagen, estos se rellenan con el valor del píxel más cercano.

Estas transformaciones se aplicarán aleatoriamente cada vez que el generador entrega un lote al modelo durante el entrenamiento. De esta manera, el modelo el modelo nunca verá dos veces exactamente la misma versión de la imagen.

Como ya se ha mencionado se ha usado *ImageDataagenerator* para formar los conjuntos de datos. Concretamente lo que permite es crear generadores, que no son más que objetos que producen datos de forma incremental, lote a lote, en lugar de cargar todo el *dataset* en memoria de golpe. El generador se conecta a un directorio con datos y los va leyendo en lotes de un tamaño determinado, en este caso se ha usado un tamaño de 32, es decir, va formando lotes de 32 imágenes y a estas se les aplica normalización y también las transformaciones de *data augmentation*.

mencionadas si son datos de entrenamiento.

De esta manera, quedan los siguientes conjuntos de datos:

- Entrenamiento: Tiene 20.686 imágenes utilizadas para entrenar el modelo. Se trata de la mayor parte de los datos, ya que el modelo necesita muchos ejemplos para aprender patrones.
- Validación: Tiene 6.683 imágenes. Estos datos se utilizan para evaluar el rendimiento del modelo durante el entrenamiento, sin afectar a los parámetros del modelo.
- Prueba: Hay 5.165 imágenes para la prueba final. Este conjunto de datos se utiliza una vez finalizado el entrenamiento para medir objetivamente el rendimiento del modelo con datos nuevos que nunca se han visto.

Otro dato importante del conjunto de entrenamiento es el número de imágenes que se tiene por clase, ya que si hubiera muchas más imágenes de una clase que de otras, el modelo podría incluir un sesgo no deseado. El número de imágenes por clase del conjunto de entrenamiento es el siguiente:

Cuadro 4.3.: Número de imágenes por clases en el conjunto de entrenamiento

Número de imágenes por clases	
Clase	Nº imágenes
Bacterial_spot	2261
Early_blight	1964
Late_blight	2491
Leaf_Mold	2204
Septoria_leaf_spot	2306
Spidermite_Two-spotted_spider_mite	1398
Target_Spot	1462
Tomato_Yellow_Leaf_Curl_Virus	1632
Tomato_mosaic_virus	1723
healthy	2441
powdery_mildew	804

La mayoría de clases tienen entre 2000 y 2400 imágenes, lo cual es aceptable. Sin embargo, existen clases claramente minoritarias, como Spidermite_Two-spotted_spider_mite y Target_Spot con menos de 1500 imágenes, o powdery_mildew con menos de 1000 imágenes. Esto puede causar que el modelo aprenda mejor las clases con más ejemplos y tienda a confundirse en las minoritarias porque tiene menos exposición a ellas.

Para evitar el posible problema de sesgo en el modelo se aplicará una técnica de balanceo al conjunto de entrenamiento. Específicamente se usará ponderación de clases con un diccionario `class_weight` que se puede añadir al modelo en forma de parámetro. Esto dará más peso a los errores en clases minoritarias, para que el modelo no las ignore.

Los pesos de las clases se han calculado usando la función `compute_class_weight` que hace que las clases con menor muestras tengan mayor peso y las que tengan más muestras tengan menor peso. Haciendo, en promedio, que todas las clases tengan la misma importancia. Los pesos aplicados son los que se muestran a continuación:

Cuadro 4.4.: Pesos asignados a cada clase en el entrenamiento del modelo

Pesos por clases		
Clase	Nº imágenes	Peso
Bacterial_spot	2261	0.8317
Early_blight	1964	0.9575
Late_blight	2491	0.7549
Leaf_Mold	2204	0.8532
Septoria_leaf_spot	2306	0.8155
Spidermite_Two-spotted_spider_mite	1398	1.3451
Target_Spot	1462	1.2862
Tomato_Yellow_Leaf_Curl_Virus	1632	1.1522
Tomato_mosaic_virus	1723	1.0914
healthy	2441	0.7703
powdery_mildew	804	2.3389

4.4. Modelado

En esta sección se describe la estrategia de modelado empleada, común a todos los experimentos realizados posteriormente. El enfoque adoptado se basa en el uso de redes neuronales convolucionales (CNN), dado que constituyen la arquitectura de referencia en tareas de clasificación de imágenes al ser capaces de extraer de manera automática y jerárquica características relevantes de los datos visuales.

Considerando las limitaciones de recursos computacionales disponibles, se optó por la técnica de *transfer learning*. Esta metodología permite aprovechar modelos previamente entrenados sobre grandes bases de datos, de modo que las capas iniciales ya contienen representaciones generales de las imágenes. Posteriormente,

dichas representaciones se ajustan a la tarea específica de clasificación de enfermedades en hojas de tomate.

Concretamente se han usado los modelos *MobileNetV2*, *NASNetMobile* y *EfficientNetB0* preentrenados con el dataset de *ImageNet*. De esta manera se aprovechan los pesos previamente aprendidos, que contienen representaciones visuales generales como bordes, texturas y formas, para inicializar la red. Posteriormente, esta red se adapta a la clasificación de enfermedades en hojas de tomate.

Además, cabe mencionar que para algunos modelos se ha usado la técnica de *fine tuning* que va un poco más allá que *transfer learning*, ya que tras entrenar un modelo con una gran base de datos se descongelan algunas de sus capas para que el modelo final se adapte más a la tarea específica que en este caso es la clasificación de enfermedades en hojas de tomate.

Al haber usado la técnica de *fine tuning* tiene sentido añadir las características principales de las arquitecturas de redes neuronales usadas. Esta tarea se lleva a cabo en la sección 4.5. Con el objetivo de gestionar mejor los recursos disponibles, se implementaron distintos callbacks:

- *HistorySaver*: permite guardar el historial de métricas en un archivo externo. Esto resulta especialmente útil en entornos con recursos limitados o sesiones interrumpibles (como Google Colab), ya que garantiza que la información del entrenamiento no se pierda. Concretamente se ha usado para guardar las métricas de precisión y pérdida de entrenamiento y validación.
- *ModelCheckpoint*: encargado de almacenar en disco el modelo con mejor desempeño en validación, según la métrica de precisión (`val_accuracy`). De este modo, se asegura la conservación de la mejor versión del modelo entrenado, evitando depender únicamente de los pesos finales.

Estos callbacks se han usado en todos los entrenamientos. Sin embargo, en algunos entrenamientos se han usado además otros dos callbacks:

- *EarlyStopping*: detiene el entrenamiento de forma anticipada si la pérdida de validación no mejora durante un número determinado de épocas consecutivas. Esto evita sobreentrenamiento y reduce el tiempo de cómputo innecesario.
- *ReduceLROnPlateau*: ajusta de manera dinámica la tasa de aprendizaje cuando la pérdida de validación alcanza una meseta. Gracias a esta reducción progresiva, el modelo puede seguir afinando sus parámetros con pasos cada vez más pequeños, lo que mejora la convergencia.

En conjunto, estos callbacks permitieron no solo optimizar el uso de los recursos computacionales disponibles, sino también obtener modelos más robustos y con mejor capacidad de generalización.

Por último cabe destacar que para que los modelos puedan ser reconstruidos por cualquiera con acceso a los mismos datos de los que se parte es necesario fijar a un valor fijo distintas semillas aleatorias:

- Para la inicialización de los pesos en el modelo se usa `tf.random.set_seed(42)`.
- Para la generación de los conjuntos de datos con `ImageDatagenerator` se usa el parámetro `seed` con valor 42.
- Para el barajado de los datos del conjunto de entrenamiento `random.seed(42)` y `np.random.seed(42)`.

4.4.1. Modelo *MobileNetV2* 1

Este modelo usa de capa base *MobileNetV2* preentrenada con los pesos de *ImageNet*. Posteriormente se congelan las capas del modelo base para evitar que los pesos se modifiquen durante el entrenamiento y se use correctamente la técnica de *transfer learning*. A continuación se añade una capa de *GlobalAveragePooling* y una capa densa con 11 neuronas y función de activación *softmax*, ya que nos enfrentamos a una clasificación de 11 clases. Esta información es la que se muestra en la Figura 4.2.

Layer (type)	Output Shape	Param #
mobilenetv2_1.00_224 (Functional)	(None, 7, 7, 1280)	2,257,984
global_average_pooling2d_2 (GlobalAveragePooling2D)	(None, 1280)	0
dense_6 (Dense)	(None, 11)	14,091
Total params: 2,272,075 (8.67 MB) Trainable params: 14,091 (55.04 KB) Non-trainable params: 2,257,984 (8.61 MB)		

Figura 4.2.: [Resumen del modelo *MobileNetV2* 1](#)

Finalmente para la compilación se usa el optimizador Adam, la función de pérdida de entropía cruzada categórica y se escoge la métrica de precisión para evaluar el rendimiento.

Se usa la función de pérdida de entropía cruzada categórica porque al crear los conjuntos de datos con `ImageDatagenerator` como se indica en la sesión 4.3 se usa el parámetro `class mode` con valor `categorical` que convierte las etiquetas a vectores `one-hot encoded`.

4.4.2. Modelo *MobileNetV2* 2

Este modelo usa de capa base *MobileNetV2* preentrenada con los pesos de *ImageNet*. Posteriormente se congelan las capas del modelo base para evitar que los pesos se modifiquen durante el entrenamiento, pero en esta ocasión se descongelen algunas capas, usando la técnica de *fine tuning*. Concretamente, se descongela a partir de la capa 100 con el objetivo de que el modelo se adapte mejor a nuestro caso concreto como se explica en el Cuadro 4.5. A continuación se añade una capa de *GlobalAveragePooling* y una capa densa con 11 neuronas y función de activación *softmax*, ya que nos enfrentamos a una clasificación de 11 clases. Esta información es la que se muestra en la Figura 4.3.

