SISTEMA AUTOMÁTICO DE CLASIFICACIÓN DEL GRADO DE SEVERIDAD DE LA MANCHA ANILLADA DE FRIJOL USANDO REDES NEURONALES

Lisbeth Lorena Nuñez Rodríguez

Samir Reyes Velásquez

Trabajo de grado para optar por el título de Ingeniero Electrónico

Director

Jaime Guillermo Barrero Pérez

Mg. en potencia eléctrica

Codirectora

Luz Nayibe Garzón Gutierrez

Dra. en ciencias agropecuarias

Universidad Industrial de Santander

Facultad de Ingenierías Fisicomecánicas

Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones

Bucaramanga

2021

Agradecimientos

En este camino llamado vida tuvimos la oportunidad de disfrutar la etapa universitaria desde el pregrado, con una gran carga de sentimientos positivos y negativos, tuvimos buenas y malas notas, experiencias motivadoras y desalentadoras, pero siempre encontramos apoyo y guía en nuestras familias, amigos, compañeros y profesores. Por ello el logro de llegar hasta este punto de finalizar exitosamente nuestra carrera y obtener el título de ingenieros electrónicos, se lo agradecemos a ellos. Gracias por siempre confiar, aconsejar, acompañar y enseñar cosas importantes para continuar nuestro desarrollo personal y profesional. Sin dejar de lado a Dios, agradecemos ser quienes somos y tener la oportunidad de compartir este logro con cada uno de ustedes.

Como agradecimiento especial a nuestro Director y profesor Jaime Barrero, que continuamente brindó guía para culminar este proyecto, y a la Codirectora Luz Garzón por permitir el desarrollo del proyecto de forma interdisciplinaria, que, con apoyo de la Bióloga Miranda, tuvimos ese acompañamiento en esa área para alcanzar el objetivo planteado.

Dedicatoria

Quiero dedicar lo que este proyecto significa para mí a Alba Margoth y Wilson Ovidio, mis padres, por la vida, por su amor, comprensión y apoyo, el cual desde que supieron que llegaría al mundo me han brindado, porque gracias a su dedicación desde cada uno de sus roles me han permitido llegar a ser la mujer que soy y prontamente lograr ser una profesional. También a mi hermano William por motivarme en toda ocasión y ser un gran apoyo en mi vida, a esos amigos y compañeros de estudio por orientarme y ayudarme a lo largo de las diferentes etapas de la carrera.

Lisbeth Nuñez

Este proyecto de investigación lo dedico a mi madre Nelly Velásquez Reyes por haberme apoyado durante toda mi vida, por haberme brindado todo lo que necesité para mi formación, por darme la fuerza para seguir adelante y nunca rendirme, por todo su amor incondicional por el que cada día trato de ser mejor persona.

A mi hermano Kevin Reyes Velásquez por ser un gran apoyo en todo este proceso, por ser el gran hermano que es, por sus consejos y por siempre estar a mi lado cuando lo necesito.

Samir Reyes

Tabla de contenido

Lista de Tablas

Lista de Figuras

Lista de Apéndices

Glosario

CIAT: El Centro Internacional de Agricultura Tropical es una institución de investigación agrícola que fomenta la agricultura eco-eficiente, dirigida a mejorar la competitividad, alcanzar niveles de productividad sostenible y dejar una huella ecológica mínima.

Convolución: Este proceso consiste en tomar "grupos de píxeles cercanos" de la imagen de entrada e ir operando matemáticamente (producto escalar) contra una pequeña matriz que se llama kernel. Ese kernel recorre todas las entradas (de izquierda-derecha, de arriba-abajo) y genera una nueva matriz de salida.

Debian: Debian es una organización dedicada a desarrollar software libre y promocionar los ideales de la comunidad del software libre. El producto principal del proyecto a la fecha es la distribución de software Debian GNU/Linux, la cual incluye Linux como núcleo del sistema operativo, así como miles de aplicaciones pre-empaquetadas.

Firmware: Es el programa básico que controla los circuitos electrónicos de cualquier dispositivo. Este programa es una porción de código encargada de controlar qué es lo que tiene que hacer el hardware de un dispositivo.

Flatten: Es una función que se utiliza para aplanar un tensor, ajustando su dimensión para tener una sola con el total de número de elementos del tensor.

Framework: Es el esquema que se establece y se aprovecha para desarrollar y organizar un software determinado, para escribir código o desarrollar una aplicación de manera más sencilla.

GPIO: *General Purpose Input Output*, es un sistema de entrada y salida de propósito general, es decir, una serie de pines o conexiones que se pueden usar como entradas o salidas para múltiples usos.

GPU: *Graphics Processing Unit*, representa el centro de una tarjeta gráfica, pues se encarga de realizar todos los cálculos complejos y mejorar el rendimiento del computador.

HSB: El modelo de color HSB (*Hue*, *Saturation*, *Brightness*) se basa en la percepción humana del color y describe tres características fundamentales del color: Matiz, Saturación, Brillo.

JPEG: (Join Photograph Expert Group). Un formato de archivo gráfico que se utiliza para mostrar imágenes en color de alta resolución. Las imágenes JPEG aplican un esquema de compresión especificado por el usuario que puede reducir considerablemente los tamaños de archivos grandes.

Jupyter: Jupyter Notebook es una aplicación cliente-servidor que permite crear y compartir documentos web en formato JSON que siguen un esquema versionado y una lista ordenada de celdas de entrada y de salida. Estas celdas albergan código, fórmulas matemáticas y ecuaciones, o contenido multimedia. El programa se ejecuta desde la aplicación web cliente que funciona en cualquier navegador estándar.

Matriz de confusión: La matriz de confusión es una herramienta fundamental para evaluar el desempeño de un algoritmo de clasificación, a partir de un conteo de los aciertos y errores de cada una de las clases en la clasificación.

Morfología: Forma o estructura de algo. En biología es el estudio de la forma de los seres vivos y de su evolución.

Python: Es un lenguaje de programación versátil multiplataforma y multiparadigma que se destaca por su código legible y limpio. Cuenta con una licencia de código abierto que permite su utilización en cualquier escenario.

RGB: *red*, *green*, *blue*. El concepto suele emplearse para referirse a un modelo cromático que consiste en representar distintos colores a partir de la mezcla de estos tres colores primarios. Empleando la luminosidad del rojo, el verde y el azul en diferentes proporciones, se produce el resto de los colores.

Sesgo: Es el error humano, intencional o no intencional que se comete al ejecutar el muestreo y que generalmente es sistemático. En estadística es la diferencia entre el valor del parámetro y su valor esperado.

Tensor: Es un vector o matriz de n-dimensiones que representa todos los tipos de datos. Todos los valores de un tensor contienen un tipo de datos idéntico con una forma conocida (o parcialmente conocida).

Vaina: Envoltura tierna y alargada en la que están encerradas en hilera las semillas de ciertas plantas y que está formada por dos piezas.

Zero padding: Es una técnica que consiste en rellenar la imagen a filtrar con ceros de esta forma no se produce una reducción en el tamaño de la imagen final, se utiliza generalmente para realizar convoluciones.

Resumen

Título: Sistema automático de clasificación del grado de severidad de la mancha anillada de frijol usando redes neuronales [1*]

Autores: Lisbeth Lorena Nuñez Rodríguez, Samir Reyes Velásquez [2**]

Palabras Clave: Entrenamiento, identificación, fitopatología.

Descripción:

Este proyecto presenta el entrenamiento de una red neuronal para clasificar los grados de severidad de la enfermedad denominada mancha anillada en los foliolos de frijol. Se hace una descripción general del frijol, de las principales características de la mancha anillada y la manera en la que se realiza la evaluación de la severidad en los foliolos. Se desarrolló una base de datos con los niveles de severidad según una escala diagramática de intervalos iguales, estos niveles se agruparon en las categorías de alta, baja y media para aumentar las fotos. Se identificaron los patrones de la enfermedad en las imágenes de los foliolos a través de una arquitectura de red convolucional CNN. Se probaron CNNs como ResNet50, MobileNet y MobilenetV2 previamente entrenadas con las que se realizó *transfer learning* para obtener un modelo con menos parámetros, menos tiempo de entrenamiento, brindando mejores resultados que con una red creada desde cero y aplicando conceptos tales como *forward and backward propagation*, funciones de activación e hiperparámetros se logró la clasificación de las tres categorías. El entrenamiento, validación y pruebas de la red se realizaron en los cuadernos de *Jupyter* del entorno gratuito de *Google Colaboratory*. Se lograron precisiones del 63,8% con una red desde cero, 77,1% con la técnica de *transfer learning* y *finetune* a la red MobileNetV2 y un 91,3% usando el *framework* aXeleRate. El uso de los sistemas embebidos como la Maixduino permitieron ver limitaciones de memoria RAM y deficiencia en tiempos de ejecución como en la Raspberry Pi 2 que obtuvo un tiempo promedio de ejecución para evaluar un foliolo de 771,98 ms pues no está diseñada específicamente para aplicaciones IA.

[1*] Trabajo de grado

[2**] Facultad de Ingenierías Fisicomecánicas. Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones. Director: Jaime Guillermo Barrero Pérez, Magíster en potencia eléctrica

Abstract

Title: Automatic system for classifying the degree of severity of Ascochyta blight of common bean using neural networks [3*]

Authors: Lisbeth Lorena Nuñez Rodríguez, Samir Reyes Velásquez [4**]

Key Words: Training, identification, phytopathology.

Description:
This project presents the training of a neural network to classify the grades of severity of the ring spot in bean leaflets. A general description of beans, a general description of the bean, the main characteristics of the ring spot and the way in which the evaluation of the severity in the leaflets is carried out. A database was developed with severity levels according to a diagrammatic scale of equal intervals, these levels were grouped in the categories of high, low and medium to increase the photos. Disease patterns in leaflet images were identified through a convolutional network architecture CNN. Previously trained CNNs such as ResNet50, MobileNet and MobilenetV2 were tested and transfer learning was performed to obtain a model with fewer parameters, less training time, providing better results than with a network created from scratch and applying concepts such as forward and backward propagation, activation functions and hyper parameters, the classification of the three categories was achieved. The training, validation and testing of the network were carried out in Jupyter's notebooks in the free Google Colaboratory environment. Accuracies of 63.8% were achieved with a network from zero, 77.1% with the transfer learning and finetune technique to the MobileNetV2 network and 91.3% using the aXeleRate framework. The use of embedded systems such as Maixduino allowed to see RAM memory limitations and runtime deficiencies as in the Raspberry Pi 2 which obtained an average runtime to evaluate a leaflet of 771.98 ms because it is not specifically designed for AI applications.

[3*] Bachelor Thesis

[4**] Faculty of Physico-Mechanical Engineering. School of Electronics and Electrical engineering and Telecommunications. Advisor: Jaime Guillermo Barrero Pérez, Master in electrical power.

Introducción

Actualmente la inteligencia artificial (IA) se está implementando en diversas áreas de aplicación, lo cual permite el avance y la innovación de quienes hacen uso de ella. Una de las herramientas que más se destacan dentro de IA, son las redes neuronales. Las redes son un modelo entrenado de manera computacional que consta de una cantidad de neuronas artificiales conectadas entre sí para transmitir información y obtener una respuesta ante una entrada, de manera similar como lo hace el cerebro humano. Una de las aplicaciones en la cual son comúnmente utilizadas es la clasificación o detección de patrones mediante imágenes.

En el presente proyecto se vio cómo las redes neuronales, puntualmente las redes neuronales convolucionales, son aplicables en el trabajo de clasificar imágenes, esto se comprobó en el caso de estudio que se desarrolla a lo largo del proyecto, el cual consistió en la clasificación de foliolos de frijol según el grado de severidad de la mancha anillada, con el objetivo de determinar la categoría correspondiente, sea alta, baja o media.

Una red neuronal que clasifica imágenes necesita de una base de datos generalmente amplia que le permita aprender a diferenciar los datos con los que se desean trabajar, por consiguiente, se construyó una carpeta que tiene 3 subconjuntos de datos o *dataset* con las categorías en las cuales se clasificó los foliolos de frijol. Para la tarea de crear, entrenar y validar la red neuronal se utilizó el entorno gratuito de ejecución en línea *Google Colaboratory* que dispone de una GPU (*graphics processing unit*) robusta en la nube, lo que facilita realizar el entrenamiento de la red sin consumir recursos propios del computador que se esté utilizando.

Como resultado del trabajo en este proyecto, se obtuvieron dos modelos de red neuronal los cuales se implementaron a dos sistemas embebidos como lo son la *Raspberry Pi 2B* y la tarjeta *Sipeed Maixduino*, con el fin de ver el rendimiento en estos dispositivos con una

capacidad de procesamiento limitada y hacer la comparación de ejecución en el entorno de ejecución virtual y dichos sistemas embebidos.

1. Objetivos

1.1.Objetivo general

Determinar el grado de severidad de la mancha anillada en los foliolos de frijol usando redes neuronales.

1.2.Objetivos específicos

Construir una base de datos con foliolos de frijol en diferentes niveles de mancha anillada y foliolos sanos para entrenar la red neuronal.

Seleccionar y entrenar una red neuronal que permita clasificar entre diferentes niveles la severidad de la mancha anillada en foliolos de frijol.

Determinar el rendimiento de la red neuronal entrenada cuando se captura la imagen del foliolo de frijol mediante un sistema digital.

2. Frijol común (Phaseolus vulgaris L)

El frijol común se encuentra dentro del género Phaseolus junto con otras especies comestibles (Fraile et al., 2007). Es la leguminosa más importante del mundo pues se considera como un alimento altamente nutritivo al ser una fuente principal de proteínas, aminoácidos, calcio, hierro, fósforo y vitaminas (Ulloa et al., 2011). Según Schwartz y Haverson (2015) plantean que es un componente esencial de la dieta del ser humano, especialmente en regiones como Centroamérica y Sudamérica, aun así, su cultivo se puede ver limitado, pues el fríjol es susceptible al desarrollo de diferentes fitopatologías, la mayoría causadas por hongo, en menor grado, las producidas por bacterias, virus y condiciones ambientales que favorecen el progreso de las enfermedades.

El consumo de frijol se da, principalmente, como grano maduro, aunque también como semilla inmadura o como vegetal, es decir hojas y vainas (Treviño y Rosas, 2013). Según el Núcleo Ambiental S.A.S (2015) se pueden encontrar ‘‘cultivos desde los 900 hasta los 2.700 metros sobre el nivel del mar (msnm) y su producción se desarrolla de forma adecuada en temperaturas promedio entre 15 y 27°C’’ (p. 11).

Según su descripción morfológica, las hojas son de dos tipos: simples y compuestas; las hojas simples caen antes de que la planta esté completamente desarrollada, las hojas compuestas tienen tres foliolos, uno central y dos laterales. La forma de los foliolos tiende a ser de ovalada a triangular, como se puede observar en la siguiente figura:

Figura 1. *Hoja compuesta de frijol*

Nota. Composición estructural de una hoja de frijol. Morfología de la planta de frijol común. Debouck & Hidalgo (1985)

2.1.Producción en Colombia

Según Jara y Giraldo (2016), el 65% del frijol producido es de tipo voluble y el 35% es arbustivo, su siembra se realiza preferiblemente en periodos de abundante lluvia como son normalmente los meses de marzo, abril, septiembre y octubre.

