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

De:
Planeta Formación y Universidades

Índice general

Índice de figuras

Índice de cuadros

Lorem ipsum (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, 2025b)

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 eruopa. También lo llevaron a Filipinas y de allí pudo entrar al continente asiático. (Ing. Agr. Miguel Silva, 2025a)

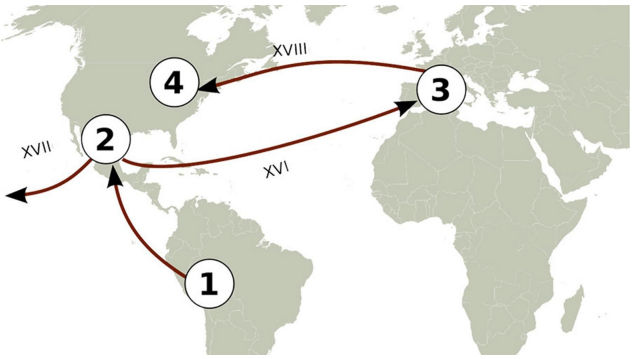

Figura 1.1.: Origen del tomate

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 esta es la evolución de los 20 países que más han producido hasta 2022 es la mostrada en el Cuadro 1.1. (Wikipedia contributors, 2025a)

Cuadro 1.1.: Top 20 países productores de tomates 2022

Titulo				
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
Esapaña	3766	4313	4313	3652

Como ya se ha mencionado, el cultivo de tomate es uno de los cultivos hortícolas 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 (*Phytophthora infestans*): Puede destruir por completo una plantación si no se controla a tiempo, especialmente en condiciones húmedas y templadas.

Figura 1.2.: Tizón tardío en una planta de tomate

- Tizón temprano (*Alternaria solani*): Produce defoliación progresiva, debilitando la planta y reduciendo el número y calidad de los frutos.

Figura 1.3.: Tizón temprano en una planta de tomate

- Fusariosis vascular (*Fusarium oxysporum*): Ataca el sistema vascular, provocando marchitez y muerte de plantas.
- Virus como TYLCV y TSWV: Pueden causar deformaciones severas y reducciones completas en la producción, especialmente cuando se transmiten por vectores como la mosca blanca.

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 inteactuado con la planta, lo que dificulta su erradicación y aumenta los costos del tratamiento. (Ing. Agr. Miguel Silva, 2025b)

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 espacial 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 precsión, 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, los métodos empleados para esta tarea incluían algoritmos 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 una segmentación previa precisa y de la extracción manual de características, lo que limitaba su capacidad de generalización y precisión en entornos reales.

Con la llegada de las redes neuronales convolucionales (CNN), se ha producido un cambio significativo en la forma de abordar este problema. Las CNN son capaces de aprender representaciones directamente a partir de los datos de imagen, eliminando la necesidad de ingeniería manual de características. Diversos estudios han demostrado su eficacia para la clasificación de enfermedades en hojas de tomate.

Por ejemplo, una revisión publicada en la Revista de Investigación e Innovación de las Ciencias de la Universidad Tecnológica de Bolívar (Martínez et al., 2024), las técnicas tradicionales de aprendizaje supervisado como SVM, KNN y lógica difusa muestran limitaciones significativas en tareas de detección de enfermedades en imágenes de frutas debido a su dependencia de extracción manual de características y segmentación previa. En contraste, las redes neuronales convolucionales han demostrado una precisión superior, mayor robustez frente a la variabilidad y mayor capacidad de generalización. Esta revisión respalda la elección de CNNs como enfoque principal en este trabajo.