Layer (type)	Output Shape	Param #
mobilenetv2_1.00_224 (Functional)	(None, 7, 7, 1280)	2,257,984
global_average_pooling2d_1 (GlobalAveragePooling2D)	(None, 1280)	0
dense_1 (Dense)	(None, 11)	14,091

Total params: 2,272,075 (8.67 MB)
Trainable params: 1,875,531 (7.15 MB)
Non-trainable params: 396,544 (1.51 MB)

Figura 4.3.: [Resumen del modelo *MobileNetV2* 2](#)

Finalmente para la compilación se usa el optimizador Adam, la función de pérdida de entropía cruzada categórica y se escoge la métrica de precisión para evaluar el rendimiento. Sin embargo, no se usa un *learning rate* por defecto ($1e-3$), se usa un *learning rate* menor ($1e-5$).

Se usa la función de pérdida de entropía cruzada categórica porque al crear los conjuntos de datos con `ImageDatagenerator` como se indica en la Sección 4.3 se usa el parámetro `class mode` con valor `categorical` que convierte las etiquetas a vectores `one-hot encoded`.

4.4.3. Modelo *MobileNetV2* 3

Este modelo usa de capa base *MobileNetV2* preentrenada con los pesos de *ImageNet*. Posteriormente se congelan las capas del modelo base para evitar que los pesos se modifiquen durante el entrenamiento, pero en esta ocasión se descongelan algunas capas, usando la técnica de *fine tuning*. Concretamente, se descongela a partir de la capa 80 con el objetivo de que el modelo se adapte mejor a nuestro caso concreto como se explica en el Cuadro 4.5. A continuación se añade una capa de *GlobalAveragePooling* y una capa densa con 11 neuronas y función de activación *softmax*, ya que nos enfrentamos a una clasificación de 11 clases. Esta información es la que se muestra en la Figura 4.4.

Layer (type)	Output Shape	Param #
mobilenetv2_1.00_224 (Functional)	(None, 7, 7, 1280)	2,257,984
global_average_pooling2d_1 (GlobalAveragePooling2D)	(None, 1280)	0
dense_1 (Dense)	(None, 11)	14,091

Total params: 2,272,075 (8.67 MB)
 Trainable params: 1,875,531 (7.15 MB)
 Non-trainable params: 396,544 (1.51 MB)

Figura 4.4.: Resumen del modelo *MobileNetV2* 3

Finalmente para la compilación se usa el optimizador Adam, la función de pérdida de entropía cruzada categórica y se escoge la métrica de precisión para evaluar el rendimiento. Sin embargo, no se usa un *learning rate* por defecto ($1e-3$), se usa un *learning rate* menor ($5e-6$).

Se usa la función de pérdida de entropía cruzada categórica porque al crear los conjuntos de datos con *ImageDataGenerator* como se indica en la Sección 4.3 se usa el parámetro *class mode* con valor *categorical* que convierte las etiquetas a vectores *one-hot encoded*.

4.4.4. Modelo *EfficientNetB0* 1

Este modelo usa de capa base *EfficientNetB0* preentrenada con los pesos de *ImageNet*. Posteriormente se congelan las capas del modelo base para evitar que los pesos se modifiquen durante el entrenamiento y se use correctamente la técnica de *transfer learning*. A continuación se añade una capa de *GlobalAveragePooling*

y una capa densa con 11 neuronas y función de activación *softmax*, ya que nos enfrentamos a una clasificación de 11 clases. Esta información es la que se muestra en la Figura 4.5.

Layer (type)	Output Shape	Param #
efficientnetb0 (Functional)	(None, 7, 7, 1280)	4,049,571
global_average_pooling2d (GlobalAveragePooling2D)	(None, 1280)	0
dense (Dense)	(None, 11)	14,091

Total params: 4,063,662 (15.50 MB)
Trainable params: 14,091 (55.04 KB)
Non-trainable params: 4,049,571 (15.45 MB)

Figura 4.5.: Resumen del modelo *EfficientNetB0* 1

Finalmente para la compilación se usa el optimizador Adam, la función de pérdida de entropía cruzada categórica y se escoge la métrica de precisión para evaluar el rendimiento.

Se usa la función de pérdida de entropía cruzada categórica porque al crear los conjuntos de datos con `ImageDataGenerator` como se indica en la Sección 4.3 se usa el parámetro *class mode* con valor *categorical* que convierte las etiquetas a vectores *one-hot encoded*.

4.4.5. Modelo *EfficientNetB0* 2

Este modelo usa de capa base *EfficientNetB0* preentrenada con los pesos de *ImageNet*. Posteriormente se congelan las capas del modelo base para evitar que los pesos se modifiquen durante el entrenamiento, pero en esta ocasión se descongelan algunas capas, usando la técnica de *fine tuning*. Concretamente, se descongela a partir de la capa 100 con el objetivo de que el modelo se adapte mejor a nuestro caso concreto como se explica en el Cuadro 4.6. A continuación se añade una capa de *GlobalAveragePooling* y una capa densa con 11 neuronas y función de activación *softmax*, ya que nos enfrentamos a una clasificación de 11 clases. Esta información es la que se muestra en la Figura 4.6.

Finalmente para la compilación se usa el optimizador Adam, la función de pérdida de entropía cruzada categórica y se escoge la métrica de precisión para evaluar el rendimiento. Sin embargo, no se usa un *learning rate* por defecto ($1e-3$), se usa un *learning rate* menor ($1e-5$).

Layer (type)	Output Shape	Param #
efficientnetb0 (Functional)	(None, 7, 7, 1280)	4,049,571
global_average_pooling2d_3 (GlobalAveragePooling2D)	(None, 1280)	0
dense_3 (Dense)	(None, 11)	14,091

Total params: 4,063,662 (15.50 MB)
Trainable params: 3,854,439 (14.70 MB)
Non-trainable params: 209,223 (817.28 KB)

Figura 4.6.: [Resumen del modelo EfficientNetB0 2](#)

Se usa la función de pérdida de entropía cruzada categórica porque al crear los conjuntos de datos con `ImageDataGenerator` como se indica en la Sección 4.3 se usa el parámetro *class mode* con valor *categorical* que convierte las etiquetas a vectores *one-hot encoded*.

4.4.6. Modelo *EfficientNetB0 3*

Este modelo usa de capa base *EfficientNetB0* preentrenada con los pesos de *ImageNet*. Posteriormente se congelan las capas del modelo base para evitar que los pesos se modifiquen durante el entrenamiento, pero en esta ocasión se descongelan algunas capas, usando la técnica de *fine tuning*. Concretamente, se descongela a partir de la capa 100 con el objetivo de que el modelo se adapte mejor a nuestro caso concreto como se explica en el Cuadro 4.6. A continuación se añade una capa de *GlobalAveragePooling* y una capa densa con 11 neuronas y función de activación *softmax*, ya que nos enfrentamos a una clasificación de 11 clases. Esta información es la que se muestra en la Figura 4.6.

Layer (type)	Output Shape	Param #
efficientnetb0 (Functional)	(None, 7, 7, 1280)	4,049,571
global_average_pooling2d_3 (GlobalAveragePooling2D)	(None, 1280)	0
dense_3 (Dense)	(None, 11)	14,091

Total params: 4,063,662 (15.50 MB)
Trainable params: 3,854,439 (14.70 MB)
Non-trainable params: 209,223 (817.28 KB)

Figura 4.7.: [Resumen del modelo EfficientNetB0 3](#)

Finalmente para la compilación se usa el optimizador Adam, la función de pérdida

de entropía cruzada categórica y se escoge la métrica de precisión para evaluar el rendimiento. Sin embargo, no se usa un *learning rate* por defecto ($1e-3$), se usa un *learning rate* menor ($3e-6$).

Se usa la función de pérdida de entropía cruzada categórica porque al crear los conjuntos de datos con `ImageDataGenerator` como se indica en la Sección 4.3 se usa el parámetro *class mode* con valor *categorical* que convierte las etiquetas a vectores *one-hot encoded*.

4.4.7. Modelo *NASNetMobile* 1

Este modelo usa de capa base *NASNetMobile* preentrenada con los pesos de *ImageNet*. Posteriormente se congelan las capas del modelo base para evitar que los pesos se modifiquen durante el entrenamiento y se use correctamente la técnica de *transfer learning*. A continuación se añade una capa de *GlobalAveragePooling* y una capa densa con 11 neuronas y función de activación *softmax*, ya que nos enfrentamos a una clasificación de 11 clases. Esta información es la que se muestra en la Figura 4.8.

Layer (type)	Output Shape	Param #
<code>nasnet_mobile (Functional)</code>	(None, 7, 7, 1056)	4,269,716
<code>global_average_pooling2d_2 (GlobalAveragePooling2D)</code>	(None, 1056)	0
<code>dense_2 (Dense)</code>	(None, 11)	11,627

Total params: 4,281,343 (16.33 MB)
Trainable params: 11,627 (45.42 KB)
Non-trainable params: 4,269,716 (16.29 MB)

Figura 4.8.: [Resumen del modelo *NASNetMobile* 1](#)

Finalmente para la compilación se usa el optimizador Adam, la función de pérdida de entropía cruzada categórica y se escoge la métrica de precisión para evaluar el rendimiento.

4.4.8. Modelo *NASNetMobile* 2

Este modelo usa de capa base *NASNetMobile* preentrenada con los pesos de *ImageNet*. Posteriormente se congelan las capas del modelo base para evitar que los

pesos se modifiquen durante el entrenamiento, pero en esta ocasión se descongelen algunas capas, usando la técnica de *fine tuning*. Concretamente, se descongela a partir de la capa 100 con el objetivo de que el modelo se adapte mejor a nuestro caso concreto como se explica en el Cuadro 4.7. A continuación se añade una capa de *GlobalAveragePooling* y una capa densa con 11 neuronas y función de activación *softmax*, ya que nos enfrentamos a una clasificación de 11 clases. Esta información es la que se muestra en la Figura 4.9.