En el país se sembraron un total de 92.412 hectáreas de frijol en 2019, con una producción de 114.408 toneladas, entre los principales departamentos productores del país están Huila, Santander, Antioquia, Nariño y Tolima con un 70.38% de la producción nacional (Fenalce, 2019). En el primer semestre del año 2020 se sembraron 41.542 hectáreas y se obtuvo una producción de 47.447 toneladas (Fenalce, 2020).

2.2.Mancha anillada de frijol

La mancha anillada (mancha de asochyta), también conocida en algunas regiones del país como “la gota”, es una enfermedad que produce más de un ciclo de infección por ciclo del cultivo (The American Phytopathological Society (APS) s.f). Como plantean Schmit y Baodoin (1992) esta enfermedad es ocasionada por cuatro especies de hongos, *Boeremia diversispora, Boeremia exigua, Beremia noackiana* y *Stagonosporopsis hortensis.* Al comienzo los síntomas de esta enfermedad se presentan en las hojas, con lesiones circulares de color oscuro, que al

progresar por lo general forma de anillos más o menos concéntricos como se puede evidenciar en la Figura 2; teniendo en cuenta a Corrales (1985) ‘‘después de un tiempo las manchas se tornan color marrón y en niveles altos de severidad hay necrosis en la planta’’ (p. 170). Las lesiones también se pueden evidenciar en las vainas, pecíolos, pedúnculos y en el tallo, aunque para este proyecto es de interés particular la identificación de la mancha en los foliolos de las hojas.

Figura 2. *Síntomas de la mancha anillada en foliolo de frijol*

Nota. Evidencia de mancha anillada en foliolo de hoja de frijol. Distribución, incidencia y severidad de la mancha anillada (Boeremia spp) del frijol de los departamentos de Antioquia, Tolima y Huila, Colombia. Miranda (2018). (p. 16)

2.2.1. *Método de evaluación de la mancha anillada*

Para evaluar una enfermedad en las plantas se determina la incidencia y la severidad de esta sobre una población de estudio. La incidencia es la relación porcentual entre el número de plantas (ya sea completa o parte de ella) que presenta la enfermedad dentro de un grupo evaluado. La severidad corresponde al área con síntomas de la enfermedad en relación con el área total que se está examinando, para este caso de estudio se indica el área total del foliolo como se ve en la siguiente ecuación planteada por Seem (1984).

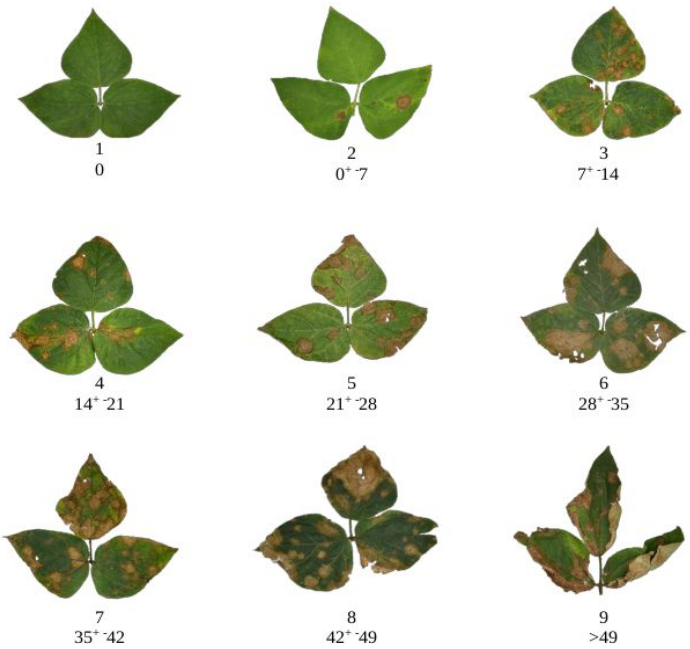
$$Severidad = \frac{Área\ con\ síntomas\ de\ la\ enfermedad}{Área\ total\ del\ foliolo} x100 \quad (1)$$

2.2.2. *Escala de severidad*

Para identificar el nivel de severidad de la mancha anillada en las hojas de frijol, se toma como referencia las escalas diagramáticas elaboradas en el trabajo de investigación Distribución,

incidencia y severidad de la mancha anillada (Boeremia spp) del frijol en los departamentos de Antioquia, Tolima y Huila, Colombia como planteó Miranda (2018), dichas escalas son ilustraciones que representan los distintos grados de severidad de la enfermedad en la hojas (Figura 3) donde se establecen nueve categorías y se delimita el intervalo de porcentaje de severidad para cada categoría.

Figura 3. *Escala diagramática de severidad con intervalos iguales para la mancha anillada en hojas de frijol*



Nota. Presentación de la escala de severidad para mancha anillada en las hojas de frijol. Distribución, incidencia y severidad de la mancha anillada (Boeremia spp) del frijol de los departamentos de Antioquia, Tolima y Huila, Colombia. Miranda (2018). (p. 43)

2.2.3. *Distribución de la mancha anillada en Colombia*

Como plante Miranda (2018) la distribución de la enfermedad en el país es amplia, se han realizado colectas de hojas y/o vainas infectadas en los siguientes departamentos como el Valle del Cauca, Boyacá, Huila, Nariño y Tolima, de los patógenos que producen la enfermedad, los cuales se encuentran en la colección del Centro Internacional de Agricultura Tropical CIAT.

En este mismo trabajo de investigación se cuantificó y se describió gráficamente la incidencia y severidad de la enfermedad en estos departamentos que aportan alta producción de dicha leguminosa en el país, ya que se desconocía el estado de la mancha anillada en estas zonas. En la tabla 1 se muestra de manera muy general parte de los resultados que se obtuvieron en el desarrollo de la investigación.

Tabla 1. *Incidencia y severidad de la mancha anillada en los departamentos de Antioquia, Tolima y Huila.*

Departamento	PE	I	RS
Antioquia	750	94%	5-6
Tolima	660	60%	2-4
Huila	750	51.7%	2-3

Nota. PE: Número de plantas evaluadas, **I**: Incidencia indicada en porcentaje, **RS**: Rango de severidad más frecuentes observados. En cuanto a los valores de severidad alta, en general todos los departamentos presentaron pocas plantas afectadas con valores de 7, 8 y 9. Distribución, incidencia y severidad de la mancha anillada (Boeremia spp) del frijol de los departamentos de Antioquia, Tolima y Huila, Colombia. Miranda (2018).

3. Inteligencia artificial

Se puede definir inteligencia artificial IA como una disciplina relacionada con la teoría de simular de manera computarizada algunas de las facultades intelectuales de los humanos, como son los procesos de percepción a través de los sentidos (visión, audición, etc) y procesos de reconocimiento de patrones, por lo que las aplicaciones más habituales de la IA son el tratamiento de datos y la identificación de modelos en sistemas artificiales (Benítez et al., 2013).

Los autores Russell y Norvig (2004) organizaron la IA en cuatro categorías; dos de ellas se relacionan a la forma de actuar de los humanos: primero, los sistemas que piensan como humanos, por ejemplo, las *redes neuronales artificiales*, y segundo, los sistemas que actúan

como humanos, como los robots. Las otras dos toman como referencia un concepto ideal de inteligencia que llaman racionalidad: a estas categorías pertenecen los sistemas que usan la lógica racional y los sistemas que actúan racionalmente.

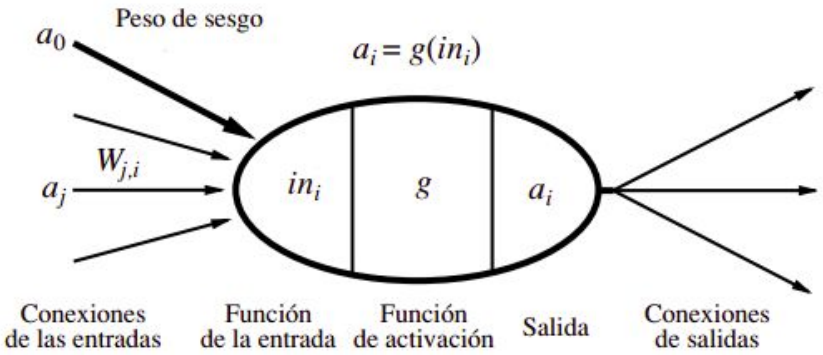
3.1.Redes neuronales

Las redes neuronales son sistemas que intentan simular el comportamiento de las neuronas biológicas, es decir, las células nerviosas que transmiten y procesan la información en el cerebro humano. Kohonen (1989) establece que:

> ‘‘una red neuronal artificial (RNA) está constituida por un conjunto de funciones matemáticas, que están interconectadas en una estructura en paralelo, obedeciendo una organización jerárquica a la cual se le llama *capa*, la cual le permite interactuar con algún sistema de la misma forma en que lo hace un sistema nervioso biológico’’. (p. 77).

Las redes neuronales se componen de *nodos* o *unidades* conectadas a través de conexiones que dirigen la *activación* a_j de una unidad *j* a una unidad *i*. Cada conexión tiene asociado un *peso de sesgo* $W_{j,i}$ que determina la fuerza de la conexión (Figura 4) (Russell y Norvig, 2004).

Figura 4. *Modelo matemático sencillo para una neurona*



Nota. Presentación de modelo matemático sencillo para una neurona. Adaptado de (Russell y Norvig, 2004, p. 839).

Primero cada unidad calcula la suma total de cada una de sus entradas:

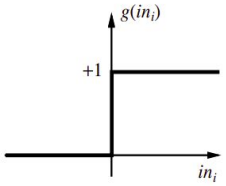
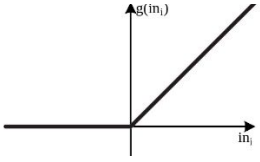
$$in_i = \sum_{j=0}^{n} w_{j,i} a_j \tag{2}$$

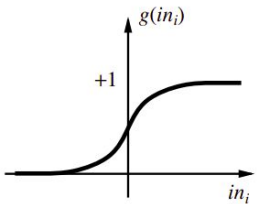
Luego se debe aplicar una *función de activación g* a esta suma para finalmente producir una salida de la unidad (ver ecuación 3). La función de activación debe ser no lineal y se utiliza para activar la unidad cuando se tengan entradas correctas, y desactivarla si las entradas son erróneas (Russell y Norvig, 2004).

$$a_i = g\left(\sum_{j=0}^{n} w_{j,i} a_j\right) \tag{3}$$

Existen diferentes funciones de activación, pero entre las más comunes en el campo de las RNAs se encuentran: la función de activación umbral o escalón, la unidad lineal rectificada (ReLU) y la función sigmoide, (Tabla 2). La función de activación se elige de acuerdo con la tarea que realiza la neurona.

Tabla 2. *Funciones de activación comunes para redes neuronales artificiales.*

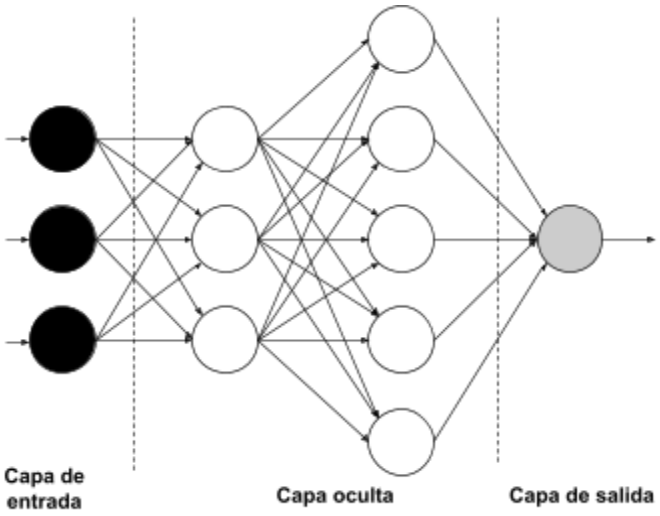
	Función	Gráfica
Umbral	$g(\text{in}_i) = \begin{cases} 0 & \text{si } \text{in}_i < 0 \\ 1 & \text{si } \text{in}_i \geq 0 \end{cases}$	
ReLU	$f(in_i)=max(0, in_i)$	

Sigmoide	$g(\text{in}_i) = \frac{1}{1 + e^{-\text{in}_i}}$	

Nota. Elaboración propia con base de datos de cursos realizados e información presentada anteriormente

El esquema de una RNA generalmente se estructura en tres niveles (Figura 5); en el primer nivel se encuentra la capa de entrada, esta capa es la encargada de recibir los datos de entrada que serán procesados, en el segundo nivel se encuentra la capa oculta, se llaman oculta porque no tienen relación directa con la información de entrada o con la de salida (Bórquez y Villanueva, 2006), en esta capa se procesa la información para finalmente dar respuesta en la última capa, la capa de salida. Para el ejemplo de la (Figura 5) se ve un esquema de red neuronal multicapa; tres unidades en la capa de entrada, dos capas ocultas con tres y cinco unidades respectivamente y una unidad en la capa de salida.

Figura 5. *Esquema de una red neuronal multicapa*

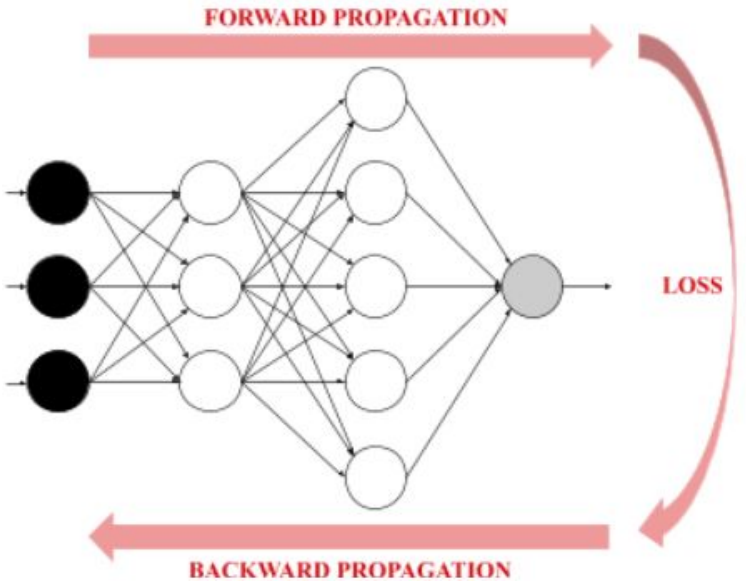


Nota. Estructura de una red neural con tres entradas, dos capas ocultas y una salida. Adaptación de Predicción de signo a tres semanas de la acción caterpillar con redes neuronales. Bórquez y Villanueva (2006)

3.1.1. Aprendizaje de una red neuronal

Entrenar una red neuronal es hacer que la red adecúe los valores de los parámetros de pesos y sesgos que tiene asociados cada conexión entre neuronas, esta es la parte fundamental de *Deep Learning*, este proceso de aprendizaje es un proceso iterativo de ida y vuelta por cada capa que conforma la arquitectura del modelo de la red (Figura 6), como planeta Andrew (2020) la cantidad de veces (*epoch*) necesarias hasta obtener un buen modelo.