Por otra parte, Valeria Maeda Gutiérrez (2019) (Gutiérrez, 2019) realizó una comparativa entre varias arquitecturas CNN, incluyendo AlexNet, GoogleNet, InceptionV3, ResNet 18 y ResNet 50 aplicadas al conjunto de datos PlantVillage. Todas las arqui-

tecturas consiguieron más del 98 % de precisión y sensibilidad, lo que confirma la ideonidad de las mismas para la tarea que se pretende hacer. Concretamente con GoogleNet consiguió una precisión del 99,3 % y una sensibilidad del 99,1 %

En otra línea, Eduardo A.Huerta-Mora, Víctor González-Huitrón, Héctor Rodríguez-Rangel y Leonel Ernesto Amabilis-Sosa (2024) (A.Huerta-Mora et al., 2024) emplearon la arquitectura VGG16 con técnicas de fine-tuning para el mismo conjunto de datos PlantVillage, obteniendo alrededor del 90 % de sensibilidad y precisión. Este hecho confirma que esta arquitectura también puede ser interesante para el estudia a realizar.

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 preprocesmiento 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 accedía 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 usuado 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 tamañao estándar que entiende cualquier modelo y es el más cercano a los dos tipo de tamños 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 vir 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.

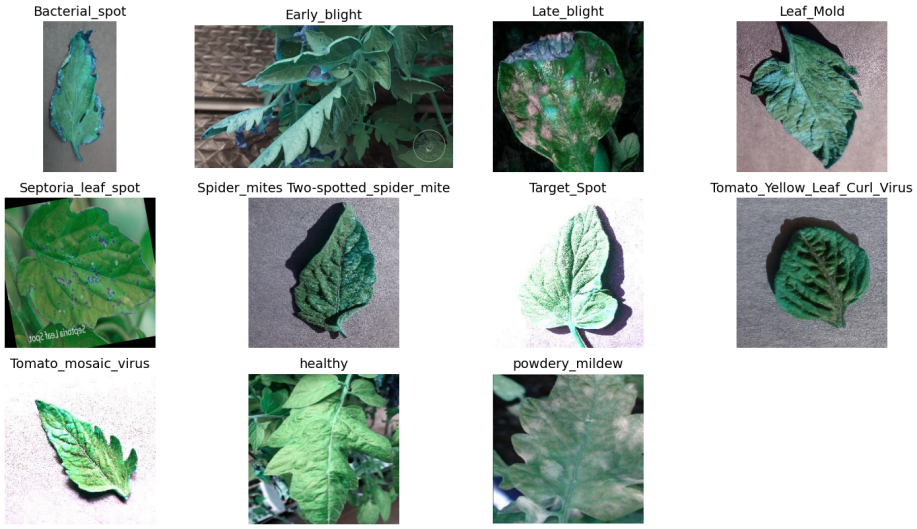

Figura 4.1.: Imagen aleatoria por clase

- 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 cunjunto 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 mismos. Para ello, se hace uso de *ImageDatagenerator* de Tensorflow. Se usa esta

función porque también sirve para aplicar la técnica de data augmentation 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 apliza 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 *ImageDatagenerator* 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 imágnes 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.

Con el objetivo de gestionar mejor los recursos disponibles, se implementaron disitntos 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 disintas 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 entrenmaiento `random.seed(42)` y `np.random.seed(42)`.

4.4.1. Modelo *MobileNetV2*

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*

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 sessción 4.3 se usa el parámetro class mode con valor categorical que convierte las etiquetas a vectores one-hot encoded

4.4.2. Modelo *EfficientNetB0*

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 4.3.

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 4.3.: Resumen del modelo *EfficientNetB0*

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.

4.4.3. Modelo *NASNetMobile*

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 4.4.

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.

Layer (type)	Output Shape	Param #
nasnet_mobile (Functional)	(None, 7, 7, 1056)	4,269,716
global_average_pooling2d_2 (GlobalAveragePooling2D)	(None, 1056)	0
dense_2 (Dense)	(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 4.4.: Resumen del modelo *NASNetMobile*

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*

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.