Layer (type)	Output Shape	Param #
nasnet_mobile (Functional)	(None, 7, 7, 1056)	4,269,716
global_average_pooling2d (GlobalAveragePooling2D)	(None, 1056)	0
dense (Dense)	(None, 11)	11,627

Total params: 4,281,343 (16.33 MB)
 Trainable params: 4,233,471 (16.15 MB)
 Non-trainable params: 47,872 (187.00 KB)

Figura 4.9.: [Resumen del modelo NASNetMobile 2](#)

Finalmente para la compilación se usa el optimizador Adam, la función de pérdida de entropía cruzada categórica y se escoge la métrica de precisión para evaluar el rendimiento. Sin embargo, no se usa un *learning rate* por defecto ($1e-3$), se usa un *learning rate* menor ($5e-6$).

Se usa la función de pérdida de entropía cruzada categórica porque al crear los conjuntos de datos con `ImageDatagenerator` como se indica en la Sección 4.3 se usa el parámetro *class mode* con valor *categorical* que convierte las etiquetas a vectores *one-hot encoded*.

4.5. Arquitecturas

En esta sección se explicarán las principales características de las arquitecturas empleadas para construir los modelos de la sección 4.4

4.5.1. MobileNetV2

MobileNetV2 es una arquitectura de CNN muy eficiente diseñada para aplicaciones de visión integrada en móviles. Ha sido desarrollada por Google para mejorar

a su antecesor *MobileNetV1* proporcionando una mejor precisión y reduciendo la capacidad de cómputo. Sus características claves son las siguientes (contributors, 2025b):

1. Bloques residuales invertidos (*Inverted Residuals*): Primero se expande el número de canales mediante una convolución 1×1 (capa de expansión). Luego se aplica una convolución "depthwise" (convolución separada por canal) para realizar el filtrado espacial. Finalmente, se reduce de nuevo el número de canales a la dimensión deseada mediante otra 1×1 (capa de proyección). Esta estructura permite conservar dimensiones de entrada/salida mientras se reduce el coste computacional.
2. Convoluciones separables en profundidad (*Depthwise Separable Convolutions*): Al igual que su antecesor, utiliza convoluciones separables, es decir, se descompone una convolución estándar en dos operaciones; primero una convolución "depthwise" por cada canal, luego una "pointwise 1×1" que mezcla los canales. Esto reduce considerablemente el número de parámetros y de cálculos, lo que mejora la eficiencia.
3. Embotellamientos lineales (*Linear Bottlenecks*): Entre capas se insertan *bottlenecks* lineales, que ayudan a que la "variedad" (*manifold*) de los datos de entrada no se comprima excesivamente. Esta práctica contribuye a conservar mayor información, lo cual mejora la precisión del modelo.
4. Función de activación ReLU6: emplea la función de activación ReLU6, una versión modificada de la ReLU clásica que limita los valores de activación al rango [0, 6]. Esto favorece la cuantización (es decir, el uso eficiente de formatos de menor precisión) en dispositivos móviles manteniendo un buen equilibrio entre eficiencia y precisión.

A continuación en el Cuadro 4.5 se expone una división funcional de la red para entender qué aprenden las capas:

4.5.2. *EfficientNetB0*

EfficientNet es una familia de redes neuronales convolucionales (CNN) cuyo objetivo es lograr un alto rendimiento con menos recursos computacionales en comparación con arquitecturas anteriores. Sus características claves son las siguientes (contributors, 2025a):

Cuadro 4.5.: División funcional de MobileNetV2

División funcional de MobileNetV2		
Zona	Capas	Aprendizaje
Zona temprana	Primeros bloques (aproximadamente hasta la capa 20)	Características muy básicas: bordes, texturas simples, colores, orientación, gradienes, es decir, filtros universales que casi cualquier red visual aprende
Zona intermedia	Bloques medios (aproximadamente de la capa 20 a la 80)	Características más complejas: combinaciones de formas, patrones locales, partes de objetos, texturas específicas de hojas/plantas, etc. Aquí ya se empiezan a adaptar a dominios más específicos.
Zona tardía / cabeza del modelo	Últimos bloques (aproximadamente de la capa 80 hasta la final)	Características de muy alto nivel: semánticas del dominio, relaciones globales en la imagen y adaptación al problema concreto

1. **Tallo (Stem):** Comienza con una etapa de stem que consiste en una convolución estándar seguida de normalización por lotes (*Batch Normalization*) y activación ReLU6.
2. **Cuerpo (Body):** se compone de una serie de bloques MBConv con diferentes configuraciones (expansión de canales, tamaños de kernel, stride, mecanismo "*squeeze-and-excitation*").
3. **Head (Cabeza):** Incluye un bloque convolutional final para clasificación.
4. **Bloques internos:** Al igual que *MobileNetV2* usa convoluciones separables en profundidad (*Depthwise Separable Convolutions*) y bloques residuales invertidos (*Inverted Residuals*). Además añade bloques *Squeeze-and-Excitation* para calibrar la importancia de los canales.

A continuación en el Cuadro 4.6 se expone una división funcional de la red para entender qué aprenden las capas:

4.5.3. **NASNetMobile**

NASNetMobile es una arquitectura de CNN . Sus características claves son las siguientes (contributors, 2025b):

- 1.

Cuadro 4.6.: División funcional de EfficientNetB0

División funcional de EfficientNetB0		
Zona	Capas	Aprendizaje
Zona temprana	Desde la entrada (<i>stem</i>) hasta los primeros bloques <i>MBConv</i> (aproximadamente hasta la capa 20)	Características muy básicas: bordes, texturas simples, colores, orientación, gradientes, es decir, filtros universales que casi cualquier red visual aprende
Zona intermedia	Bloques medios de <i>MBConv</i> con mayor expansión y mayor profundidad (aproximadamente de la capa 20 a la 80)	Características más complejas: combinaciones de formas, patrones locales, partes de objetos, texturas específicas de hojas/plantas, etc. Aquí ya se empiezan a adaptar a dominios más específicos.
Zona tardía / cabeza del modelo	Últimos bloques (aproximadamente de la capa 80 hasta la final)	Características de muy alto nivel: semánticas del dominio, relaciones globales en la imagen y adaptación al problema concreto

2.

3.

4.

A continuación en el Cuadro 4.7 se expone una división funcional de la red para entender qué aprenden las capas:

Cuadro 4.7.: División funcional de NASNetMobile

División funcional de NASNetMobile		
Zona	Capas	Aprendizaje
Zona temprana	Primeros bloques (aproximadamente hasta la capa 20)	Características muy básicas: bordes, texturas simples, colores, orientación, gradienes, es decir, filtros universales que casi cualquier red visual aprende
Zona intermedia	Bloques medios (aproximadamente de la capa 20 a la 80)	Características más complejas: combinaciones de formas, patrones locales, partes de objetos, texturas específicas de hojas/plantas, etc. Aquí ya se empiezan a adaptar a dominios más específicos.
Zona tardía / cabeza del modelo	Últimos bloques (aproximadamente de la capa 80 hasta la final)	Características de muy alto nivel: semánticas del dominio, relaciones globales en la imagen y adaptación al problema concreto

5. Evaluación y resultados

En este capítulo se presentan los resultados obtenidos al usar los datos preprocesados explicado en la sección 4.3 a los modelos descritos en la sección 4.4.

5.1. Modelo *MobileNetV2* 1

En esta sección se van a mostrar los resultados del entrenamiento del modelo descrito en la subsección 4.4.1. A este modelo se han aplicado varios entrenamientos distintos explicados en las subsecciones 5.1.1, 5.1.2 y 5.1.3.

5.1.1. Entrenamiento 1

Este entrenamiento se ha realizado usando los pesos de clases descritos en el Cuadro 4.4. Solo se ha sometido a 10 épocas y se han usado los callbacks de *ModelCheckpoint* e *HistorySaver*, ya que el objetivo de este entrenamiento es observar la tendencia inicial para poderlo comparar con el entrenamiento de la sección 5.1.2 y poder determinar qué tanto puede influir los pesos en las clases.

Para mostrar los resultados de este entrenamiento se va a comenzar mostrando las métricas recogidas en el fichero generado por el callback *HistorySaver* en la Figura 5.1.

En esta figura se puede observar el comportamiento de las métricas de precisión y pérdida de entrenamiento y validación a lo largo de las épocas a las que se ha sometido el modelo descrito en la subsección 4.4.1 en el entrenamiento. A continuación se describe lo que se puede interpretar:

- Precisión de entrenamiento: Comienza alrededor de 0.6 y sube rápidamente a un valor cercano a 0.8, mostrando que el modelo aprende bien de los datos de entrenamiento.