Figura 6. *Propagación hacia adelante y hacia atrás en una red neuronal*



Nota. Adaptación de cursos. Ciclo de entrenamiento, se propaga hacia adelante, los datos se procesan y se obtiene un valor de pérdida y se propaga nuevamente hacia atrás. Adaptación de Predicción de signo a tres semanas de la acción caterpillar con redes neuronales. Bórquez y Villanueva (2006)

La propagación hacia adelante es el proceso conocido como *forward propagation,* este proceso es el encargado de exponer los datos de entrenamiento y cruzarlos por toda la red hacia adelante, de esta forma todas las neuronas aplican sus cálculos y transformaciones a la

información de neurona a neurona, para que finalmente las predicciones sean calculadas. Se usa una función de pérdida o error, *loss*, para comparar y medir si fue bueno el resultado de predicción en comparación con el resultado correcto, se espera que idealmente el valor de pérdida fuera cero y la comparación entre los resultados fuera la misma, por eso, a lo largo del entrenamiento de la red neuronal se van modificando y adecuando los pesos para lograr una mejor predicción (Andrew, 2020).

Calculado el valor de pérdida, la información se propaga hacia atrás, este proceso iterativo se conoce como *back propagation.* La información ahora cruza las capas de la red en sentido contrario, desde la última capa hasta la primera, la cantidad de señal de pérdida que recibe cada una de las neuronas depende de la contribución que haya aportado a la salida. Con esta propagación se pueden volver ajustar los pesos entre las neuronas, de tal forma que la pérdida se acerque más a cero la próxima vez que se realice una predicción. Para ir cambiando los valores de los pesos se usa un optimizador que va actualizando los valores, los optimizadores más usados en problemas de clasificación están basados en el *descenso de gradiente* según Khandelwal (2019) como los siguientes:

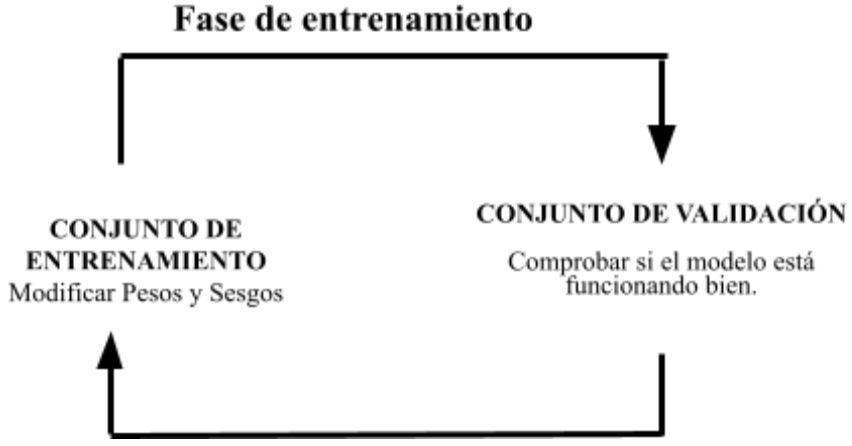
- *Adagrad*; en este optimizador algunos de los pesos de la red tendrán diferentes *learning rates* o tasa de aprendizaje que otros, aunque hace el entrenamiento de la red muy lento.
- *RMSprop*; es la versión especial de Adagrad que busca corregir los problemas que tiene el optimizador Adagrad.
- *Adam*; este optimizador es que reduce la velocidad para una búsqueda segura del mínimo de la función de pérdida, de manera que no se salte por encima de él al ir demasiado rápido.

3.1.2. Entrenamiento, validación y prueba

Para construir, entrenar y evaluar cualquier modelo de red neuronal, se debe utilizar una base de datos, generalmente amplía, según sea su aplicación. Esta única base de datos se suele dividir en otros tres subconjuntos de datos o *datasets* en el proceso de desarrollo de una red neuronal. Estos *datasets* se clasifican de la siguiente manera: datos de entrenamiento (*training*), datos de validación (*validation*) y datos de prueba (*test*) (Ng Andrew, 2020).

Los datos de entrenamiento son aquellos datos que la red neuronal utiliza para ajustar el modelo y modificar los parámetros de la red (*pesos y sesgos*). Estos datos están identificados con sus respectivas etiquetas para que el modelo aprenda de forma iterativa. Los datos de validación son aquellos que proporcionan una evaluación imparcial del modelo entrenado con los datos de entrenamiento, la red neuronal no debe usar el conjunto de validación para decidir sus pesos y sesgos, y saber si se está sobreajustando el entrenamiento, ver Figura 7. Finalmente, los datos de prueba son aquellos que son usados para proporcionar una evaluación imparcial del modelo final.

Figura 7. *Conjunto de datos para entrenar una red neuronal*



Nota. Elaboración propia con base en datos de curso virtual: Intro to TensorFlow for Deep Learning. (Hyttsten, M., Delgado, J., & Bailey, P., 2020).

La proporción de la cantidad de los datos para entreno y validación, generalmente es de un 70 y 30% respectivamente, más aparte una cantidad de datos para realizar las pruebas de la

red. Los datos para entrenamiento siempre será mayor, ya que según Hyttsten et al., (2020) ‘‘es el conjunto más importante para el proceso de aprendizaje del modelo de la red’’ (p.16).

3.1.3. Hiperparámetros

Los parámetros usados en una red neuronal, son variables de configuración internas propias del modelo, las cuales son estimadas con los datos e información de la red. Los hiperparámetros son variables que se configuran de manera externa y que no pueden ser calculadas con los datos propios de la red, sino que son configurados por las personas que estén desarrollando el modelo antes de la fase de entrenamiento, para configurar el algoritmo de aprendizaje del mismo. Algunos de los principales hiperparámetros planteados por Torres (2018) son los siguientes:

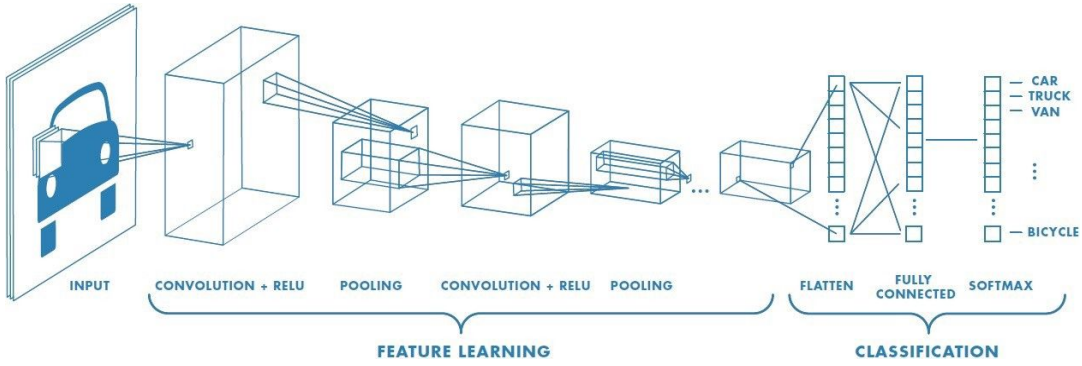
- Número de *epochs*; representa el número de las veces que los datos pasan por la red neuronal durante el proceso de entrenamiento o veces que se repite el ciclo mostrado en la (Figura 6).
- *Batch size*; los datos de entrenamiento se pueden dividir en lotes o *batchs*. Este hiperparámetro indica el número de muestras que hay en cada lote en cada una de las iteraciones en el entrenamiento.
- *Learning rate*; establece el paso en cada iteración del entrenamiento para alcanzar el mínimo de una función de *loss*. Es importante elegir adecuadamente el valor de este hiperparámetro, pues de ser muy pequeño el aprendizaje será lento, en cambio sí es muy grande puede que se salte el valor mínimo buscado para *loss.*

3.1.4. Redes neuronales convolucionales

La red neuronal convolucional (CNN *Convolutional neural network*) es una red neuronal multicapa que tiene una topología capaz de trabajar con entradas de más de una dimensión, para

el caso de estudio las imágenes 2D. Según Pattanayak (2017) las CNN se basan en la operación matemática de *convolució*n entre matrices, con el fin de realizar una detección de características de la imagen basándose en la aplicación de un determinado filtro. La CNN agrupa a las capas en detección de características y en *clasificación* (Figura 8); las capas de detección de características desarrollan una de las tres capas de operaciones: capa de *convolución*, capa de reducción (*pooling* o *subsampling*) o capa de clasificación (ReLU). En la capa de clasificación se utilizan capas totalmente conectadas (*full connected*) y la función de activación *softmax* (MathWorks, 2017).

Figura 8. *Estructura de una Red Neuronal Convolucional*

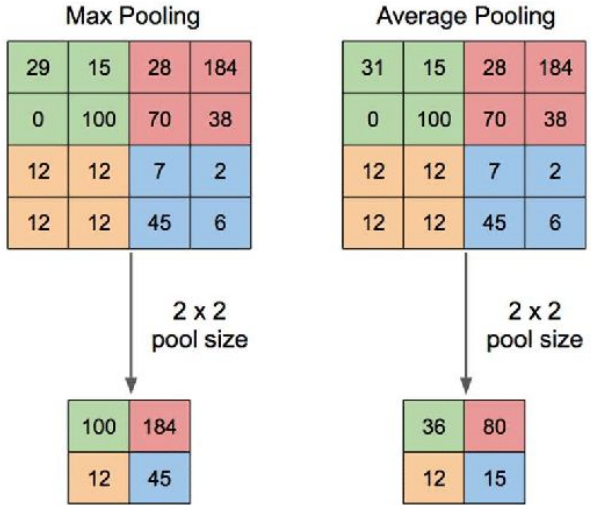


Nota. Descripción de la estructura de la Red Neuronal Convolucional con sus características y clasificación. A comprehensive guide to convolutional neural networks. Saha (2018).

Para una aplicación en imágenes la idea de una capa de convolución es crear otra cuadrícula o matriz de números llamada núcleo o filtro para realizar circunvolución en cada pixel de la imagen, si se hace convolución en imágenes a color, por ejemplo, RGB, se debe convolucionar cada canal de color con su propio filtro bidimensional, la salida depende del número de filtros que se utilizan, esto definirá una nueva profundidad en la imagen convolucionada. La capa de *pooling* es una técnica de agrupación diseñada para ser utilizada cuando la entrada de la red tiene demasiada información, con el fin de hacer una reducción de la

misma, el tipo de *pooling* más utilizado es el llamado *max pooling*, el cual se encarga de agrupar diferentes valores de píxeles escogiendo el de mayor valor como salida, eliminando los restantes (Hyttsten et al., 2020). Otro tipo de *pooling* utilizado es el *average pooling* que consiste en agrupar píxeles y calcular el valor promedio de los mismos. Un ejemplo de *max pooling* y *average pooling* se muestra en la (Figura 9), donde se utiliza una matriz de agrupación de tamaño 2x2 a cada una de las entradas y se reduce la información, pasando de entradas de tamaño 4x4 a tamaño 2x2.

Figura 9. *Tipos de pooling.*

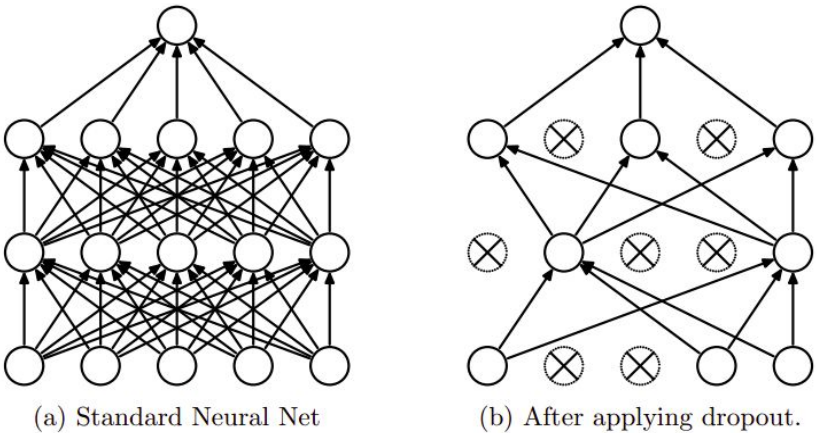


Nota. Tipos de Pooling. Izquierda: aplicación de Max pooling. Derecha: aplicación de Average pooling. Application of transfer learning using convolutional neural network method for early detection of terry's nail. In Journal of Physics: Conference Series. Yani (2019). (p. 3).

La capa *fully connected* o totalmente conectada se encarga de generar una salida monodimensional y de realizar clasificación o razonamientos de alto nivel puesto que todas las neuronas de esa capa con la anterior están conectadas. Esta capa utiliza un parámetro de gran utilidad, el *Dropout*, Según Srivastava et al., plantean que desconecta neuronas aleatoriamente en cada época para permitir que otras intensifiquen su entrenamiento (2014), este parámetro permite reducir el *overfitting*, o *sobreentrenamiento*, de una red como se presenta en la (Figura 10).

Figura 10. *Modelo de red neuronal de Dropout*

.



Nota. Representación de modelo de red neuronal de Dropout, *(a) Red neuronal estándar con 2 capas* ocultas. (b) Red producida al aplicar *Dropout* a la red de la izquierda. Se han eliminado las unidades cruzadas. Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning. Srivastava et al., (2014).

Para el desarrollo de este proyecto se estudiaron y utilizaron 3 arquitecturas de redes convolucionales que se describen a continuación:

3.1.4.1. Red Restnet50.

Resnet es una red neuronal convolucional usada para la clasificación de objetos mediante imágenes. De manera general las imágenes de entrada son de tamaño 224x224, en la (Figura 11) se resalta entre líneas rojas la arquitectura correspondiente a la red *ResNet50 que* posee 50 capas en total; consta inicialmente de una capa convolucional compuesta de 64 filtros kernel, seguido hay una capa de *max pooling*, luego se tienen 4 bloques que se repiten 3, 4, 6 y 3 veces respectivamente, conformados por filtros kernel de diferentes tamaños. Finalmente se tiene la operación de reducción mediante el *average pooling*, una capa totalmente conectada con 1000 nodos y con función de activación *softmax* que es la indicada para hacer clasificación de imágenes. Esta red cuenta con aproximadamente 23 millones de parámetros.

Figura 11. *Estructura general de las redes ResNet*

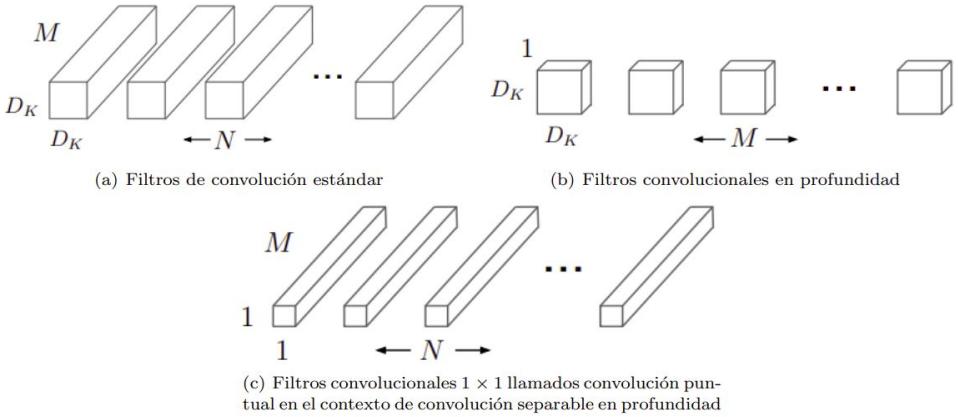
layer name	output size	18-layer	34-layer	50-layer	101-layer
conv1	112×112	7×7, 64, stride 2			
conv2_x	56×56	3×3 max pool, stride 2			
		[3×3, 64; 3×3, 64] ×2	[3×3, 64; 3×3, 64] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3
conv3_x	28×28	[3×3, 128; 3×3, 128] ×2	[3×3, 128; 3×3, 128] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4
conv4_x	14×14	[3×3, 256; 3×3, 256] ×2	[3×3, 256; 3×3, 256] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×23
conv5_x	7×7	[3×3, 512; 3×3, 512] ×2	[3×3, 512; 3×3, 512] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3
	1×1	average pool, 1000-d fc, softmax			

Nota. Estructura general de las redes ResNet. La columna enmarcada en rojo indica la red ResNet50. How to interpret ResNet50 Layer Types (2018).