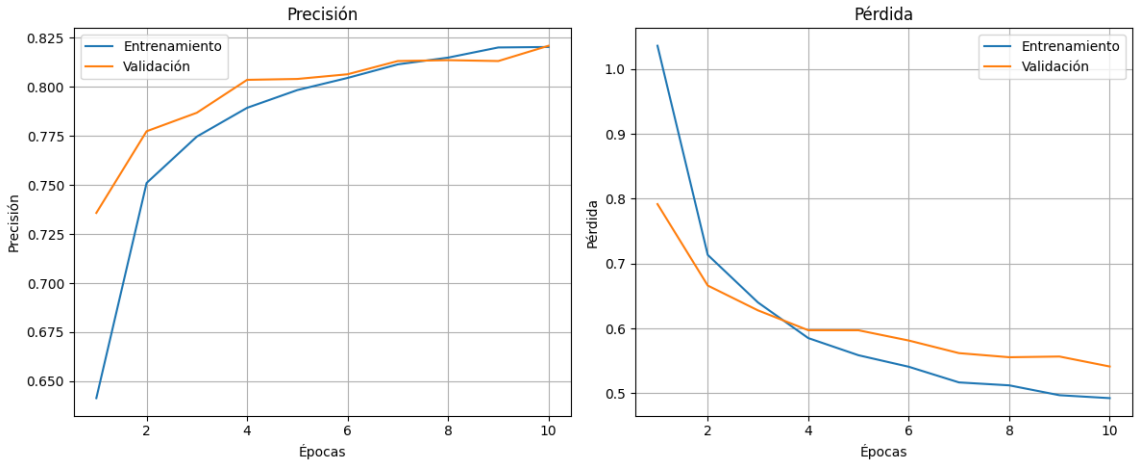

Figura 5.1.: Histórico de métricas del entrenamiento 1 del modelo *MobileNetV2*

- 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* 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 practicamente 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`