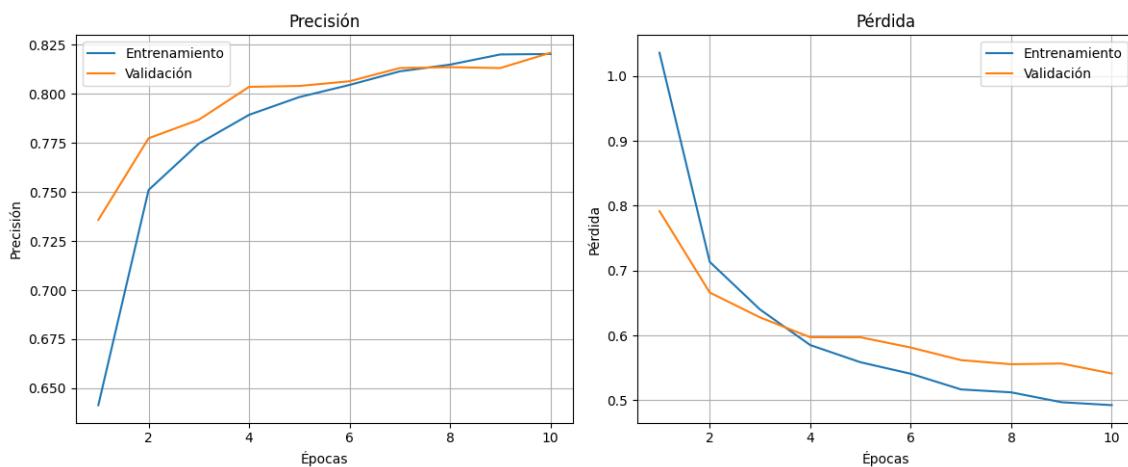


Figura 5.1.: Histórico de métricas del entrenamiento 1 del modelo *MobileNetV2 1*

- Pérdida de entrenamiento: Disminuye de un valor cercano a 1 hasta cerca de 0.6, lo que confirma que el modelo está optimizando correctamente con la función de pérdida elegida para los datos de entrenamiento.
- Precisión de validación: Aumenta de un valor cercano a 0.7 a un valor cercano a 0.8, lo que muestra que el modelo aprende bien de datos distintos a los de entrenamiento.
- Pérdida de validación: Disminuye de un valor cercano a 0.8 a un valor cercano a 0.5, confirmando que la función de pérdida optimiza bien el modelo.

En resumen, ambas métricas de precisión tienden a subir acercándose a 1 y las métricas de pérdidas a bajar acercándose a 0. Ambas tendencias se realizan de manera progresiva sin señales de *overfitting*.

La precisión de entrenamiento se estabiliza alrededor de 0.8, con incrementos pequeños al final. Esto puede indicar que el modelo se está acercando a su límite de capacidad con la configuración actual. Para exprimir aún más este modelo con intención de mejorarlo se podría aumentar el número de épocas con el callback *EarlyStopping* para que pare el entrenamiento en caso de que no mejore y el callback de *ReduceLROnPlateau* para que reduzca el *learning rate* en el entrenamiento y haga que no se estanquen sus valores.

Para poder comparar este entrenamiento con el de la subsección 5.1.2 es necesario realizar un análisis de métricas por clases. Para ello se van a mostrar las métricas del método `classification_report` de la librería *Sklearn* en la Figura 5.2 y la matriz

de confusión en la Figura 5.3.

	precision	recall	f1-score	support
Bacterial_spot	0.800	0.791	0.795	565
Early_blight	0.780	0.692	0.734	491
Late_blight	0.818	0.768	0.793	622
Leaf_Mold	0.801	0.842	0.821	550
Septoria_leaf_spot	0.757	0.705	0.730	576
Spider_mites Two-spotted_spider_mite	0.849	0.837	0.843	349
Target_Spot	0.647	0.868	0.742	365
Tomato_Yellow_Leaf_Curl_Virus	0.939	0.948	0.944	407
Tomato_mosaic_virus	0.948	0.840	0.890	430
healthy	0.949	0.862	0.904	610
powdery_mildew	0.640	0.935	0.760	200
accuracy			0.814	5165
macro avg	0.812	0.826	0.814	5165
weighted avg	0.823	0.814	0.815	5165

Figura 5.2.: [Resumen de métricas del modelo MobileNetV2 1 en el entrenamiento 1](#)

Para entender las métricas de la Figura 5.2 se va a hacer una breve explicación de cada una de ellas:

- Precisión (*Precision*): De todas las imágenes que el modelo predijo como una clase, cuántas realmente pertenecen a esa clase.
- Sensibilidad (*Recall*): De todas las imágenes que son realmente de una clase, cuántas fueron detectadas correctamente.
- *F1-Score*: Es el promedio armónico entre precisión y sensibilidad.
- Soporte (*Support*): El número de imágenes reales de cada clase en el conjunto de test.

Sabiendo esto se puede decir que el modelo tiene un rendimiento homogéneo entre clases habiendo un balance bastante bueno entre precisión y sensibilidad en prácticamente todas las clases.

Para extraer resultados de la Figura 5.3 se va a explicar qué es una matriz de confusión. Una matriz de confusión es una herramienta que permite la visualización del desempeño de un modelo de clasificación. Cada columna representa el número de predicciones de cada clase, mientras que cada fila representa a las instancias en la clase real.

De esta manera se puede afirmar que:

- Las clases Tomato_Yellow_Leaf_Curl_Virus, Tomato_mosaic_virus y healthy

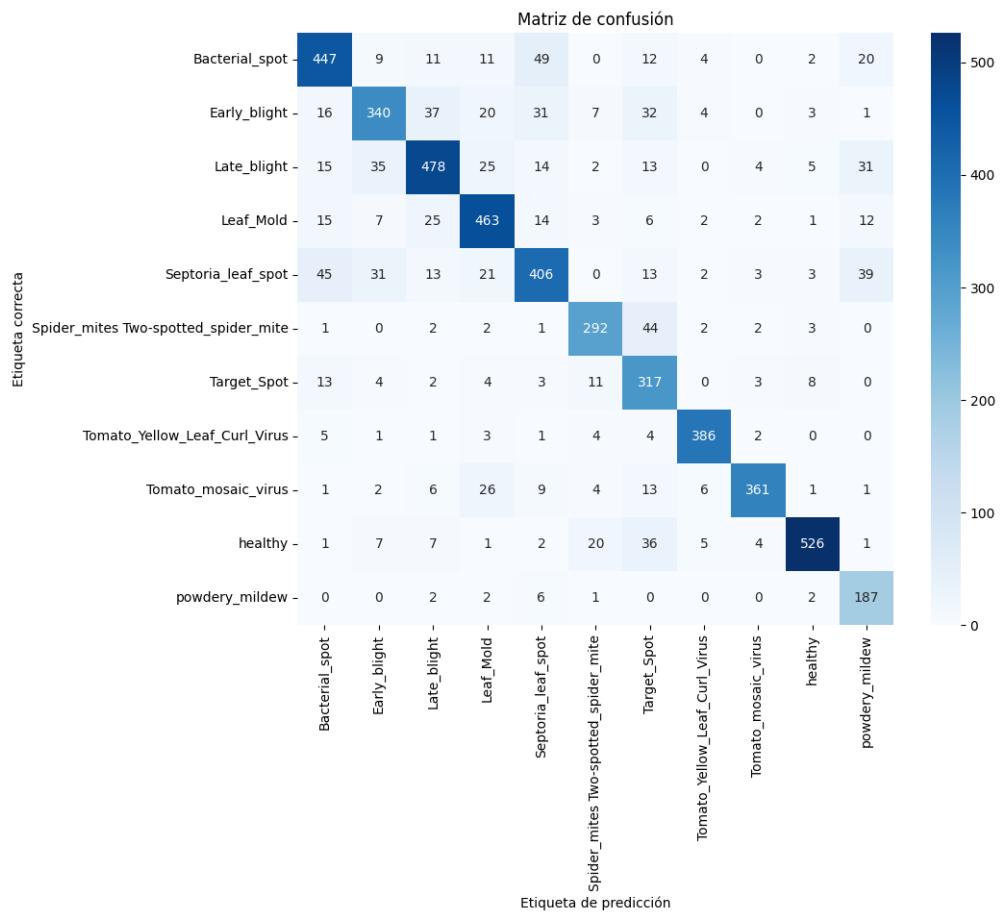


Figura 5.3.: Matriz de confusión del modelo MobileNetV2 1 en el entrenamiento 1

son clasificadas casi a la perfección. Cuando nuestro modelo predice esas clases tiene una tas de acierto mayor al 90 %.

- Las clases Target_Spot, powdery_mildew, Early_blight y Septoria_leaf_spot son las más conflictivas, ya que el modelo tiende a clasificar otras imágenes como ellas.

Este modelo tiene un comportamiento sólido y equilibrado, generaliza bien y consigue que incluso las clases con menos muestras tengan buenos niveles de recall.

5.1.2. Entrenamiento 2

Este entrenamiento es igual que el anterior, pero sin tener en cuenta los pesos de las clases.

Para mostrar los resultados de este entrenamiento se va a comenzar mostrando las métricas recogidas en el fichero generado por el callback *HistorySaver* en la Figura 5.4.

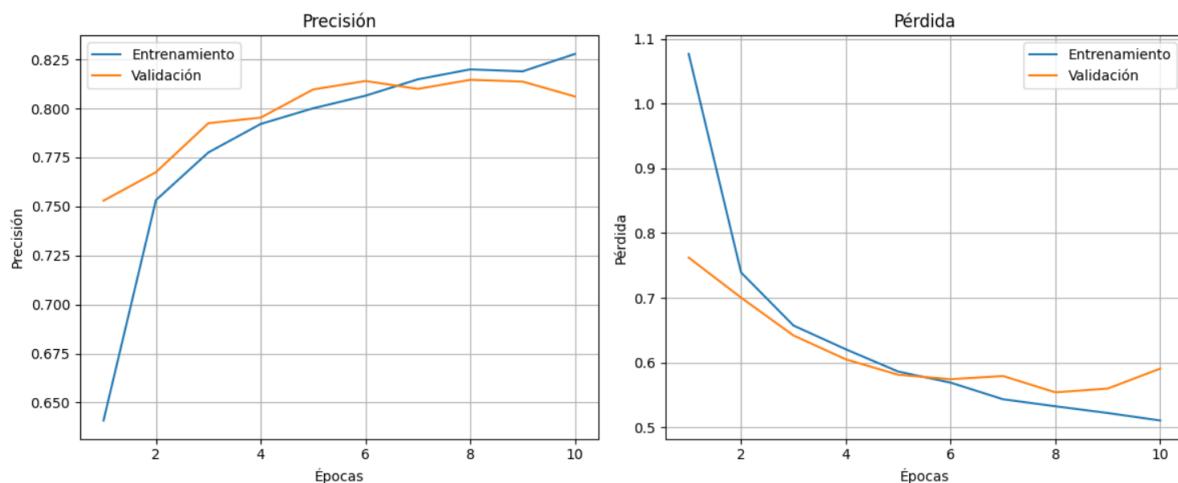


Figura 5.4.: [Histórico de métricas del entrenamiento 2 del modelo MobileNetV2 1](#)

En la Figura 5.4 podemos observar el comportamiento de las métricas de precisión y pérdida de entrenamiento y validación a lo largo de las épocas a las que se ha sometido el modelo descrito en la subsección 4.4.1 en el entrenamiento. A continuación se describe lo que se puede interpretar:

- Precisión de entrenamiento: Comienza alrededor de 0.6 y sube rápidamente a un valor cercano a 0.8, mostrando que el modelo aprende bien de los datos de entrenamiento.
- Pérdida de entrenamiento: Disminuye de un valor cercano a 1 hasta cerca de 0.6, lo que confirma que el modelo está optimizando correctamente con la función de pérdida elegida para los datos de entrenamiento.
- Precisión de validación: Aumenta de un valor cercano a 0.7 a un valor cercano a 0.8, lo que muestra que el modelo aprende bien de datos distintos a los de entrenamiento.
- Pérdida de validación: Disminuye de un valor cercano a 0.8 a un valor cercano a 0.5, confirmando que la función de pérdida optimiza bien el modelo.