3.1.4.2. Red MobileNet.

Como plantean Howard et al., (2017), la red MobileNet se basa en convoluciones separables en profundidad, las cuales son una combinación de convoluciones en profundidad (*Depthwise*) y puntuales (*Pointwise*) 1x1. Inicialmente la convolución profunda aplica un solo filtro a cada canal de entrada. Luego se aplica una convolución de 1x1 para combinar las salidas con la de profundidad. Esto se hace con el objetivo de dividir en dos capas separables, una para filtrar y una para combinar, lo que tiene el efecto de reducir drásticamente el cálculo y el tamaño del modelo de la red. La (Figura 12) muestra cómo una convolución estándar 12(a) se factoriza en una de profundidad 12(b) y una puntual de 1x1 12(c), donde D_K es la dimensión del kernel que se asume cuadrado, M y N es el número de canales de entrada y salida.

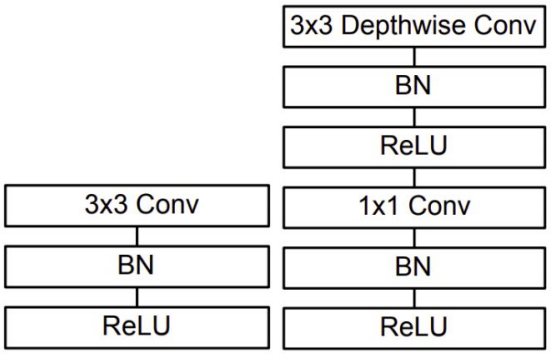
Figura 12. *Filtro separable en profundidad.*



Nota. Representación del filtro separable en profundidad, *(a) Convolución estándar. (b) Convolución en profundidad. (c) Convolución puntual.* MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. Howard et al., (2017).

En la arquitectura de la MobileNet, la primera capa es como se muestra en la Figura 13 (a); consta de una convolución estándar seguida de una normalización por lotes (*Batchnorm*) y la función de activación ReLU. Luego se realizan las capas de convoluciones separables en profundidad como se muestra en la (Figura 13) (b); una convolución *depthwise* y una convolución *pointwise*, seguidas de *batchnorm* y la función de activación ReLU. Finalmente está la capa de *average pooling* que reduce la resolución espacial a 1, una capa totalmente conectada (*fully connected*) y la función de clasificación *Softmax.* La descripción general de la arquitectura de la red MobileNet se muestra en la tabla de la (Figura 14). Esta red tiene un poco más de 3 millones de parámetros.

Figura 13. *Capas convolucionales*



Nota. Presentación de capas convolucionales, (a) Convolución estándar. (b) Convolución separable en profundidad. Elaboración propia con base de datos de MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. (Howard et al., 2017).

Figura 14. *Arquitectura de MobileNet*

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

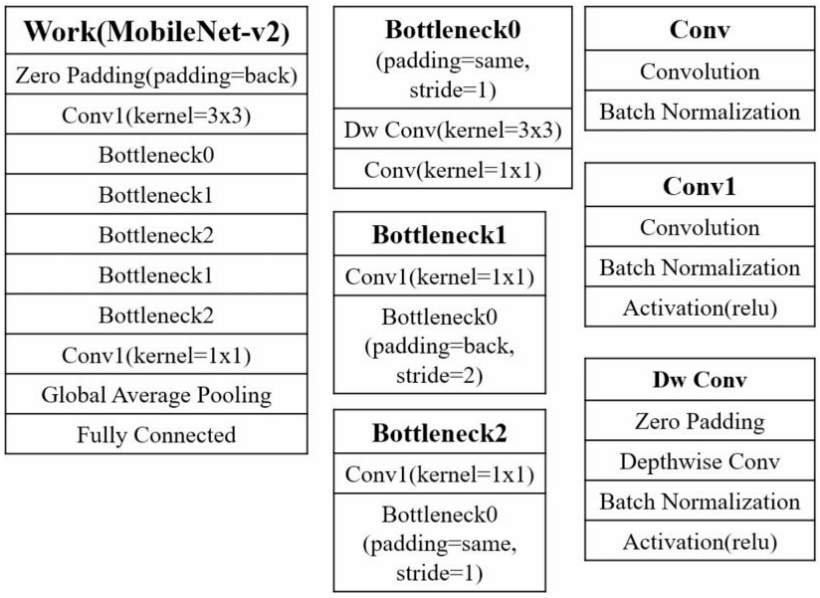
Nota. Transfer learning using Mobilenet and keras. Culfaz (2018)

3.1.4.3. Red MobileNetV2.

La arquitectura de la red MobileNetV2 es una mejora de la MobileNet mencionada anteriormente, mejorando el tamaño del modelo. Como se muestra en la (Figura 15) la

arquitectura de red incluye capas de relleno cero (*zero padding*), una capa de convolución estándar, capas de *batchnorm*, capas de activación, capas de cuello de botella (*bottleneck*), capas de convolución *pointwise*, una capa de *average pooling* y una capa *fully connected*. Esta arquitectura de red tiene alrededor de 2 millones de parámetros lo que la hace menos pesada en comparación con la MobileNet.

Figura 15. *Arquitectura de la red MobileNetV2*



Nota. Using Quantization-Aware Training Technique with Post-Training Fine-Tuning Quantization to Implement a MobileNet Hardware Accelerator. (Chung et al., 2020)

Esta red fue utilizada para clasificar 1000 imágenes de diferentes objetos, la base de datos empleada para el entrenamiento se encuentra en *ImageNet* un repositorio de imágenes con miles de fotos que pueden ser utilizadas por el público.

3.1.5. Transfer learning

Es una técnica de aprendizaje por transferencia utilizada en *deep learning* que consiste en adaptar y hacer uso de alguna arquitectura de red neuronal grande y previamente entrenada para solucionar un problema, con el fin de emplear toda la información que ya posee, como reconocimiento de figuras, líneas, colores, entre otras, y aplicarla a un nuevo problema relacionado. Esta técnica solo funciona en *deep learning* si las características aprendidas del modelo ya entrenado son generales. Utilizar *transfer learning* brinda la ventaja de disminuir el tiempo de entrenamiento en un nuevo modelo de red neuronal, ya que se pueden reutilizar las capas y sus pesos como punto de partida para el proceso de entrenamiento y adaptarse en respuesta al nuevo problema (Brownlee, 2017). Se puede hacer uso total o parcial del modelo entrenado y opcionalmente en casos generales se hace una sintonización del modelo o *Finetune* que consiste en adaptar el modelo en las últimas capas y reentrenar con un el *dataset* particular para el nuevo problema.

4. Sistemas de ejecución

4.1.Google Colaboratory

Google Colaboratory, conocido también como *Google Colab*, como lo menciona Thakur (2019)

> ‘‘es un entorno de libre uso de *Jupyter Notebook* que permite la creación y ejecución de cuadernos con código *Python* totalmente en la nube y además da acceso gratuito a GPUs simplemente cambiando el entorno de ejecución, lo que brinda recursos suficientemente potentes para llevar a cabo trabajos costosos tanto en tiempo como en potencia’’.

Colab tiene como objetivo difundir la educación y la investigación sobre *machine learning* e IA, por tanto, cuenta con las bibliotecas esenciales para realizar redes neuronales,

como TensorFlow y Keras (Carneiro et al., 2018). También está integrado con Google Drive, lo cual permite compartir y editar el mismo cuaderno con más personas.

4.2.Raspberry Pi 2 Model B V1.1

La Raspberry Pi para Pajankar et al., (2016)

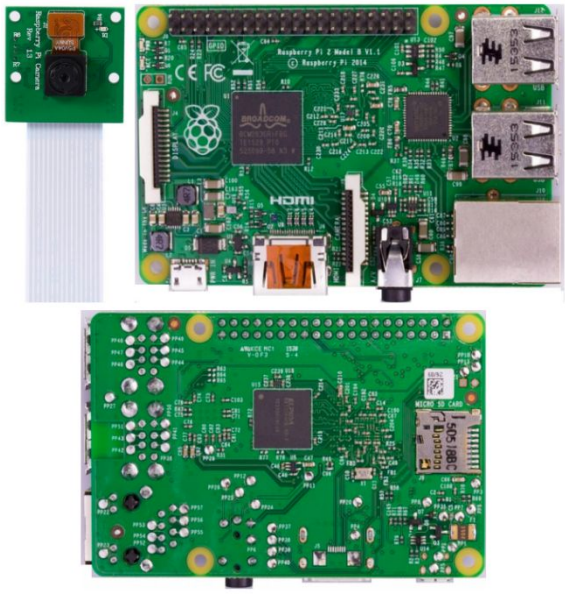
> ‘‘es una pequeña computadora de placa única, de bajo costo que incluye un procesador, *Random Access Memory* (RAM), entrada/salida (I/O) y puertos de red para conectarse a más dispositivos, funciona de manera similar a un computador estándar para lo cual necesita una fuente de alimentación, un teclado para recibir la entrada de comandos y una unidad de visualización o monitor’’ (p.36).

Es la placa más conocida en el mundo pues fue diseñada para la enseñanza de computación en las escuelas, aunque no es una placa de uso específico para aplicaciones con redes neuronales, puede ejecutar aplicaciones de *deep learning* mediante el uso de la librería *OpenCV* (*Open Computer Vision*) la cual es una librería para visión artificial y permite utilizarse en el lenguaje de programación *Python*. En general las placas Raspberry Pi se pueden programar en el *software Raspbian OS* de código abierto, el cual es una variante de *Debian* que proporciona mejor rendimiento en las placas, además cuenta con más de 35.000 paquetes y también muchos programas precompilados (Raji, et al., 2019).

Hasta la actualidad se han sacado varios modelos y versiones de los mismos, el más moderno es la Raspberry Pi 4 modelo B con hasta 8 Gbytes de memoria, aunque para el desarrollo de este proyecto se utilizó la Raspberry pi 2 modelo B con procesador BCM2836 de cuatro núcleos ARM Cortex-A7 a 32 *bits* y frecuencia de 900 MHz (Pajankar et al,. 2016), en la Tabla 3 se puede ver más información sobre la placa. Se utilizó el monitor y un teclado de

computador para interactuar y programar la Raspberry, adicional a esto se adquirió un módulo de cámara de 5 megapíxeles OV5647 con resolución de 2592x1944 (Figura 16).

Figura 16. *Vista frontal y posterior de Raspberry Pi*



Nota: Raspberry Pi Camera 1.3 Raspberry Pi 2 Model B vista frontal y posterior. Elaboración propia.

Tabla 3. *Información de la placa Raspberry Pi 2 Model B V1.1*

Procesador Broadcom BCM2836 quad-core ARM Cortex-A7
Memoria (compartido con GPU) 1GB SDRAM
4 puertos USB 2.0
Puerto HDMI
Conector *Display Serial Interface* DSI
Conector *Camera Serial Interface* CSI
Ranura para tarjeta Micro SD
Ethernet (RJ-45) de 10/100 Mbit/s

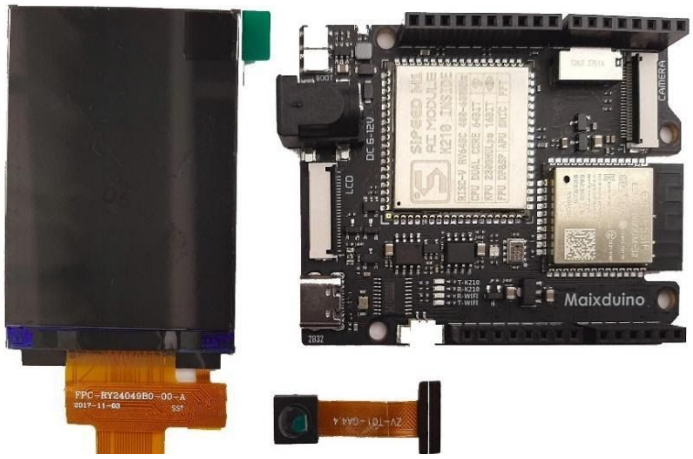
Alimentación 5V Micro USB o GPIO
Consumo energético 800mA (4W)
Precio de tarjeta con cámara 52.98 USD

Nota. Elaboración propia

4.3. Sipeed Maixduino

El prototipo de desarrollo de Maixduino de *Sipeed* tiene un procesador de doble núcleo CPU RISC-V que trabaja con un sistema de 64 *bits*, frecuencia de 400 MHz, *Neural Network Processor* o Procesador de Red Neuronal (KPU) y *Floating-Point Processor* o Procesador Punto Flotante (FPU), está diseñada para aplicaciones de Inteligencia Artificial e Internet de las cosas *(AI + IoT)*. La placa de Maixduino fue diseñada en un factor de forma Arduino Uno, con módulo ESP32 para conectividad *WiFi* y *Bluetooth*, posee también un módulo MAIX AI en el borde o *Edge*; esto es para mover los modelos de IA de la nube a los dispositivos en el borde de la red, donde se pueden ejecutar más rápido mejorando el tiempo de respuesta y a un menor costo. El procesador *Kendryte* K210 es un SOC (*System on a chip*) y es el microcontrolador central de esta placa de desarrollo que trabaja con código de programación abierto, puede ser programada a través de *MicroPython* que es una implementación sencilla del lenguaje de programación *Python 3* el cual facilita la programación en *hardware*, *MaixPy* llevó *MicroPython* al procesador K210, este lenguaje se puede ejecutar desde el terminal serie sin necesidad de una IDE, aunque la IDE como plantea Sipeed (2018) facilita la edición del *script* y el manejo de archivos en la placa de desarrollo. También se puede programar a través de código de programación de Arduino y su entorno Arduino IDE. Para el desarrollo del clasificador del grado de severidad de la mancha anillada de frijol, se adquirió el *kit* de Maixduino, ver Figura 17; este *kit* cuenta con una cámara OV2640 con conector de 24 pines, una pantalla LCD de 2.4 pulgadas y resolución 320x240. En la Tabla 4 se presenta información complementaria de la placa Maixduino.