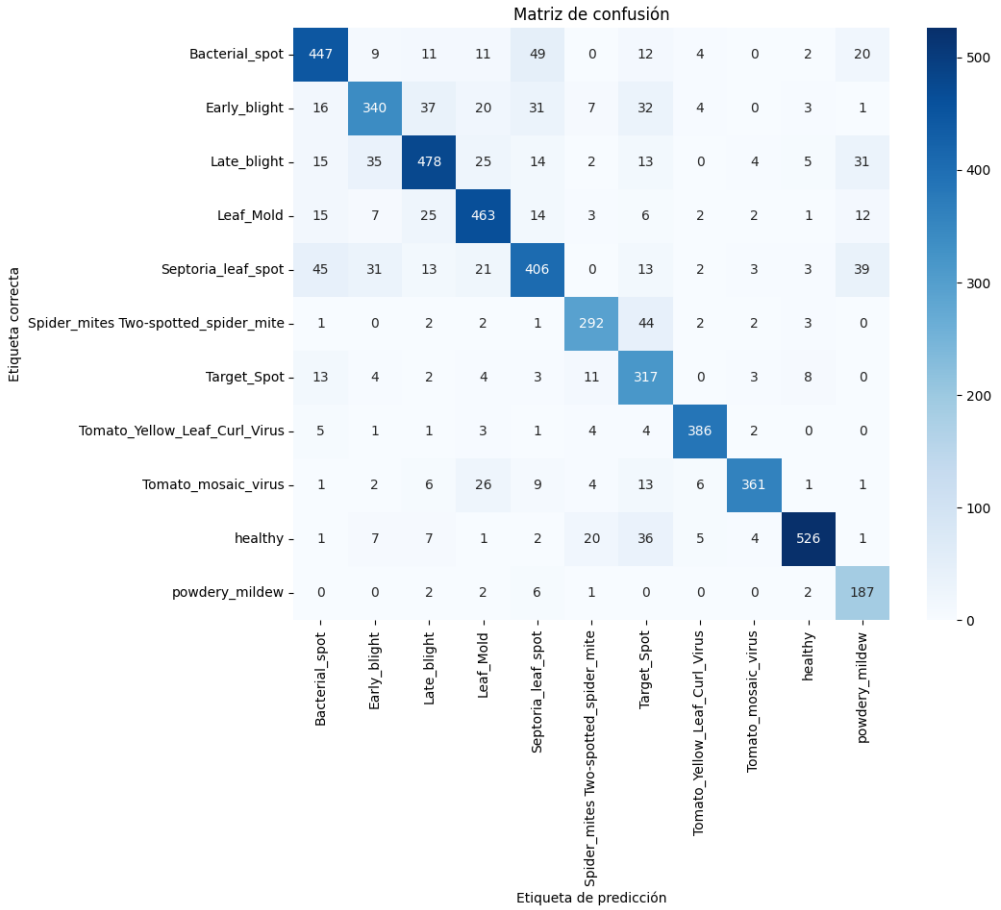

Figura 5.3.: Matriz de confusión del modelo *MobileNetV2* en el entrenamiento 1

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 5.3.

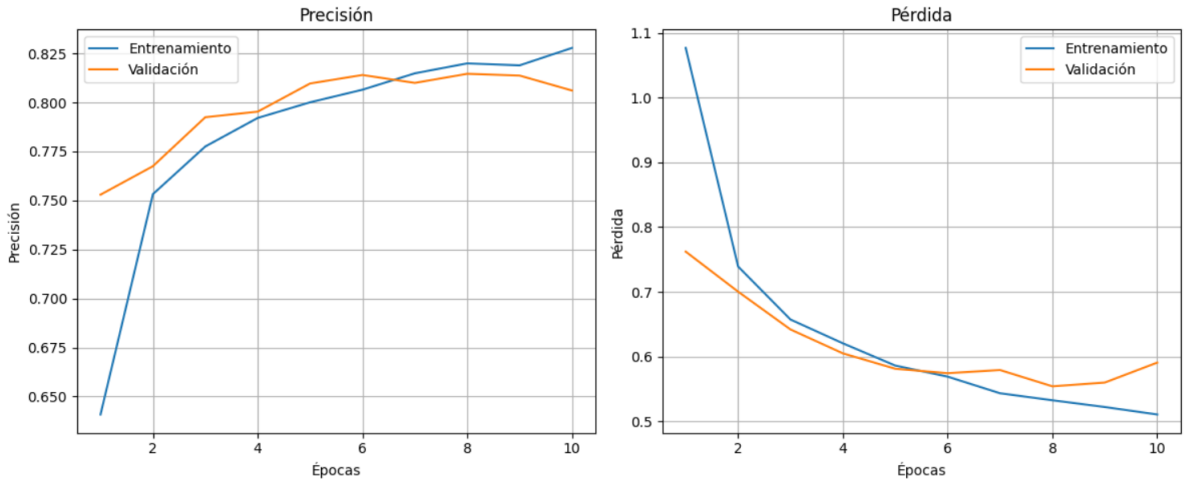

Figura 5.4.: Histórico de métricas del entrenamiento 2 del modelo *MobileNetV2*

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 entrenmaiento 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* en el entrenamiento 2

En general se obtiene un buen rendimiento, existiendo un buen balance entre precisión y sensibilidad en practicamente 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 bastantes 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