En resumen, ambas métricas de precisión tienden a subir acercándose a 1 y las métricas de pérdidas a bajar acercándose a 0. Ambas tendencias se realizan de manera progresiva sin señales de *overfitting*.

El rendimiento global es prácticamente el mismo que el conseguido con el entrenamiento de la subsección 5.1.1, por lo que para poder compararlos más detalladamente es necesario realizar un análisis de métricas por clases. Para ello se van a mostrar las métricas del método `classification_report` de la librería `Sklearn` en la Figura 5.5 y la matriz de confusión en la Figura 5.6.

	precision	recall	f1-score	support
Bacterial_spot	0.742	0.851	0.793	565
Early_blight	0.757	0.684	0.719	491
Late_blight	0.763	0.793	0.778	622
Leaf_Mold	0.717	0.878	0.789	550
Septoria_leaf_spot	0.824	0.611	0.702	576
Spider_mites Two-spotted_spider_mite	0.928	0.705	0.801	349
Target_Spot	0.712	0.811	0.758	365
Tomato_Yellow_Leaf_Curl_Virus	0.997	0.912	0.953	407
Tomato_mosaic_virus	0.905	0.905	0.905	430
healthy	0.875	0.928	0.901	610
powdery_mildew	0.821	0.805	0.813	200
accuracy			0.808	5165
macro avg	0.822	0.807	0.810	5165
weighted avg	0.815	0.808	0.807	5165

Figura 5.5.: [Resumen de métricas del modelo MobileNetV2 1 en el entrenamiento 2](#)

En general se obtiene un buen rendimiento, existiendo un buen balance entre precisión y sensibilidad en prácticamente todas las clases. Aún así, se pueden observar diferencias en algunas clases con respecto al modelo resultante de la subsección 5.1.1. En concreto se puede determinar una disminución en la métrica de sensibilidad en en clases menos representadas en el dataset como las clases `powdery_mildew` (de 0.93 a 0.80), `Septoria_leaf_spot` (de 0.71 a 0.61) y `Spidermite_Two-spotted_spider_mite` (de 0.84 a 0.70).

En cuanto a los resultados de la Figura 5.6 son bastante similares a los de la subsección 5.1.1

- Las clases `Tomato_Yellow_Leaf_Curl_Virus`, `Tomato_mosaic_virus` y `healthy` siguen siendo las que mejor se detectan.
- Las clases `Target_Spot`, `powdery_mildew`, `Early_blight` y `Septoria_leaf_spot` siguen siendo las más conflictivas.

Se puede decir que este modelo rinde igual en términos generales, pero pierde sensibilidad en las clases menos frecuentes. Esto demuestra la función de los pesos en las clases, es decir el balanceo de las mismas.

El modelo con obtenido en la subsección 5.1.1 equilibra mejor los resultados sin

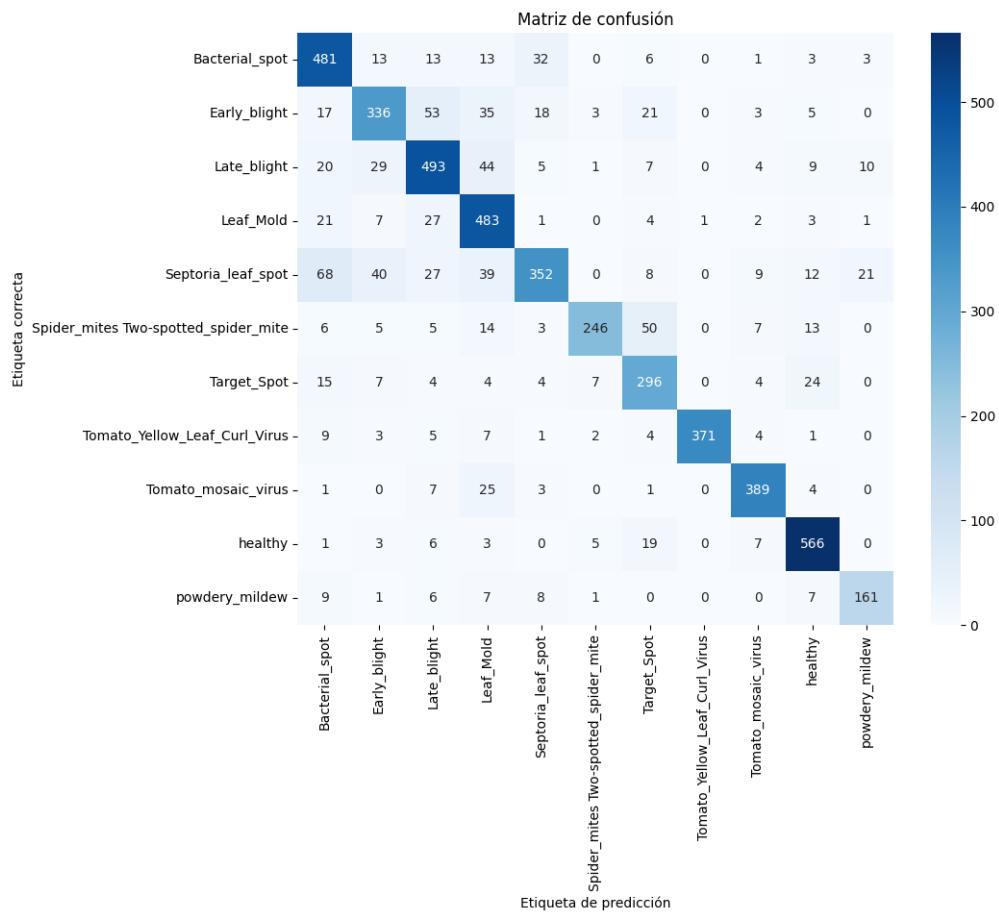


Figura 5.6.: [Matriz de confusión del modelo MobileNetV2 1 en el entrenamiento 2](#)

sacrificar precisión general, por lo que es la opción más sólida para un sistema de diagnóstico de enfermedades, donde todas las clases tienen importancia clínica. Por tanto el modelo que se seguirá mejorando en la suubsección subsección 5.1.3

5.1.3. Entrenamiento 3

Este entrenamiento se ha realizado usando los pesos de clases descritos en el Cuadro 4.4. Solo se ha sometido a 20 épocas y además de los callbacks de *ModelCheckpoint* e *HistorySaver*, se han usado también los callbacks de *ReduceLROnPlateau* y *EarlyStopping*.

Para mostrar los resultados de este entrenamiento se va a comenzar mostrando las métricas recogidas en el fichero generado por el callback *HistorySaver* en la Figura 5.7.

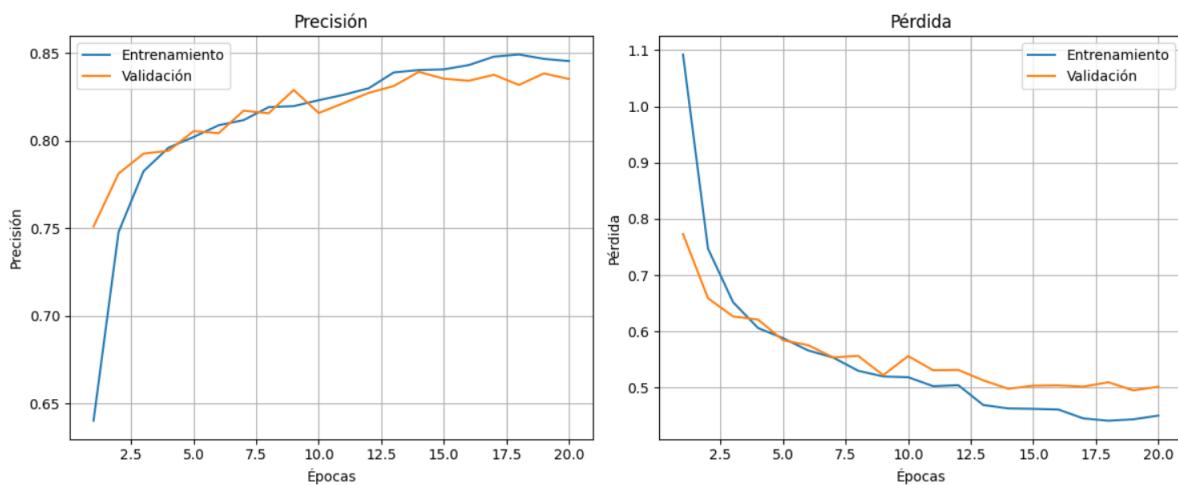


Figura 5.7.: [Histórico de métricas del entrenamiento 3 del modelo MobileNetV2 1](#)

En esta figura se puede observar el comportamiento de las métricas de precisión y pérdida de entrenamiento y validación a lo largo de las épocas a las que se ha sometido el modelo descrito en la subsección 4.4.1 en el entrenamiento. A continuación se describe lo que se puede interpretar:

- Precisión de entrenamiento: Comienza alrededor de 0.65 y sube rápidamente a un valor cercano a 0.85, mostrando que el modelo aprende bien de los datos de entrenamiento.
- Pérdida de entrenamiento: Disminuye de un valor cercano a 1.1 hasta cerca de 0.5, lo que confirma que el modelo está optimizando correctamente con la función de pérdida elegida para los datos de entrenamiento.
- Precisión de validación: Aumenta de un valor cercano a 0.75 a un valor cercano a 0.85, lo que muestra que el modelo aprende bien de datos distintos a los de entrenamiento.
- Pérdida de validación: Disminuye de un valor cercano a 0.8 a un valor cercano a 0.5, confirmando que la función de pérdida optimiza bien el modelo.