Figura 17. *Kit Maixduino de Sipeed*



Nota. Elaboración propia

Tabla 4. *Información de la tarjeta Maixduino*

Procesador Kendryte K210
Memoria RAM: 8 Mbytes
Memoria Flash: 16 Mbytes
Ranura para tarjeta Micro SD
Micrófono MEMS integrado
Conector LCD 24P
Conector cámara 24P
Conector DC Input: 6-12 V, Output: 5V 1.2A
Modulo ESP32 2.4G WiFi Bluetooth 4.2
Conector USB Tipo C
Precio con cámara 23.9 USD

Nota. Elaboración propia con base de datos de (Sipeed, 2018)

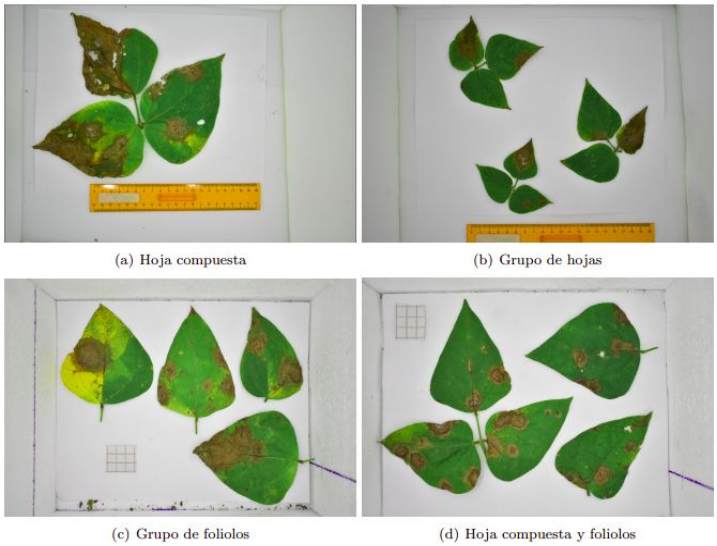
5. Metodología de desarrollo

5.1. Base de datos

5.1.1. Adquisición de fotos

Para construir la base de datos con los foliolos de frijol en diferentes niveles de la mancha anillada con la que se entrena la red neuronal, inicialmente se contaba con una cantidad de fotos de hojas que quedaron como parte de los resultados en el trabajo de investigación: Distribución, incidencia y severidad de la mancha anillada (Boeremia spp.) del frijol en los departamentos de Antioquia, Tolima y Huila, Colombia (Miranda, 2018). Estas fotos fueron tomadas en campo bajo condiciones controladas; luz, distancia y fondo, con el fin de obtener un buen procesamiento de las imágenes en el programa *ImageJ*. En este paquete de fotos hay fotos de hojas compuestas, grupos de hojas compuestas, grupos de foliolos, hoja compuesta y foliolos, véase la (Figura 18). Para aumentar las fotos, aunque teniendo en cuenta las condiciones de pandemia del COVID-19, se buscaron cultivos de frijol cercanos a nuestro lugar de residencia, para adquirir la mayor cantidad de fotos posible, se encontró un cultivo afectado por la fitopatología en la zona rural de la Chiquinquirá Boyacá, Colombia, no fueron encontrados muchos cultivos con las condiciones necesarias puesto que estos son fumigados y tratados para evitar patógenos que los afecten, y de manera general el cultivo de frijol se presenta por temporadas. Estas fotos también fueron tomadas bajo condiciones controladas y en grupos, como se ve en la (Figura 18) (c).

Figura 18. *Clasificación de hojas*



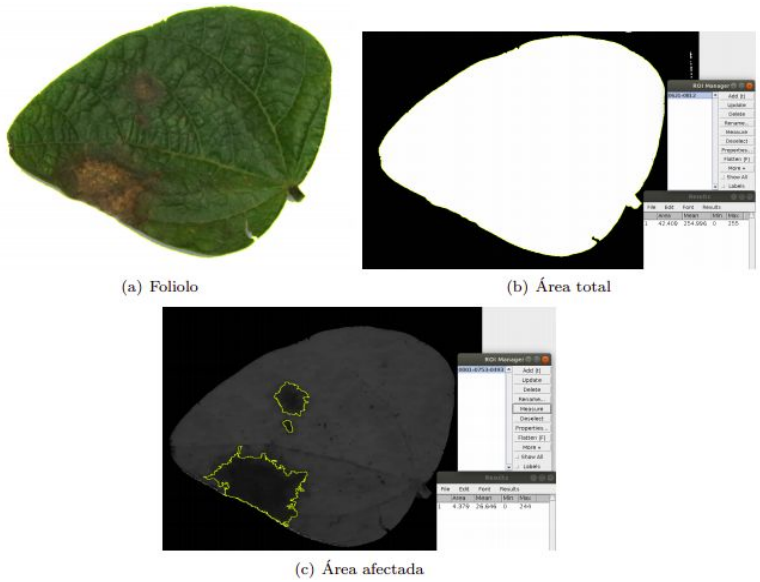
Nota. Ejemplo de fotos sin evaluar ni procesar. (a) Hoja compuesta. (b) Grupo de hojas. (c) Grupo de foliolos. (d) Hoja compuesta y foliolos. Elaboración propia con base de datos de base de datos de Distribución, incidencia y severidad de la mancha anillada (Boeremia spp.) del frijol en los departamentos de Antioquia, Tolima y Huila, Colombia. (Miranda, 2018)

5.1.2. Procesamiento de las imágenes y evaluación de los foliolos

Se tomó la decisión de separar las fotos en foliolos individuales y hacer su respectiva evaluación, con el fin de aumentar la cantidad de imágenes para entrenar la red neuronal. El software que se utilizó para procesar las imágenes fue el software de código abierto de procesamiento de imagen digital *ImageJ* Para calcular la severidad de la mancha anillada se tomó la medida de área total del foliolo y área afectada por la enfermedad, Ecuación 1, primero se debe configurar el programa desde la barra de herramientas de la siguiente manera: *Analyze-> Set Measurements-> Area*, luego establecer una referencia, en este caso una distancia tomada con la herramienta de línea recta, los pasos para establecer la referencia son: *Analyze-> Set Scale ->Known distance-> Global*, para el caso de las fotos como las de la Figura 18, se usó 1 cm como referencia ya que se puede ver un segmento de regla o unos recuadros en las fotos que indican la medida, esto se realiza para saber cuántos pixeles por cm hay en cualquier segmento

que se quiera medir de esa foto. Para calcular el área total se utilizó una técnica de binarización para segmentar los foliolos del resto de la imagen, en este proceso se realizan los siguientes pasos: *Process-> Binary-> Make Binary->* seleccionar el segmento del foliolo con la herramienta de varita (*Wand tool*)*-> Analyze-> Tools-> ROI Manager-> Add-> Measure*, Figura 19 (b). Para calcular el área afectada por la enfermedad se establece la imagen a color: *File-> Revert*, ahora se debe identificar las zonas del foliolo afectadas por la enfermedad, para esto se puede cambiar el formato de la imagen por HSB: *Image-> Type-> HSB Stack->* seleccionar las zonas afectas con *Wand tool-> Analyze-> Tools-> ROI Manager-> Add-> Measure*, (Figura 19) (c).

Figura 19. *Ejemplo de evaluación de severidad mancha anillada*



Nota. Ejemplo de evaluación de la severidad de la mancha anillada en un foliolo de frijol. (a) Imagen RGB. (b) Imagen binarizada. (c) Imagen HSB. Elaboración propia.

5.1.3. Estructura de los datos

Como se muestra en la (Figura 3), hay nueve grados de severidad de la mancha anillada, pero se tomó la decisión de agrupar los nueve grados en tres categorías; alta, media y baja. La agrupación de los nueve grados permite que haya más imágenes en cada una de las categorías

con las que se entrena la red neuronal y obtener la mejor precisión posible. Los datos quedaron estructurados de la siguiente manera:

Tabla 5. *Estructura de la base de datos para el entrenamiento, validación y prueba de la red neuronal*

Conjunto	Categoría	Imágenes
Entrenamiento	Alta	286
	Baja	290
	Media	297
Validación	Alta	71
	Baja	73
	Media	74
Prueba	Alta	25
	Baja	20
	Media	28
Total		1164

Nota. Elaboración propia

5.2. Entrenamiento y validación de la red neuronal

Para iniciar el trabajo de programación y entrenamiento de la red, era necesario identificar qué tipo de red neuronal se adapta mejor al objetivo de este proyecto, como se explicó en el capítulo 3; cuando se tiene como entrada imágenes, es apropiado trabajar una red neuronal convolucional *CNN*, posterior a esto se debe importar todas las librerías y funciones necesarias para implementar *deep learning*:

Tabla 6. *Códigos de importaciones y definiciones de librerías*

```
import tensorflow as tf
import glob
```

```
import shutl
import io
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, Dropout,
MaxPooling2D
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import layers
from keras.models import Model
from keras.optimizers import Adam
from keras.layers import Dense,GlobalAveragePooling2D
import zipfile
import os
import numpy as np
import matplotlib.pyplot as plt
```

Nota. Presentación de códigos de importaciones y definiciones de librerías necesarias para el desarrollo de redes neuronales. Elaboración propia.

Al momento de subir la base de datos al entorno de ejecución de Google Colab es necesario que las imágenes se encuentren organizadas en una carpeta que a su vez contiene subcarpetas con los nombres respectivos de cada categoría en las que se clasifica la severidad de la mancha anillada, para el caso de estudio son 3 carpetas, alta, media y baja, el *dataset* de entreno y validación se encuentra en Google Drive para tenerlo almacenado permanentemente en la nube, Google Colab se puede enlazar con la cuenta de Google Drive para tener así acceso al *dataset*, posterior a esto se le proporciona una dirección URL previamente autorizada y se debe activar drive desde los archivos del cuaderno de Colab.

Tabla 7. *Código de importación de drive*

```
from google.colab import drive
drive.mount('/content/drive/')
base_dir='/content/drive/My Drive/proyecto_de_grado/Base_de_datos'
```

Nota. Código para obtener la base de datos almacenada en la cuenta de drive. Elaboración propia

Unas líneas de código hacen la respectiva separación de los datos en entrenamiento y validación, distribuyendo los datos respectivamente en una cantidad del 80 y 20% como se muestra a continuación:

Tabla 8. *Código para separar la base de datos de entrenamiento y validación*

```
for cl in classes:
  img_path = os.path.join(base_dir, cl)
  images = glob.glob(img_path + '/*.jpg')
  print("{}: {} Images".format(cl, len(images)))
  num_train = int(round(len(images)*0.8)) #80% de los datos para
entrenar
  train, val = images[:num_train], images[num_train:]
#crear nuevo directorio para entrenamiento 'train'
  for t in train:
    if not os.path.exists(os.path.join(base_dir, 'train', cl)):
      os.makedir(os.path.join(base_dir, 'train', cl))
    shutil.copy(t, os.path.join(base_dir, 'train', cl))
#crear nuevo directorio para validación 'val'
  for v in val:
    if not os.path.exists(os.path.join(base_dir, 'val', cl)):
      os.makedir(os.path.join(base_dir, 'val', cl))
    shutil.copy(v, os.path.join(base_dir, 'val', cl))
```

Nota. Se divide el 80% de datos para entrenar y el 20% para validar. Elaboración propia

Las imágenes de la base de datos están en un formato que no admite Google Colab, además son de diferentes tamaños y muy grandes, por lo cual se debe hacer el preprocesamiento de los datos para que puedan ser trabajados en la red. El formato en el que deben estar las imágenes es JPEG extensión *jpg,* para esta parte se realizó un código en *Matlab* con la función

para procesamiento de imágenes *imwrite* utilizada para escribir los datos de la imagen en un archivo JPEG, PNG o TIFF, según se necesite. El código de matlab utilizado es el siguiente:

Tabla 9. *Código en matlab. .m para la extensión de las imágenes*

```
n= 1164; % Cantidad de fotos
for i=1:n
    I= imread([num2str(i) '.png']); % Leer imagen
    imwrite(I,[int2str(i) '.jpg']); % Escribir el nuevo formato
end
```

Nota. Se cambian en formato png a formato jpg. Elaboración propia

En Colab se realizaron unas líneas de código para cambiar el tamaño de las imágenes a 224 x 224 y escalar cada una de ellas, pues inicialmente cada pixel de la imagen tiene un valor entre 0 y 255, luego de escalarlas, los valores de los píxeles obtienen un valor entre 0 y 1, esto permite más eficiencia en el entrenamiento. Lo anterior se realizó con las siguientes líneas de código:

Tabla 10. *Código de descripción general de imágenes*

```
IMG_SHAPE = 224 #Definir el tamaño de la imagen
batch_size = 32 #Definir el tamaño de los lotes
train_image_generator = ImageDataGenerator(
                  rescale=1./255) #Escalar las imágenes
train_data_gen = train_image_generator.flow_from_directory(
                  batch_size=batch_size,
                  directory=train_dir,
                  shuffle=True,
                  target_size=(IMG_SHAPE,IMG_SHAPE)#Tamaño imágenes
                  class_mode='categorical') #Define varias clases
validation_image_generator = ImageDataGenerator(
                  rescale=1./255) #Escalar las imágenes
val_data_gen = validation_image_generator.flow_from_directory(
```

```
                    batch_size=batch_size,
                    target_size=(IMG_SHAPE,IMG_SHAPE)#Tamaño imágenes
                    class_mode='categorical'
                    )
```

Nota. Códigos para definir tamaño de imágenes y tamaño de lotes, escala de las imágenes y se define qué clase de calcifican por categorías. Elaboración propia

Se estudiaron tres alternativas para realizar el entrenamiento de la de red neuronal capaz de clasificar el grado de severidad de la mancha anillada en un foliolo de frijol, las cuales fueron: primero la creación de una red neuronal convolucional desde cero, segundo aplicar la técnica de *transfer learning* con tres redes conocidas y tercero utilizar el *framework* aXeleRate (Maslov, 2020).

5.2.1. Red neuronal creada

Inicialmente se construyó una red neuronal convolucional desde cero, se utilizaron dos capas de convolución con función de activación ReLu, las ventanas de convolución de tamaño 3x3 y 2x2 respectivamente, seguidas cada una de una capa de *max pooling* de tamaño 2x2. Una capa *flatten* y dos *capas densas*, la última capa densa será la que contenga el número de clases con las que se trabaja y utiliza la función de activación *softmax*.