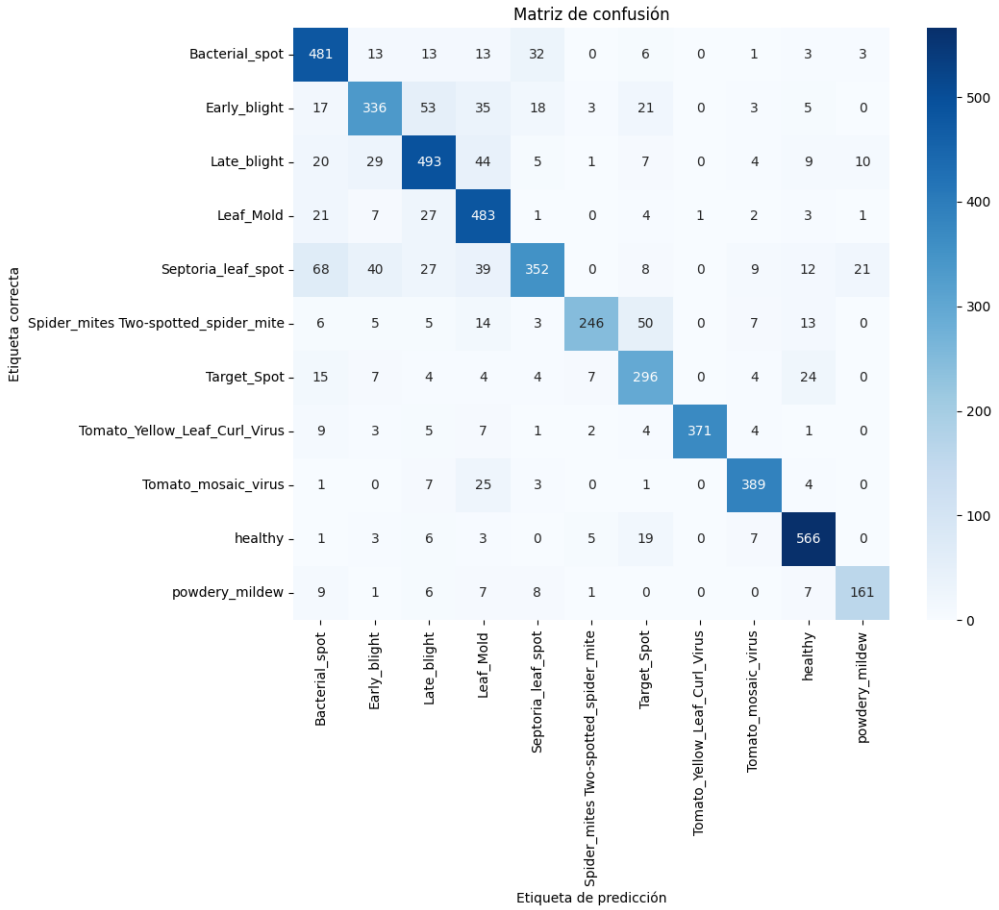

Figura 5.6.: Matriz de confusión del modelo *MobileNetV2* en el entrenamiento 2

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

5.2. Modelo *EfficientNetB0*

En esta sección se van a mostrar los resultados del entrenamiento del modelo descrito en la subsección 4.4.2. 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 5.7.

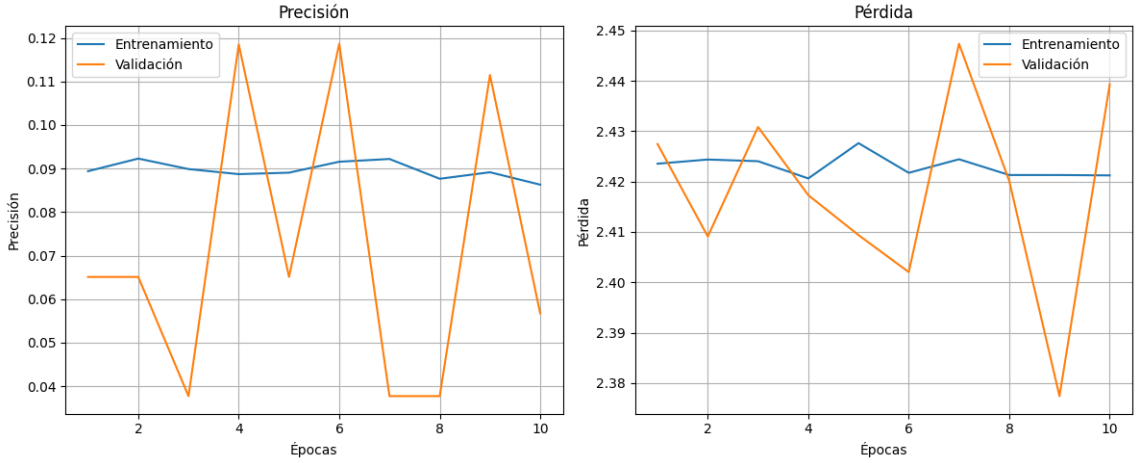

Figura 5.7.: Histórico de métricas del entrenamiento 2 del modelo *EfficientNetB0*

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: 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 practicamente 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*.