En resumen, ambas métricas de precisión tienden a subir acercándose a 1 y las métricas de pérdidas a bajar acercándose a 0. Ambas tendencias se realizan de manera progresiva sin señales de *overfitting*. Cabe destacar que el callback de ReduceLROnPlateau se ha activado en la época 13 reduciendo el learning rate de $1e-3$ a $5e-4$ y en la época 18 reduciendo el learning rate de $5e-4$ a $2.5e-4$, explicando la estabilización que existe.

La precisión de entrenamiento se estabiliza alrededor de 0.85, con incrementos pequeños al final. Esto puede indicar que el modelo se está acercando a su límite de capacidad con la configuración actual. Para exprimir aún más este modelo con intención de mejorarlo se podrían congelar algunas capas del modelo base (*fine tuning*) y reducir el learning rate. Esto dará como resultado el modelo del la Subsección 4.4.2.

Para poder comparar este entrenamiento con el resto es necesario realizar un análisis de métricas por clases. Para ello se van a mostrar las métricas del método `classification_report` de la librería *Sklearn* en la Figura 5.8 y la matriz de confusión en la Figura 5.9.

	precision	recall	f1-score	support
Bacterial_spot	0.803	0.809	0.806	565
Early_blight	0.827	0.711	0.765	491
Late_blight	0.813	0.817	0.815	622
Leaf_Mold	0.800	0.844	0.821	550
Septoria_leaf_spot	0.751	0.724	0.737	576
Spider_mites Two-spotted_spider_mite	0.903	0.822	0.861	349
	0.702	0.838	0.764	365
Tomato_Yellow_Leaf_Curl_Virus	0.992	0.929	0.959	407
Tomato_mosaic_virus	0.929	0.907	0.918	430
	0.864	0.936	0.899	610
healthy	0.838	0.830	0.834	200
powdery_mildew				
accuracy			0.831	5165
macro avg	0.838	0.833	0.834	5165
weighted avg	0.834	0.831	0.831	5165

Figura 5.8.: [Resumen de métricas del modelo MobileNetV2 1 en el entrenamiento 3](#)

En general se obtiene un buen rendimiento, existiendo un buen balance entre precisión y sensibilidad en prácticamente todas las clases. Todos los valores de ambas métricas se encuentran entre 0.7 y 0.99, demostrando que no existen demasiadas diferencias entre las clases, afirmando la robustez del modelo.

En cuanto a los resultados de la Figura 5.9 :

- Las clases Tomato_Yellow_Leaf_Curl_Virus, Tomato_mosaic_virus y Spidermite_Two_spot son clasificadas casi a la perfección. Cuando nuestro modelo predice esas clases tiene una tas de acierto mayor al 90 %.
- Las clases Target_Spot, Early_blight y Septoria_leaf_spot son las más conflictivas, ya que el modelo tiende a clasificar otras imágenes como ellas.

Este modelo tiene un comportamiento sólido y equilibrado, generaliza bien y consigue que incluso las clases con menos muestras tengan buenos niveles de precisión

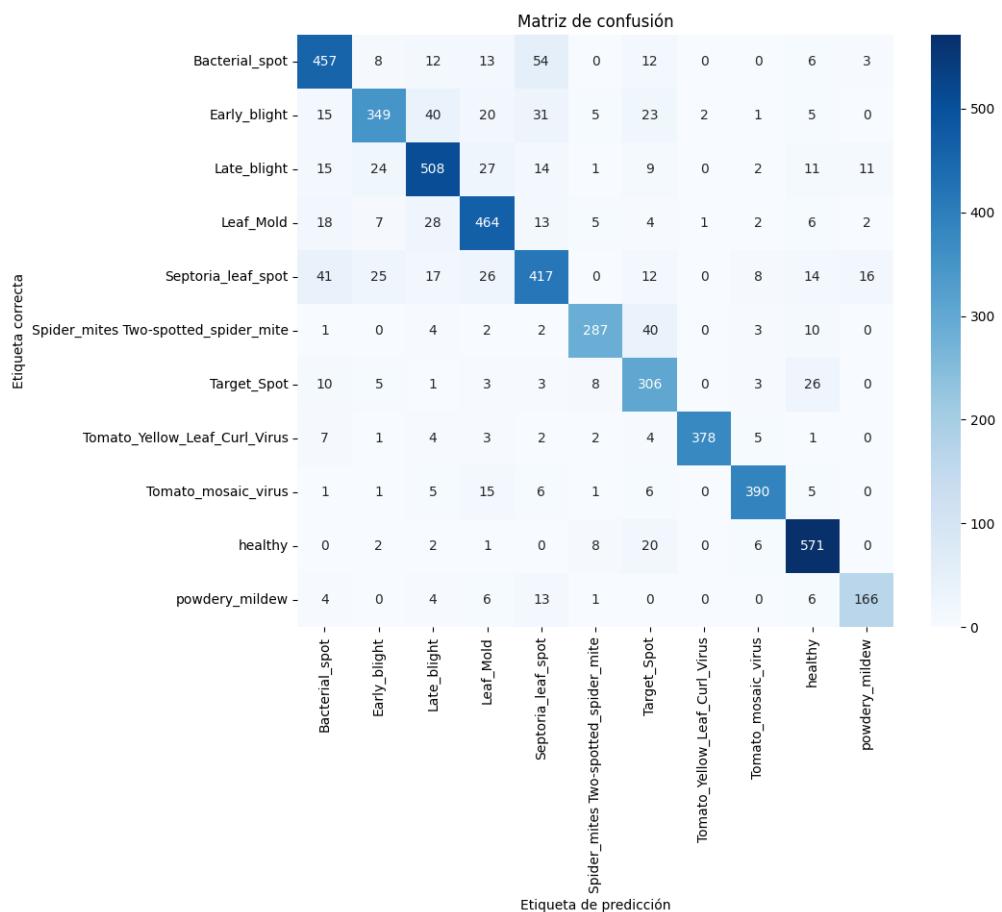


Figura 5.9.: [Matriz de confusión del modelo MobileNetV2 1 en el entrenamiento 3](#)

y sensibilidad. Para seguir refinando este modelo se descongelarán algunas capas del modelo base (fine tuning) y se reducirá el learning rate, estos cambios se aplican en el modelo de la Subsección 4.4.2.

5.2. Modelo *MobileNetV2* 2

5.3. Modelo *MobileNetV2* 3

5.4. Modelo *EfficientNetB0* 1

En esta sección se van a mostrar los resultados del entrenamiento del modelo descrito en la subsección 4.4.4. Este entrenamiento se ha realizado usando los pesos de clases descritos en el Cuadro 4.4. Solo se ha sometido a 10 épocas y se han usado los callbacks de *ModelCheckpoint* e *HistorySaver*, ya que el objetivo de este entrenamiento es observar la tendencia inicial.

Para mostrar los resultados de este entrenamiento se va a comenzar mostrando las métricas recogidas en el fichero generado por el callback *HistorySaver* en la Figura 5.10.

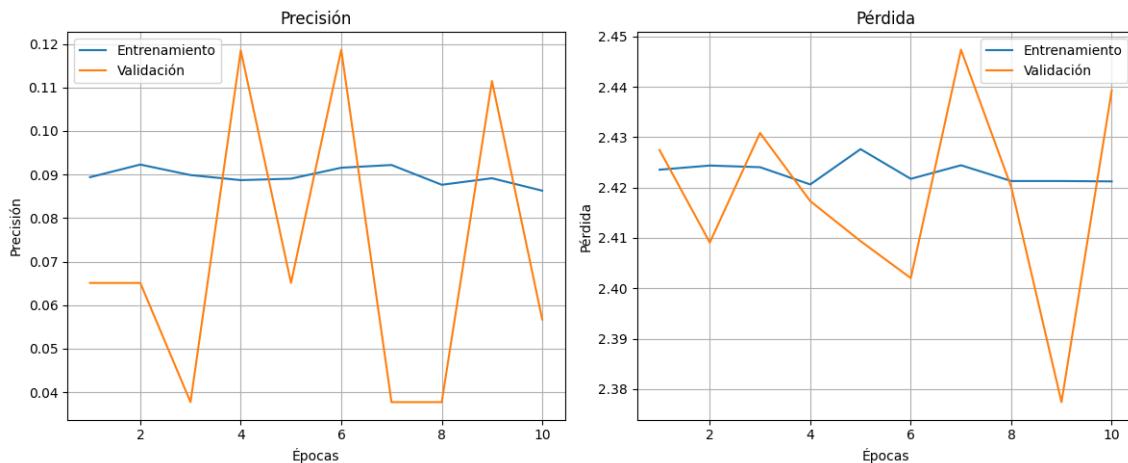


Figura 5.10.: Histórico de métricas del modelo *EfficientNetB0* 1

En esta figura se puede observar el comportamiento de las métricas de precisión y pérdida de entrenamiento y validación a lo largo de las épocas a las que se ha sometido el modelo descrito en la subsección 4.4.4 en el entrenamiento. A continuación se describe lo que se puede interpretar:

- Precisión de entrenamiento: Queda constante en aproximadamente 0.09, un valor muy lejano a 1, lo que demuestra que el modelo no aprende de los datos de entrenamiento.