Tabla 11. *Códigos de descripción de capa de red neuronal convolucionar sencilla.*

```
filtrosConv1 = 32
filtrosConv2 = 64
tamañofiltro1 = (3, 3) # Tamaño capa de convolución 1
tamañofiltro2 = (2, 2) # Tamaño capa de convolución 2
tamañopool = (2, 2) # Tamaño capa de max pooling
clases = 3
modelo = Sequential()
# 2 Capas de convolución seguidas de máxima agrupación
modelo.add(Conv2D(filtrosConv1, tamañofiltro1, padding ="same",
input_shape=(224, 224, 3), activation='relu')),
modelo.add(MaxPooling2D(pool_size=tamañopool))
```

```
# Capa de convolución seguida de máxima agrupación
modelo.add(Conv2D(filtrosConv2, tamañofiltro2, padding ="same"))
modelo.add(MaxPooling2D(pool_size=tamañopool))
# Capas de clasificación
modelo.add(Flatten())
modelo.add(Dense(256, activation='relu'))
modelo.add(Dropout(0.5))
modelo.add(Dense(clases, activation='softmax'))
```

Nota. Descripción dos capas de convolución; una capa de máxima agrupación y una capa de clasificación. Elaboración propia

Después se utiliza categorical_crossentropy como *loss function* y *Adam* como optimizador para entrenar la red, utilizando un *learning rate* de 0.0001. Además, se establece un tamaño de lote de 32 y 10 epochs para entrenar la red:

Tabla 12. *Código para entrenar el modelo de red neuronal*

```
modelo.compile(optimizer='Adam',loss='categorical_crossentropy',metr
ics=['accuracy'])
step_size_train=train_data_gen//train_data_gen.batch_size
history = modelo.fit(train_data_gen,
                     steps_per_epoch=step_size_train,
                     validation_data = val_data_gen,
                     epochs=10
                     )
```

Nota. Códigos para ejecutar todos los valores establecidos. Elaboración propia

Una de las técnicas que ayudan a mejorar la precisión y previene el *overfitting* es *data augmentation* la cual, mediante la herramienta *ImageDataGenerator,* se aplicaron las transformaciones de rotación, desplazamientos, espejo horizontal y zoom a las imágenes de entrenamiento de nuestra base de datos:

Tabla 13. *Código Data augmentation*

```
train_image_generator = ImageDataGenerator(
                    rescale=1./255, #Escalar
                    rotation_range=45, #Rotar
                    width_shift_range=.15, #Traslación Ancho
                    height_shift_range=.15, #Traslación Alto
                    horizontal_flip=True, #Espejo
                    zoom_range=0.1 #Acercar)
```

Nota. Código para ejecutar técnica de transformación para aumentar la base de datos. Elaboración propia.

Al haber utilizado la arquitectura de red neuronal creada presentada en la (Figura 20), no se observaron los resultados buscados, pues se obtuvo una precisión alrededor del 63% y un modelo con aproximadamente 51 millones de parámetros, lo cual lo hace pesado para implementar en un sistema embebido.

Figura 20. *Resumen de la arquitectura de la red creada*

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 224, 224, 32)      896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 112, 112, 32)      0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 112, 112, 64)      8256
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 56, 56, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 200704)            0
_________________________________________________________________
dense (Dense)                (None, 256)               51380480
_________________________________________________________________
dropout (Dropout)            (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 771
=================================================================
Total params: 51,390,403
Trainable params: 51,390,403
Non-trainable params: 0
```

Nota. Resumen de datos de códigos de descripción de capa de red neuronal convolucionales sencilla. Elaboración propia

Usar redes neuronales previamente entrenadas, ya sea en su totalidad o de forma parcial, mediante la técnica de *transfer learning*, reduce la cantidad de parámetros, el tiempo de entrenamiento y mejora la precisión de un modelo.

5.2.2. Transfer learning con MobileNetV2

Se utilizó la técnica de *transfer learning* haciendo adaptaciones y entrenando con nuestra base de datos. Se probaron diferentes topologías de redes neuronales convolucionales; *ResNet50, MobileNet y MobileNetV2*, la red que arrojó mejores resultados fue la MobilenetV2. A continuación se muestran las líneas de código que permite utilizar de manera parcial el modelo y se define la capa de clasificación.

Tabla 14. *Código importar y usar el modelo MobileNetV2*

```
import keras

from keras.applications import MobileNetV2 #Importar el modelo
Mobile = MobileNetV2(weights = 'imagenet', include_top = False,
input_shape = (224,224,3)) # Se carga el modelo entrenado con la
base de datos de ImageNet, no se utiliza la capa de clasificación y
se declara el tamaño de la imagen de entrada

for layer in Mobile.layers:

  layer.trainable = False #Apagar las capas entrenadas

x=Mobile.output #Salida del modelo

x=GlobalAveragePooling2D()(x) #Se agrega average pooling

preds=Dense(3,activation='softmax')(x) #Se agrega la capa de
clasificación

model = Model(inputs = Mobile.input, outputs = preds)

model.compile(optimizer='Adam',loss='categorical_crossentropy',metri
cs=['accuracy'])
```

Nota. Importación de modelo y se define qué capas se van utilizar. Elaboración propia

Antes de iniciar el entrenamiento se indica que utilice nuestra base de datos ya procesada con las siguientes líneas de código:

Tabla 15. *Código para utilizar la base de datos en drive*

```
history = model.fit(train_data_gen #Datos de entrenamiento
                    steps_per_epoch=step_size_train,
                    validation_data = val_data_gen, #Datos de
validación
                    epochs=10)
```

Nota. Códigos utilizados para acceder a la base de datos en drive. Elaboración propia.

Una vez entrenado el modelo con MobileNetV2, se logró aumentar la precisión de validación a un 77% y como se muestra en la (Figura 21) se reduce la cantidad de parámetros a aproximadamente 2 millones, lo cual hace al modelo más viable para implementarlo en un sistema embebido.

Figura 21. *Resumen de los parámetros de red con transfer learning MobileNetV2*

```
Total params: 2,261,827
Trainable params: 3,843
Non-trainable params: 2,257,984
```

Nota. Resultados arrojados luego de ejecutar los códigos. Elaboración propia

5.2.3. Framework aXeleRate

Se usó el *framework* aXeleRate (Maslov, 2020), que es una librería disponible de Python para entrenar una red neuronal conocida y obtener el modelo en formatos .h5 para ejecutar en *Google Colab*, .kmodel compatible con MaixPy y .tflite compatible con Raspberry Pi. El código que se utilizó fue modificado de una libreta de prueba de aXeleRate planteada por Maslov (2020) usada para clasificar cinco clases de animales, las modificaciones se encuentran principalmente en la instalación de las librerías con sus respectivas versiones, como se muestra a continuación:

Tabla 16. *Importación e instalación de librerías*

```
%tensorflow_version 1.x
!pip install axelerate==0.6.0
!pip install tensorflow==1.15.0
!pip install keras==2.3.1
```

```
import sys
sys.path.append('/content/aXeleRate')
from axelerate import setup_training,setup_inference
```

Nota. Se instala de aXeleRate, tensorflow y keras. Elaboración propia

En las siguientes líneas de código se ejecutan para mostrar imágenes del *dataset* y se definen las tres categorías; alto, bajo y medio.

Tabla 17. *Código para graficar imágenes existentes en la base de datos*

```
%matplotlib inline
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import glob
def show_image(filename):
   print(filename)
   image = mpimg.imread(filename)
   plt.figure()
   plt.imshow(image)
   plt.show()
alto = os.path.join('/content/drive/MyDrive/PROYECTO DE
GRADO/Base_de_datos/alto')
bajo = os.path.join('/content/drive/MyDrive/PROYECTO DE
GRADO/Base_de_datos/bajo')
medio = os.path.join('/content/drive/MyDrive/PROYECTO DE
GRADO/Base_de_datos/medio')
print('total training alto images:', len(os.listdir(alto)))
print('total training bajo images:', len(os.listdir(bajo)))
print('total training medio images:', len(os.listdir(medio)))
alto_files = os.listdir(alto)
print(alto_files[:10])
bajo_files = os.listdir(bajo)
print(bajo_files[:10])
medio_files = os.listdir(medio)
print(medio_files[:10])
pic_index = 2
next_alto = [os.path.join(alto, fname)
               for fname in alto_files[pic_index-2:pic_index]]
next_bajo = [os.path.join(bajo, fname)
               for fname in bajo_files[pic_index-2:pic_index]]
```

```
next_medio = [os.path.join(medio, fname)
                for fname in medio_files[pic_index-2:pic_index]]
for i, img_path in enumerate(next_alto+next_bajo+next_medio):
 show_image(img_path)
```

Nota. Código para graficar 6 imágenes por categoría. Elaboración propia

A continuación se configura la red conocida que se va a implementar, en el caso de estudio se implementó la MobileNet, también se establecen parámetros tales como número de épocas, *batch size* entre otros, y finalmente se convierte al formato .kmodel o .tflite.

Tabla 18. *Código de configuración del modelo usando aXeleRate*

```
config = {
       "model" : {
           "type":                 "Classifier",
           "architecture":         "MobileNet7_5", # Red a utilizar
           "input_size":           224, # Tamaño de entrada
           "fully-connected":      [100,50],
           "labels":               [],
           "dropout" :     0.5
       },
       "weights" : {
           "full":             "",
           "backend":            "imagenet",
           "save_bottleneck":      False
       },
       "train" : {
           "actual_epoch":         10,
           "train_image_folder":   "/content/drive/MyDrive/PROYECTO
DE GRADO/Base_de_datos/train", #Base de entrenamiento
           "train_times":          2,
           "valid_image_folder":   "/content/drive/MyDrive/PROYECTO
DE GRADO/Base_de_datos/val", #Base de validación
           "valid_times":          2,
           "valid_metric":         "val_accuracy",
           "batch_size":           32,
           "learning_rate":        1e-3,
           "saved_folder":       "classifier",
           "first_trainable_layer": "",
```

```
        "augumentation":        True
    },
    "converter" : {
        "type":             ["k210"]}}
```

Nota. Se utiliza una red MoviNed para el entrenamiento. Elaboración propia

Finalmente se realiza el entrenamiento con la función *setup_training* y usando la configuración definida anteriormente:

Tabla 19. *Código para entrenar*

```
model_path = setup_training(config_dict=config)
```

Nota. Se realiza el entrenamiento. Elaboración propia

Una vez ejecutado los códigos se verifica el rendimiento real haciendo inferencia con el *dataset* de validación, este modelo alcanzó un 91% de precisión, menos de 2 millones de parámetros como se muestra en (Figura 22) y pesos de 7.6 MB, 2.2 MB en formatos .tflite y .kmodel respectivamente.

Figura 22. *Resumen de los parámetros de red con aXeleRate*

```
Total params: 1,915,079
Trainable params: 1,898,663
Non-trainable params: 16,416
```

Nota. Resumen de los códigos ejecutados y entrenados Elaboración propia

5.3. Prueba de la red neuronal en Google Colab

Para realizar estas pruebas se utilizan los archivos .h5 de los modelos previamente entrenados y se reservaron un conjunto fotos, o sea un *dataset* exclusivo para *te*st. Primero se realizan las debidas importaciones de las librerías; en las siguientes líneas de código se presentan importaciones adicionales a las que ya se presentaron anteriormente.

Tabla 20. *Código para probar un modelo previamente entrenado*

```
from tensorflow.python.keras import layers
```

```
from keras.preprocessing.image import load_img, img_to_array
from keras.models import load_model
# Importar archivos desde el computador
from google.colab import files
uploaded= files.upload()
# Enlazar cuenta de Drive
from google.colab import drive
drive.mount('/content/drive/', force_remount=True)
# Direccionar el modelo de la red ya entrenada .h5
model_dir = '/content/model(1).h5'
modelo = load_model(modelo_dir) # Cargar el modelo
# Direccionar el conjunto de fotos para la prueba
test_dir='/content/drive/My Drive/proyecto de grado/TEST'
classes = ['alto', 'bajo', 'medio'] # Definir clases
```

Nota. Importación de librerías, se importa el modelo ya entrenado y se utiliza los datos de drive**.** Elaboración propia

Se realizó el procesamiento de las imágenes, tal cual como ya se mostró; definiendo el tamaño de las imágenes, el tamaño del lote (para el caso de estudio se definió en 32), escalando las imágenes y las clases para este caso '*sparse*'. Seguidamente se realizan las predicciones y la matriz de confusión para tener las pruebas de la clasificación de la mancha anillada en un foliolo de frijol mediante una la red neuronal.

Tabla 21. *Código que ejecuta la prueba*

```
#  Se extraen los datos y las etiquetas del conjunto de imágenes ya
procesadas
data_test, label_test = test_data_gen[0]
classes_names = np.array(classes) # Se define como arreglo las
clases
# Para realizar la predicción con los datos y las etiquetas
prediccion = cnn.predict(data_test)
prediccion = tf.squeeze(prediccion).numpy()
```

```
prediccion_id = np.argmax(prediccion, axis=-1)

predicted_class_names = classes_names[prediccion_id] #
Emparejamiento

predicted_class_names # Imprimir las etiquetas de predicción

# Importaciones para realiza la matriz de confusión

from mlxtend.plotting import plot_confusion_matrix

from sklearn.metrics import confusion_matrix

from sklearn.metrics import accuracy_score

mat= confusion_matrix(label_test, prediccion_id) #Llenado de la
matriz

fig, ax= plot_confusion_matrix(conf_mat= mat, figsize=(10, 10),
show_normed=True) # Imprimir la imagen de la matriz
```

Nota. Los resultados en la prueba arrojan una matriz de confusión y una imagen con las predicciones**.** Elaboración propia.

5.4. Ejecución de la red en la Raspberry Pi 2

Para ejecutar la red neuronal ya entrenada en la Raspberry se necesita utilizar TensorFlow, para esto se creó una carpeta a la cual se le realizó un entorno virtual de ejecución y dentro de la misma carpeta se hizo la instalación de las librerías tensorflow. Para lo anterior se utilizaron los siguientes códigos en la terminal.

Tabla 22. *Código para definir entornos virtuales*

```
#Se instala virtualenv para crear entornos virtuales

sudo pip3 install virtualenv

#Se crea el entorno virtual en la carpeta donde se va a trabajar

python3 -m venv tflite1-env

#Para activar el entorno virtual cada vez que se vaya a trabajar

source tflite1-env/bin/activate
```

Nota. Se establecen los códigos del entorno virtual. Elaboración propia

Para ejecutar el modelo de la red neuronal ya entrenada, es necesario hacer la conversión del archivo de .h5 (extensión keras) a .tflite, cuya extensión es compatible con la Raspberry Pi, para dicha conversión se utilizan estas líneas de código en un cuaderno de colab.

Tabla 23. *Código para convertir el modelo. tflite*