5.3. Modelo *NASNetMobile*

En esta sección se van a mostrar los resultados del entrenamiento del modelo descrito en la subsección 4.4.3. A este modelo se han aplicado varios entrenamientos distintos explicados en las subsecciones 5.3.1, .

5.3.1. Entrenamiento 1

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 5.8.

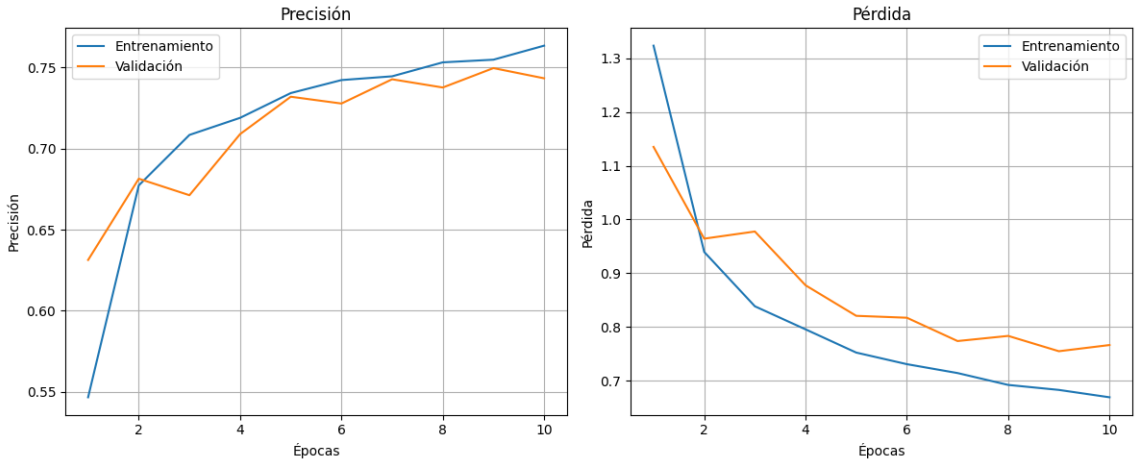

Figura 5.8.: Histórico de métricas del entrenamiento 1 del modelo *NASNetMobile*

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

6. Conclusiones

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

A. Anexo I: Ejemplo de anexo

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. Primer elemento.
2. Segundo elemento
3. Tercer elemento.
    a) Primer subelemento.
    b) Segundo subelemento.
        - Primer punto.
        - Segundo punto.

Bibliografía

A.Huerta-Mora, E., González-Huitrón, V., Rodríguez-Rangel, H., & Amabilis-Sosa, L. E. (2024). Detección de enfermedades foliares con arquitecturas de redes neuronales convolucionales [Consultado el 30 de julio de 2025]. https://rinderesu.com/index.php/rinderesu/article/view/46/50

Gutiérrez, V. M. (2019). Comparación de arquitecturas de redes neuronales convolucionales para la clasificación de enfermedades en tomate [Consultado el 30 de julio de 2025]. https://sedici.unlp.edu.ar/handle/10915/139770

Ing. Agr. Miguel Silva. (2025a, julio). 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

Ing. Agr. Miguel Silva. (2025b, julio). 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

Martínez, M. Y., Molina, M. M., García, N. M., & López, E. V. (2024). *Técnicas de aprendizaje supervisado para la detección y clasificación de enfermedades y defectos en imágenes de frutas: revisión* [Consultado el 30 de julio de 2025]. https://revistas.utb.edu.ec/index.php/magazine/article/view/2330/1983

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

Wikipedia contributors. (2025a, julio). 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. (2025b, julio). *Solanum lycopersicum*. Wikipedia. Consultado el 28 de julio de 2025, desde https://es.wikipedia.org/wiki/Solanum_lycopersicum