- Pérdida de entrenamiento: Queda prácticamente constante en un valor cercano a 2.4, bastante lejano a 0.
- Precisión de validación: Oscila entre valores cercanos a 0.1, por lo que el modelo tampoco generaliza.
- Pérdida de validación: Oscila entre valores cercanos a 2.4, bastante lejano a 0.

El modelo no está aprendiendo absolutamente nada más allá del azar. Para mejorar esta situación se descongelarán algunas capas superiores del modelo base *EfficientNetB0*, haciendo uso de la técnica de *fine tuning*.

5.5. Modelo *EfficientNetB0* 2

En esta sección se van a mostrar los resultados del entrenamiento del modelo descrito en la subsección 4.4.5. Este entrenamiento se ha realizado usando los pesos de clases descritos en el Cuadro 4.4. Solo se ha sometido a 10 épocas y se han usado los callbacks de *ModelCheckpoint* e *HistorySaver*, ya que el objetivo de este entrenamiento es observar la tendencia inicial.

Para mostrar los resultados de este entrenamiento se va a comenzar mostrando las métricas recogidas en el fichero generado por el callback *HistorySaver* en la Figura 5.11.

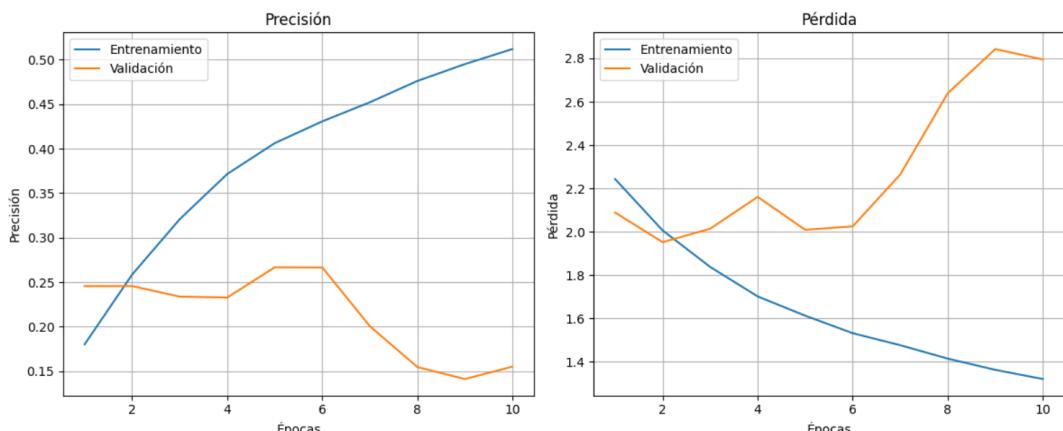


Figura 5.11.: [Histórico de métricas del entrenamiento del modelo *EfficientNetB0* 2](#)

En esta figura se puede observar el comportamiento de las métricas de precisión y pérdida de entrenamiento y validación a lo largo de las épocas a las que se ha some-

tido el modelo descrito en la subsección 4.4.5 en el entrenamiento. A continuación se describe lo que se puede interpretar:

- Precisión de entrenamiento: Aumenta desde un valor aproximado a 0.2 a un valor aproximado a 0.5, lo que demuestra que el modelo está aprendiendo de los datos de entrenamiento.
- Pérdida de entrenamiento: Disminuye de un valor cercano a 2.2 a un valor aproximado a 1.4, mostrando una tendencia decreciente.
- Precisión de validación: Decrece de un valor cercano a 0.25 a un valor cercano a 0.15, por lo que el modelo no generaliza.
- Pérdida de validación: Aumenta desde un valor aproximado a 2.2 a un valor aproximado a 2.8, bastante lejano a 0.

En definitiva, se puede decir que el modelo está aprendiendo a memorizar el conjunto de entrenamiento, pero pierde capacidad de generalización, es decir tiene sobreajuste. Para mejorar esta situación, se va a disminuir el *learning rate* en el modelo de la sección 4.4.6

5.6. Modelo *EfficientNetB0* 3

En esta sección se van a mostrar los resultados del entrenamiento del modelo descrito en la subsección 4.4.6. Este entrenamiento se ha realizado usando los pesos de clases descritos en el Cuadro 4.4. Solo se ha sometido a 10 épocas y se han usado los callbacks de *ModelCheckpoint* e *HistorySaver*, ya que el objetivo de este entrenamiento es observar la tendencia inicial.

Para mostrar los resultados de este entrenamiento se va a comenzar mostrando las métricas recogidas en el fichero generado por el callback *HistorySaver* en la Figura 5.12.

En esta figura se puede observar el comportamiento de las métricas de precisión y pérdida de entrenamiento y validación a lo largo de las épocas a las que se ha sometido el modelo descrito en la subsección 4.4.6 en el entrenamiento. A continuación se describe lo que se puede interpretar:

- Precisión de entrenamiento: Aumenta desde un valor aproximado a 0.15 a un valor aproximado a 0.35, lo que demuestra que el modelo está aprendiendo de

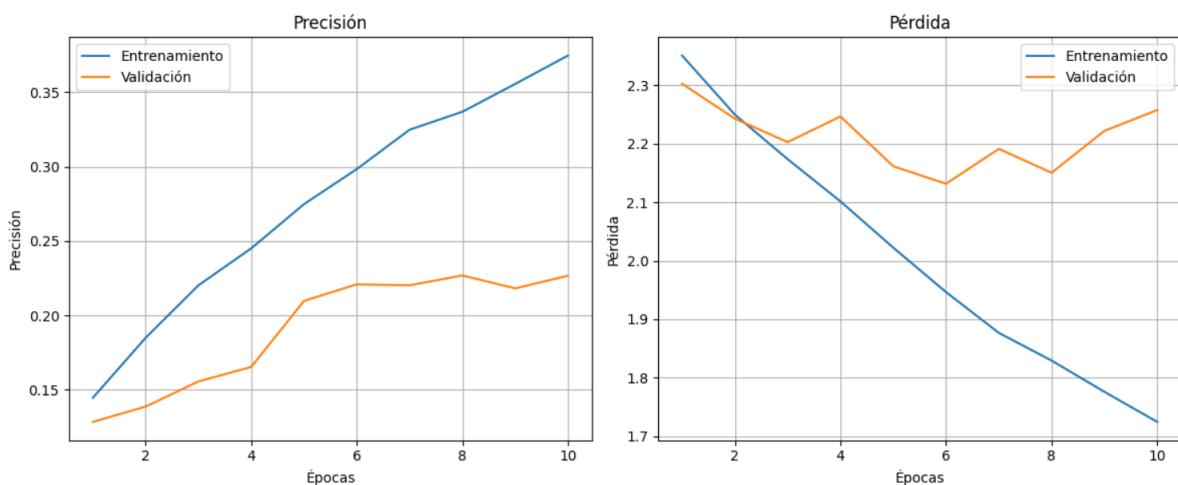


Figura 5.12.: [Histórico de métricas del entrenamiento del modelo *EfficientNetB0 3*](#)

los datos de entrenamiento.

- Pérdida de entrenamiento: Disminuye de un valor cercano a 2.3 a un valor aproximado a 1.8, mostrando una tendencia descendente.
- Precisión de validación: Aumenta de un valor cercano a 0.15 a un valor cercano a 0.25, por lo que el está empezando a generalizar.
- Pérdida de validación: Oscila en un valor cercano a 2.2, por lo que no existe una generalización clara.

En definitiva, se puede decir que se ha disminuido el sobreajuste, el modelo ha comenzado a generalizar, pero no lo suficiente y con la suficiente estabilidad para considerarlo un buen resultado. Para afinar más este modelo se podría incluir una capa de *dropout* antes de la capa densa y entrenar más épocas con los callbacks de *ReduceLROnPlateau* y *EarlyStopping*. Aún así, no se esperan excelentes resultados, simplemente una mejora en la regularización.

5.7. Modelo *NASNetMobile 1*

En esta sección se van a mostrar los resultados del entrenamiento del modelo descrito en la subsección 4.4.7. Para mostrar los resultados de este entrenamiento se va a comenzar mostrando las métricas recogidas en el fichero generado por el callback *HistorySaver* en la Figura 5.13.