```
#Cargar el modelo

model = tf.keras.models.load_model('/content/modelo(1).h5')

converter = tf.lite.TFLiteConverter.from_keras_model(model)

tflite_model = converter.convert() #Convertir el modelo

 #Guardar el modelo convertido a tflite

open("/content/converted_model.tflite", "wb").write(tflite_model)
```

Nota. Se cambia extensión .h5 a. tflite. Elaboración propia

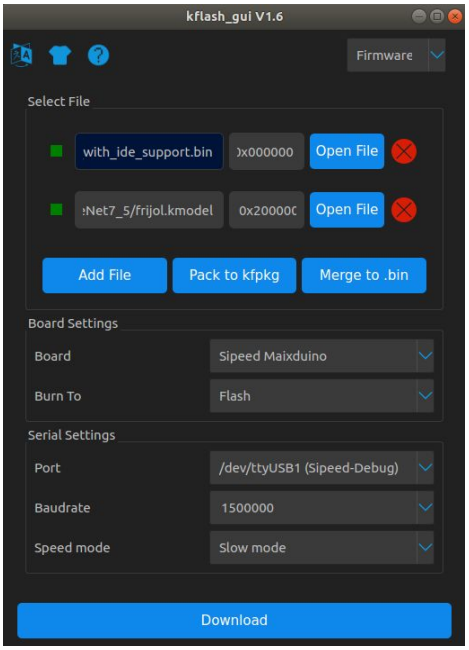
Para ejecutar el modelo, se tomó como base el código que utiliza la guía de TensorFlow para ejecutar archivos tflite en Raspberry (Reyes,2021), se hizo el debido ajuste del código para hacer la debida clasificación de las categorías de la mancha anillada. El código utilizado en la Raspberry se muestra en el Apéndice A ó en el repositorio de *GitHub*.

5.5. Ejecución de la red en la Sipeed Maixduino

Para hacer uso de la tarjeta se instaló la IDE de MaixPy, que como se mencionó anteriormente es donde se realiza el script para interactuar con el hardware, también se instaló Kflash, lo cual es una herramienta para flashear el dispositivo; es decir subir el *firmware* y los modelos ya entrenados de redes neuronales en formato. kmodel, como se muestra en la (Figura 23). El proceso de instalación de estos programas se encuentra en el sitio web oficial de MaixPy Sipeed (Sipeed, 2018).

Para implementar el modelo de red para clasificar la mancha anillada en foliolos de frijol entrenada y programar la tarjeta Sipeed Maixduino, en MaixPy IDE se ejecuta el código mostrado en el Apéndice B ó en el repositorio de *GitHub* (Nuñez, 2021).

Figura 23. *Flasheo de la tarjeta Maixduino*



Nota. Instalación de archivos en la memoria Flash, se carga firmware y el modelo de red neuronal.

6. Resultados

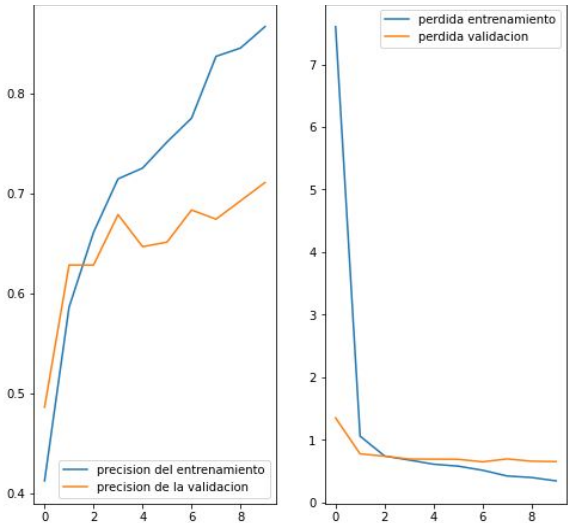
A continuación, se muestran los resultados de los entrenamientos de las diferentes arquitecturas de redes que fueron utilizadas para clasificar la mancha anillada en foliolos de frijol. Se realizaron las pruebas de clasificación con el *dataset* de la base de datos dirigida para *test,* que como se muestra en la (Tabla 5) cuenta con 73 imágenes, se definió un lote de 32 imágenes por prueba. Se presentan las gráficas de pérdida y precisión a lo largo de las épocas, con el fin de exponer mejor los resultados se realizó una figura con los foliolos evaluados y su

respectiva clasificación hecha por las redes; sí la etiqueta es de color azul, indica que la clasificación es correcta, sí es de color rojo, indica que está mal clasificada.

También se muestra una matriz de confusión que permite revisar cómo fueron clasificados los foliolos; en la diagonal principal quedarán la cantidad de aciertos y en el resto de la matriz se muestra la categoría con la que se confundieron los resultados erróneos.

6.1.Red neuronal creada

Figura 24. *Curvas de precisión y pérdida de entrenamiento y validación*



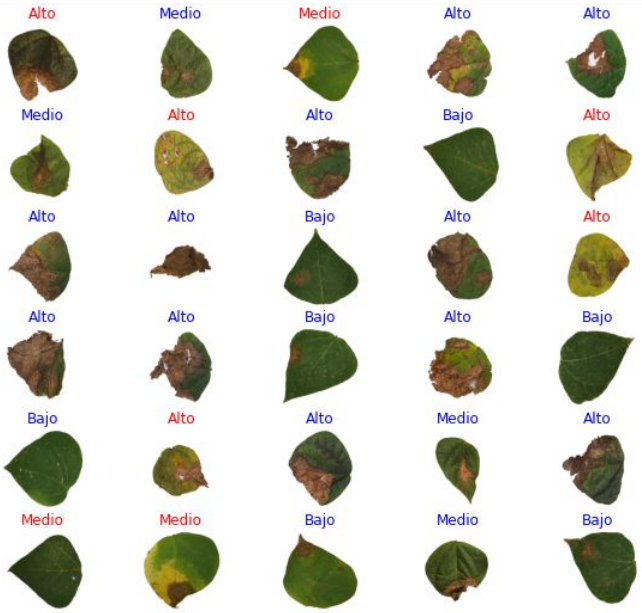
Nota. Curvas de precisión y pérdidas de entrenamiento y validación de la red creada**.** Elaboración propia

Se logró una precisión en la validación del 63,8% como se ve en la (Figura 24), con la red convolucional creada desde cero, esta red tiene más de 51 millones de parámetros y en formato .h5 tiene un peso de 588.16Mbytes.

En la (Figura 25) se muestra las predicciones de 30 imágenes con sus respectivas etiquetas y color según haya sido su clasificación. En la (Figura 26) se complementa el resultado con la matriz de confusión, el *dataset* para esta prueba fue de 11 foliolos en alto de los cuales todos quedaron bien clasificados, 12 foliolos en bajo; 8 clasificaciones correctas y 4 se

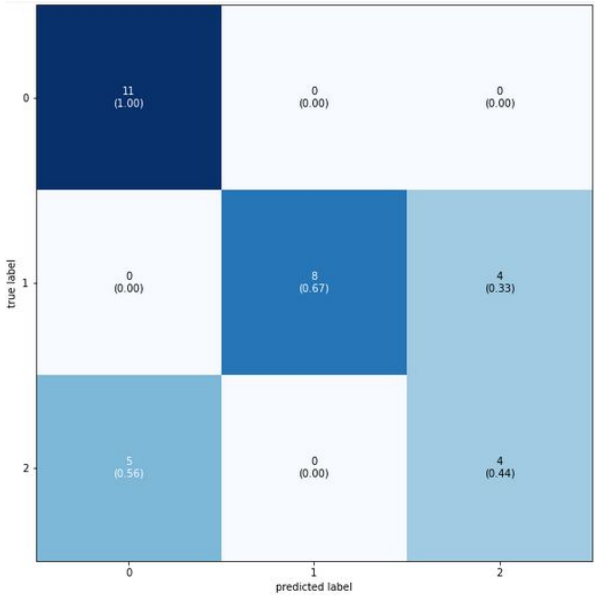
confundieron con la categoría media, y 9 foliolos en medio; 4 clasificaciones correctas, pero 5 se confundieron con nivel alto. Estos resultados empezaron a indicar que la categoría media tendría dificultades para las clasificaciones.

Figura 25. *Predicción de la red creada*



Nota. Imágenes de hoja de frijol de la red creada. Elaboración propia.

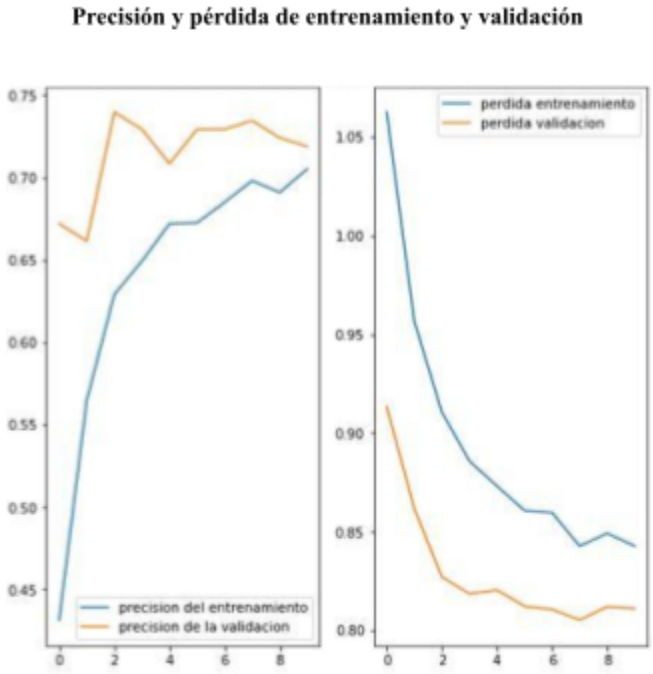
Figura 26. *Matriz de confusión de la clasificación con la red neuronal creada*



Nota. Clasificación realizada con la red neuronal. Elaboración propia.

6.2.Red con transfer learning ResNet50

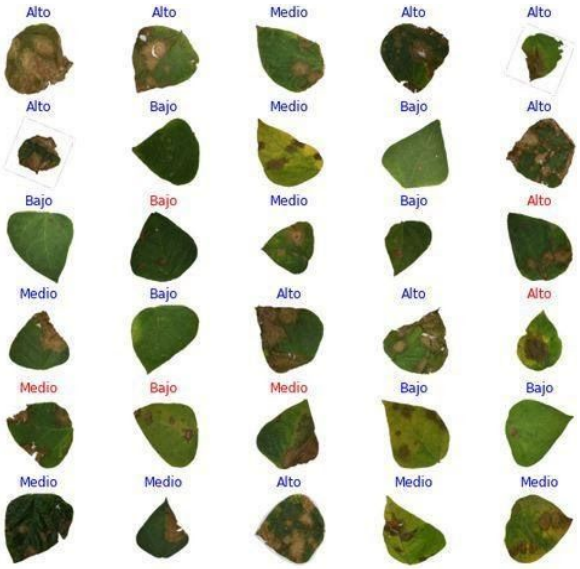
Figura 27. *Curvas de precisión y pérdidas de entrenamiento y validación ResNet50*



Nota. Representación gráfica de curvas de precisión, pérdidas de entrenamiento y validación ResNet50. Elaboración propia

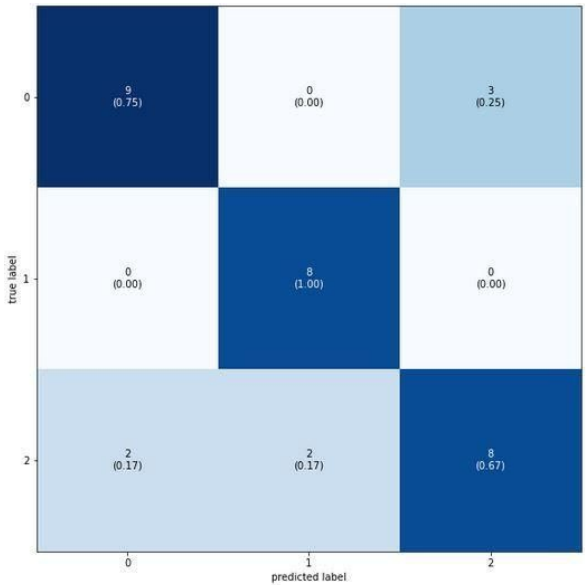
En la (Figura 27) se ve en comportamiento del entrenamiento que tuvo haciendo *transfer learning* con la arquitectura de red ResNet50, con esta red se tuvo una precisión en validación de 71.9%, un resultado mejor en comparación con la red creada, la cantidad de parámetros luego del *finetune* con la capa de clasificación no aumentó significativamente con respecto a la arquitectura propia de la red. Las curvas de precisión muestran una mejora en el entrenamiento a medida que pasan las épocas, aunque la precisión en la validación (curva de color naranja) se mantuvo sobre el 65% desde las primeras épocas.

Figura 28. *Predicciones de la red ResNet50*



Nota. Imágenes arrojadas por la red ResNet50 de la hoja de frijol. Elaboración propia.

Figura 29. *Matriz de confusión de la clasificación con la red ResNet50*



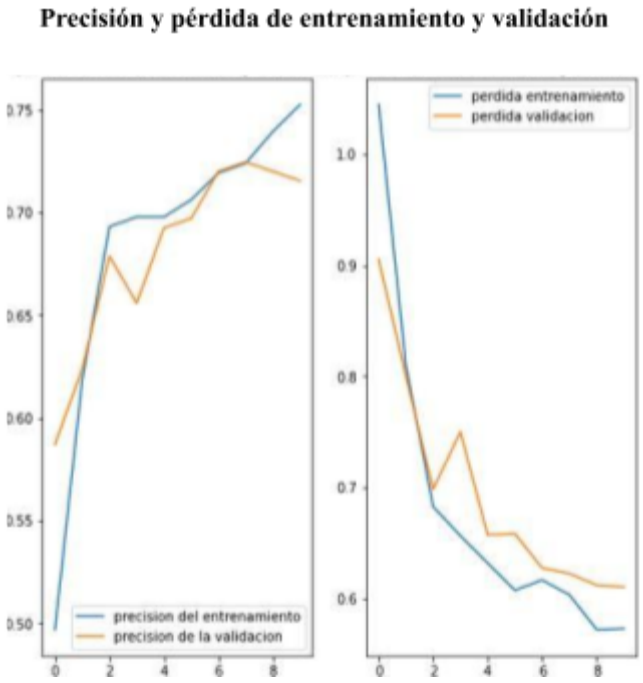
Nota. Clasificación realizada con la red ResNet50. Elaboración propia