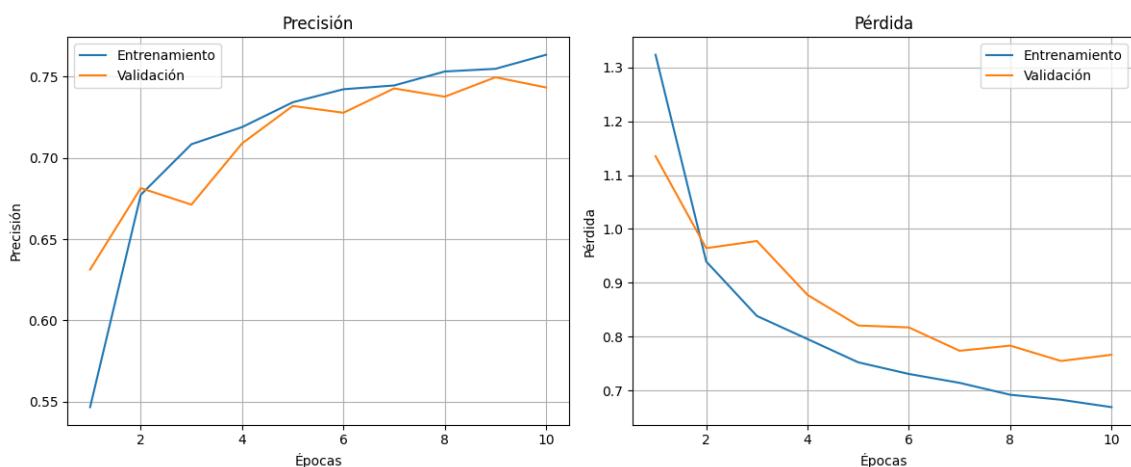


Figura 5.13.: [Histórico de métricas del entrenamiento 1 del modelo *NASNetMobile*](#)

En esta figura se puede observar el comportamiento de las métricas de precisión y pérdida de entrenamiento y validación a lo largo de las épocas a las que se ha sometido el modelo descrito en la subsección 4.4.1 en el entrenamiento. A continuación se describe lo que se puede interpretar:

- Precisión de entrenamiento: Comienza alrededor de 0.5 y sube rápidamente a un valor cercano a 0.7, mostrando que el modelo aprende bien de los datos de entrenamiento.
- Pérdida de entrenamiento: Disminuye de un valor cercano a 1.3 hasta cerca de 0.7, lo que confirma que el modelo está optimizando correctamente con la función de pérdida elegida para los datos de entrenamiento.
- Precisión de validación: Aumenta de un valor cercano a 0.65 a un valor cercano a 0.75, lo que muestra que el modelo aprende bien de datos distintos a los de entrenamiento.
- Pérdida de validación: Disminuye de un valor cercano a 1.1 a un valor cercano a 0.8, confirmando que la función de pérdida optimiza bien el modelo.

El modelo mejora rápidamente durante las primeras 5 épocas. A partir de ese punto, tanto la precisión de entrenamiento como la de validación se estabilizan alrededor del 75 %, lo cual indica que el modelo ha aprendido bien las características generales, pero está alcanzando su límite.

Para mejorar este resultado se podría entrenar más épocas añadiendo los callbacks *EarlyStopping* para que pare la ejecución si el modelo no mejora y con *Reduce*-

`ceLROnPlateau` para que disminuya el *learning rate* en el entrenamiento y haga que no se estanquen sus valores.

5.8. Modelo *NASNetMobile 2*

6. Conclusiones

Se selecciona el mejor modelo y se muestra una gráfica probando el modelo (desarrollar)

A. Anexo I: Ejemplo de anexo

Bibliografía

- Al Sahili, Z., & Awad, M. (2022). The Power of Transfer Learning in Agricultural Applications: AgriNet. *arXiv*. <https://doi.org/10.1101/101182>
- Atila, Ü., Uçar, M., Akyol, K., & Uçar, E. (2021). Plant leaf disease classification using EfficientNet deep learning model. *Ecological Informatics*, 61, 101-182. <https://doi.org/10.1101/101182>
- Bay, H., Ess, A., Tuytelaars, T., & Van Gool, L. (2008). Speeded-Up Robust Features (SURF). *Computer Vision and Image Understanding*, 110(3), 346-359. <https://doi.org/10.1016/j.cviu.2007.09.014>
- Baysal-Gurel, Fulya and Miller, Sally. (2010). *Early Blight Management for Organic Tomato Production*. eOrganic. Consultado el 21 de octubre de 2025, desde <https://eorganic.org/node/4961>
- Chéné, Y., Rousseau, D., Lucidarme, P., Bertheloot, J., Caffier, V., Morel, P., Belin, É., & Chapeau-Blondeau, F. (2012). On the use of depth camera for 3D phenotyping of entire plants. *Computers and Electronics in Agriculture*, 82, 122-127. <https://doi.org/10.1016/j.compag.2011.12.007>
- contributors, G. (2025a). *Efficientnet Architecture*. GeeksforGeeks. Consultado el 28 de octubre de 2025, desde <https://www.geeksforgeeks.org/computer-vision/efficientnet-architecture/>
- contributors, G. (2025b). *Mobilenet V2 Architecture in Computer Vision*. GeeksforGeeks. Consultado el 28 de octubre de 2025, desde <https://www.geeksforgeeks.org/computer-vision/mobilenet-v2-architecture-in-computer-vision/#key-features-of-mobilenet-v2>
- Dalal, N., & Triggs, B. (2005). Histograms of oriented gradients for human detection. *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, 1, 886-893. <https://doi.org/10.1109/CVPR.2005.177>
- Davis, R. M., Miyao, G., Subbarao, K. V., Stapleton, J. J., & Aegeerter, B. J. (s.f.). *Tomato (*Solanum lycopersicum*)-Bacterial Spot*. University of California. Consultado el 23 de octubre de 2025, desde <https://ipm.ucanr.edu/agriculture/tomato/bacterial-spot/#gsc.tab=0>

- Díaz-Pendón, J. A., Cañizares, M. C., Moriones, E., Bejarano, E. R., Czosnek, H., & Navas-Castillo, J. (2010). Tomato yellow leaf curl viruses: ménage à trois between the virus complex, the plant and the whitefly vector. *Molecular Plant Pathology*, 11(4), 441-450. <https://doi.org/10.1111/j.1364-3703.2010.00618.x>
- Dong, X., Wang, K., Huang, Q., Ge, Q., Zhao, K., Wu, X., Wu, X., Liang, L., & Hao, G. (2023). PDDD-PreTrain: A Series of Commonly Used Pre-Trained Models Support Image-Based Plant Disease Diagnosis. *Plant Phenomics*, 5, 0054. <https://doi.org/10.34133/plantphenomics.0054>
- Editors of Encyclopedia Britannica. (2024, agosto). *Late blight*. Encyclopedia Britannica. Consultado el 22 de octubre de 2025, desde <https://www.britannica.com/science/late-blight>
- Garcia-Ruiz, F., Sankaran, S., Maja, J. M., Lee, W. S., Rasmussen, J., & Ehsani, R. (2012). Comparison of two aerial imaging platforms for identification of Huanglongbing-infected citrus trees. *Computers and Electronics in Agriculture*, 91, 106-115. <https://doi.org/10.1016/j.compag.2012.12.002>
- Gevens, A., & Wilbur, J. (2014). *Tomato Late Blight*. University of Wisconsin-Madison. Consultado el 22 de octubre de 2025, desde <https://vegpath.plantpath.wisc.edu/diseases/tomato-late-blight/>
- Hazzard, R. (2022, julio). *Two-spotted Spider Mite*. University of Massachusetts Amherst. Consultado el 23 de octubre de 2025, desde <https://www.umass.edu/agriculture-food-environment/vegetable/fact-sheets/two-spotted-spider-mite>
- Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. En F. Pereira, C. J. C. Burges, L. Bottou & K. Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems* (pp. 1097-1105). Curran Associates, Inc.
- Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60, 91-110. <https://doi.org/10.1023/B:VISI.0000029664.99615.94>
- Lu, J., Liu, X., Ma, X., Tong, J., & Peng, J. (2023). Improved MobileNetV2 crop disease identification model for intelligent agriculture (M. Murugappan, Ed.). *PeerJ Comput Science*, 9, e0123262. <https://doi.org/10.7717/peerj-cs.1595>
- McGrath, Margaret. (s.f.). *Early blight on tomatoes*. Cornell CALS. Consultado el 21 de octubre de 2025, desde <https://blogs.cornell.edu/livegpath/gallery/tomato/early-blight/>
- Mohanty, S. P., Hughes, D. P., & Salathé, M. (2016). Using Deep Learning for Image-Based Plant Disease Detection. *Frontiers in Plant Science*, 7. <https://doi.org/10.3389/fpls.2016.01419>

- Motwani, A., & Khan, Q. (2025, octubre). Tomato Leaves Dataset. *Kaggle*. Consultado el 4 de octubre de 2025, desde <https://www.kaggle.com/datasets/ashishmotwani/tomato/data>
- Pacific Northwest contributors. (2025a). *Tomato (Solanum lycopersicum)-Bacterial Spot*. Pacific Northwest Handbooks. Consultado el 23 de octubre de 2025, desde <https://pnwhandbooks.org/plantdisease/host-disease/tomato-solanum-lycopersicum-bacterial-spot>
- Pacific Northwest contributors. (2025b). *Tomato (Solanum lycopersicum)-Early Blight*. Pacific Northwest Handbooks. Consultado el 21 de octubre de 2025, desde <https://pnwhandbooks.org/plantdisease/host-disease/tomato-solanum-lycopersicum-early-blight>
- Raza, S.-e.-A., Prince, G., Clarkson, J. P., & Rajpoot, N. M. (2015). Automatic detection of diseased tomato plants using thermal and stereo visible light images. *PLoS ONE*, 10(4), e0123262. <https://doi.org/10.1371/journal.pone.0123262>
- Silva, M. (s.f.-a). *Cultivo de tomate: Cómo se realiza, plagas e importancia*. Agrotendencia TV. Consultado el 28 de julio de 2025, desde https://agrotendencia.tv/agricultura/cultivos/hortalizas/el-cultivo-de-tomate/#Historia_del_tomate_o_jitomate
- Silva, M. (s.f.-b). *Cultivo de tomate: Cómo se realiza, plagas e importancia*. Agrotendencia TV. Consultado el 28 de julio de 2025, desde https://agrotendencia.tv/agricultura/cultivos/hortalizas/el-cultivo-de-tomate/#Historia_del_tomate_o_jitomate
- Wikipedia contributors. (s.f.-a). *Producción mundial del tomate*. Wikipedia. Consultado el 28 de julio de 2025, desde https://es.wikipedia.org/wiki/Anexo:Producci%C3%B3n_mundial_de_tomate
- Wikipedia contributors. (s.f.-b). *Solanum lycopersicum*. Wikipedia. Consultado el 28 de julio de 2025, desde https://es.wikipedia.org/wiki/Solanum_lycopersicum