En la (Figura 28 y 29) se muestra el resultado de la red ResNet50, el *dataset* para la prueba de esta red fue de 12 foliolos en alto; 9 clasificados correctamente y 3 confundidos con la categoría media, 8 foliolos en bajo; todos bien clasificados y 12 foliolos en media; 8 clasificaciones correctas, 2 y 2 confundidos con las otras dos categorías.

6.3.Red con transfer learning MobileNet

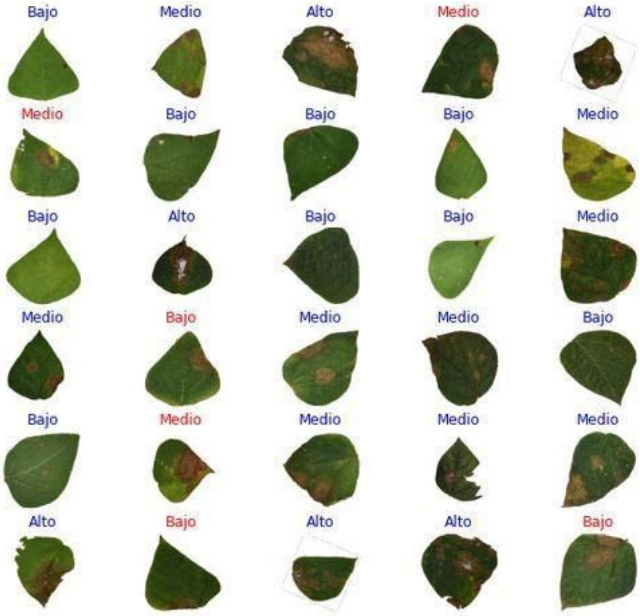
Haciendo *transfer learning* con la red MobileNet se obtuvo una precisión en validación del 71.6%, con resultados muy similares a los obtenidos con la ResNet50, las curvas entre el entrenamiento y la validación, tanto en precisión como en pérdidas son similares y parejas a lo largo de todo el entrenamiento como se ve en la Figura 30. Este comportamiento en las curvas de pérdidas permite verificar que la red neuronal no presente *overfitting*, por tanto, la red está memorizando los datos de entrenamiento.

Figura 30. *Curvas de precisión y pérdidas de entrenamiento y validación MobileNet*



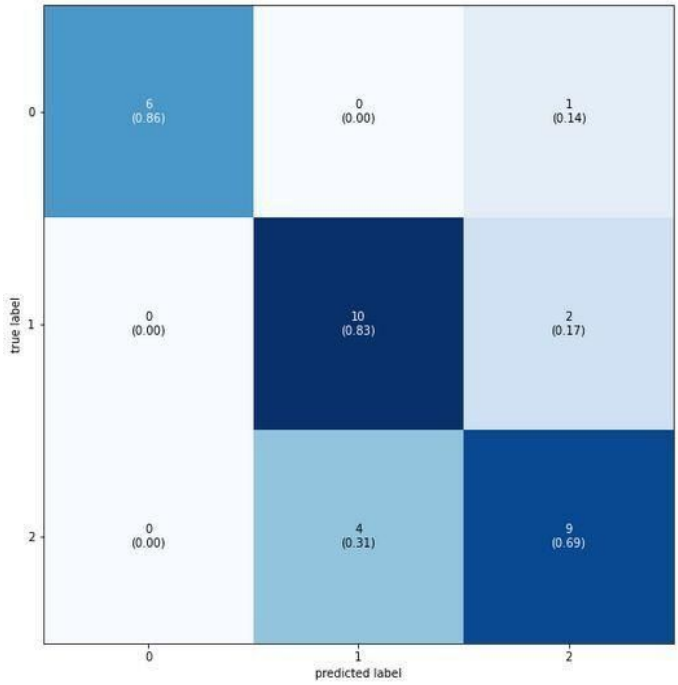
Nota. Representación gráfica curvas de precisión, pérdida de entrenamiento y validación MobileNetV2. Elaboración propia.

Figura 31. *Predicciones con la red MobileNetV2*



Nota. Imágenes de predicciones con la red MobileNetV2. Elaboración propia

Figura 32. *Matriz de confusión de la clasificación con la red MobileNet*

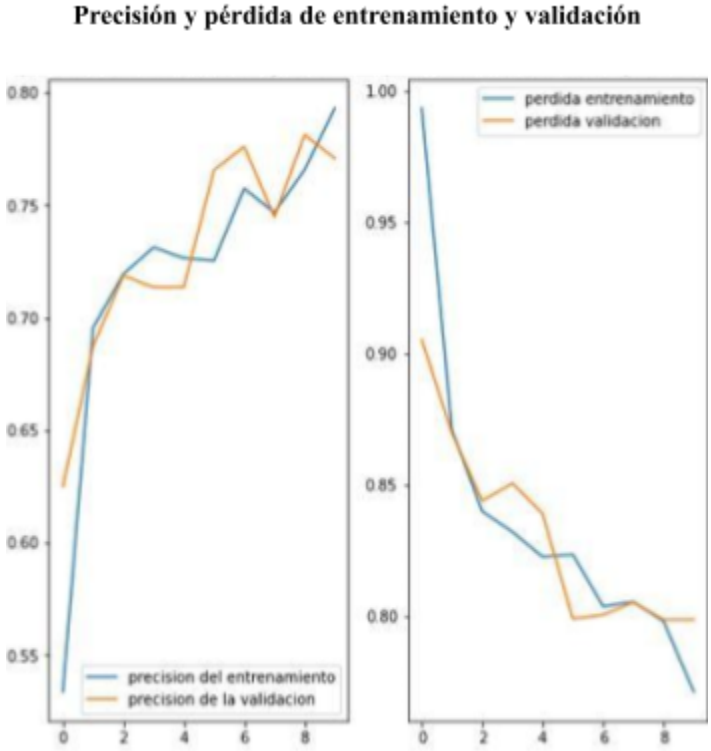


Nota. Clasificación con la red MobileNet. Elaboración propia

Los resultados de la red MobileNet se muestran en las (Figuras 31 y 32), el *dataset* utilizado para probar esta red se distribuyó de la siguiente forma: 7 foliolos en alto; 6 bien clasificados y 1 confundido con la categoría media, 12 foliolos en baja; 10 clasificados correctamente y 2 confundidos en media, y 13 foliolos en media; 9 clasificaciones correctas y 4 confundidos en categoría baja.

6.4.Red con transfer learning MobiliNetV2

Figura 33. *Curvas de precisión y pérdidas de entrenamiento y validación MobileNetV2*



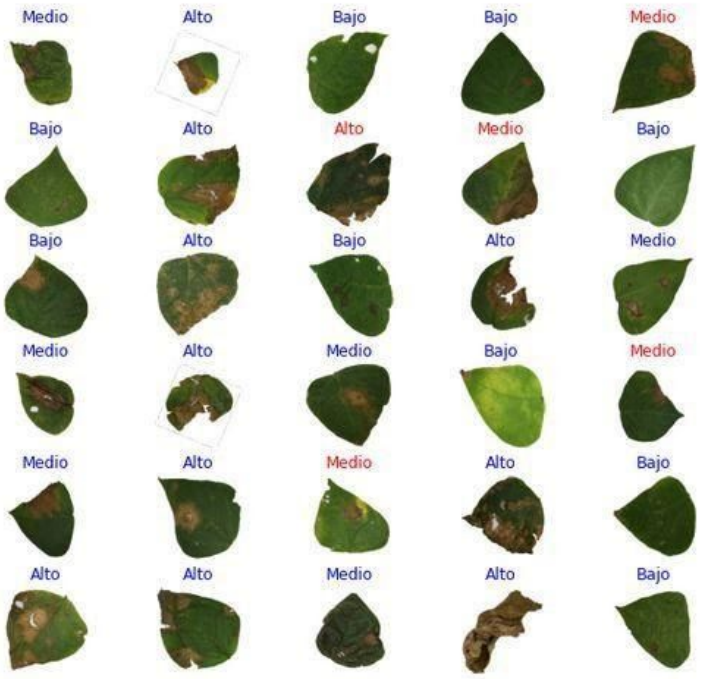
Nota. Representación gráfica curvas de precisión, pérdida de entrenamiento y validación MobileNetV2. Elaboración propia

Haciendo *transfer learning* con la red MobileNetV2 se alcanzó una precisión en validación del 77% con un peso de 8.67 MB en formato. tflite, siendo la mejor entre los anteriores resultados para implementar en la Raspberry Pi ya que para implementarla en la

Sipeed Maixduino hay limitación de memoria pues este modelo en .kmodel pesa 7.25 MB. Las curvas de la (Figura 33) presentan similitud con las resultantes de la red MobileNet.

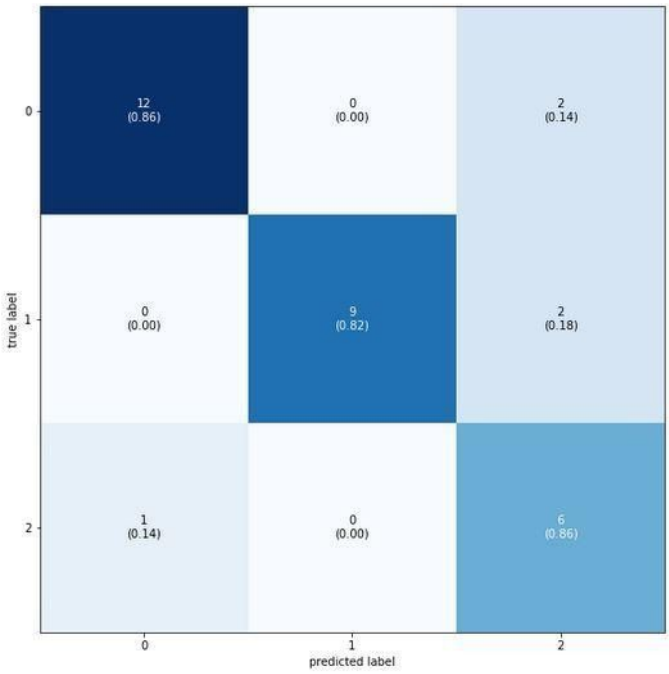
En la (Figura 34) se muestra las predicciones de 30 imágenes con sus respectivas etiquetas y color; 25 foliolos en azul y solo 5 en rojo lo que indica un 83% en aciertos. En la Figura 35 se presenta la matriz de confusión donde el *dataset* para esta prueba fue de 14 foliolos en alto; 12 quedaron bien clasificados y 2 se confundieron con categoría media, 11 foliolos en bajo; 9 clasificaciones correctas y 2 se confundieron con la categoría media, y 7 foliolos en medio; 6 clasificaciones correctas y solo 1 se confundió con nivel alto.

Figura 34. *Predicciones con la red MobileNetV2*



Nota. Imágenes de la hoja de frijol con la red MobileNetV2. Elaboración propia

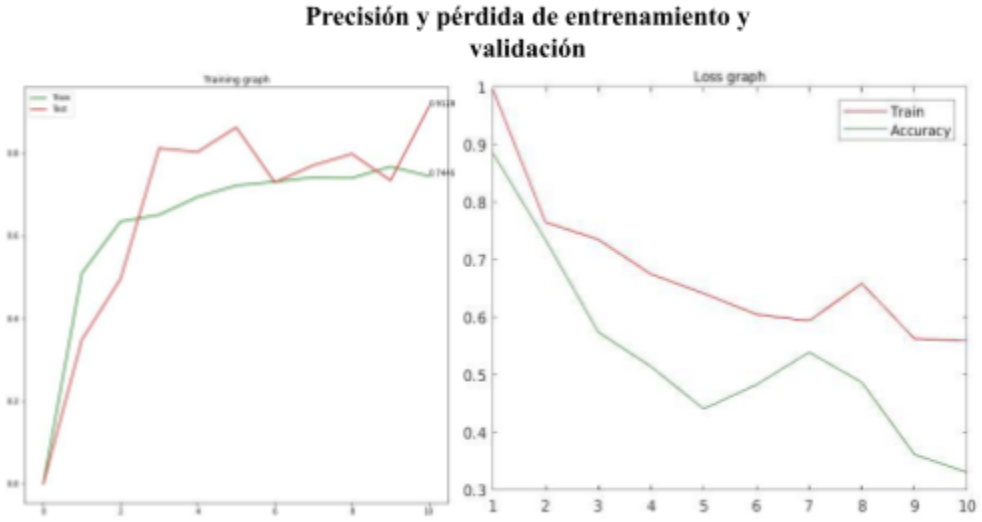
Figura 35. *Matriz de confusión de la clasificación con la red MobileNetV2*



Nota. Clasificación con la red MobileNetV2. Elaboración propia

6.5.Red con el framework aXeleRate

Figura 36. *Curvas de precisión y pérdidas de entrenamiento y validación de la red usando aXeleRate*

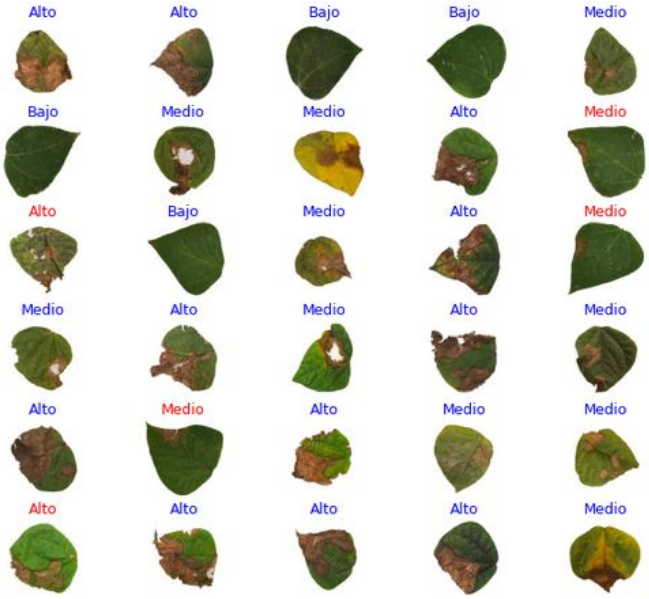


Nota. Representación gráfica de precisión y pérdida de entrenamiento y validación de la red aXeleRate. Elaboración propia

Usar el *framework* de aXeleRate con la arquitectura MobileNet, permitió elevar la precisión en validación a un 91%, con pesos de 7.6 MB. tflite y 2.2 MB .kmodel, lo cual lo hace el mejor modelo entre los anteriores presentados y funcional para implementar en los dos sistemas embebidos. En las curvas de la (Figura 36) se observan las curvas de precisión y pérdidas tanto en entrenamiento como en validación, se puede observar que el valor de pérdida es de 0.3309 que es el menor valor logrado en los entrenamientos.

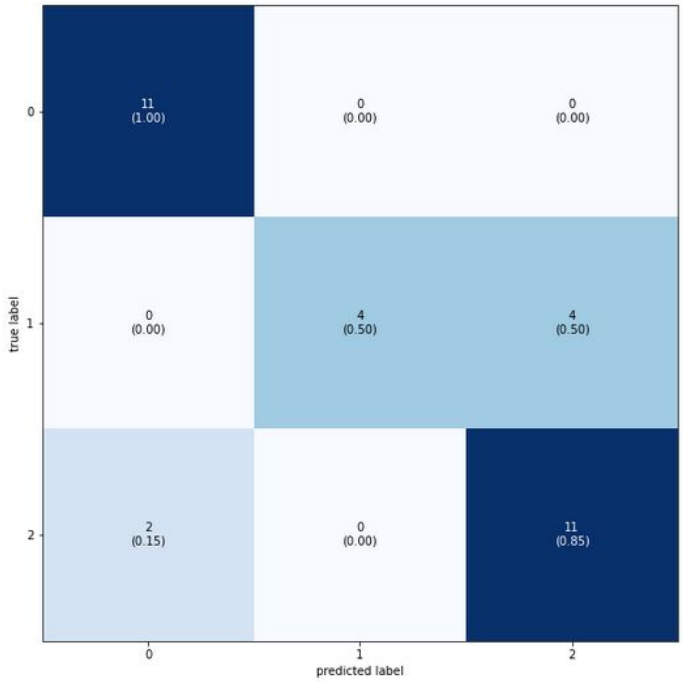
En la (Figura 37 y 38) se muestra el resultado de la prueba con la red de aXeleRate, el *dataset* que se clasificó fue de 11 foliolos en alto; todos bien clasificados, 8 foliolos en bajo; con 4 aciertos y 4 erróneos en categoría media y 13 foliolo para la categoría media; 11 de ellos clasificados correctamente y 2 confundidos en categoría alta.

Figura 37. *Predicciones de la red usando aXeleRate*



Nota. Imágenes de predicciones de la red con aXeleRate. Elaboración propia.

Figura 38. *Matriz de confusión de la clasificación con la red usando aXeleRate*



Nota. Clasificación con la red aXeleRate. Elaboración propia.

En la (Tabla 6) se exponen los resultados anteriormente descritos, con esta tabla determinamos que el modelo de red que clasifica mejor es la que se desarrolló usando el *framework aXeleRate*, esta fue la que se implementó en los dos sistemas embebidos.

Tabla 24. *Resultados de las redes neuronales utilizadas para la clasificación de la mancha anillada en foliolo de frijol en el entorno Google Colab*

	Entrenamiento		**Validación**		**Prueba**	
RED	**Pérdida**	**Precisión**	**Pérdida**	**Precisión**	**Aciertos**	**Desaciertos**
Red creada	0.8056	0.6040	0.7541	0.6376	23	9
ResNet50	0.8427	0.7051	0.8111	0.7188	25	7
MobileNet	0.573	0.7527	0.6104	0.7156	25	7

MobileNetV2	0.7713	0.7931	0.7987	0.7708	27	5
aXeleRate	0.5594	0.77446	0.3309	0.9128	26	6

Nota. Elaboración propia

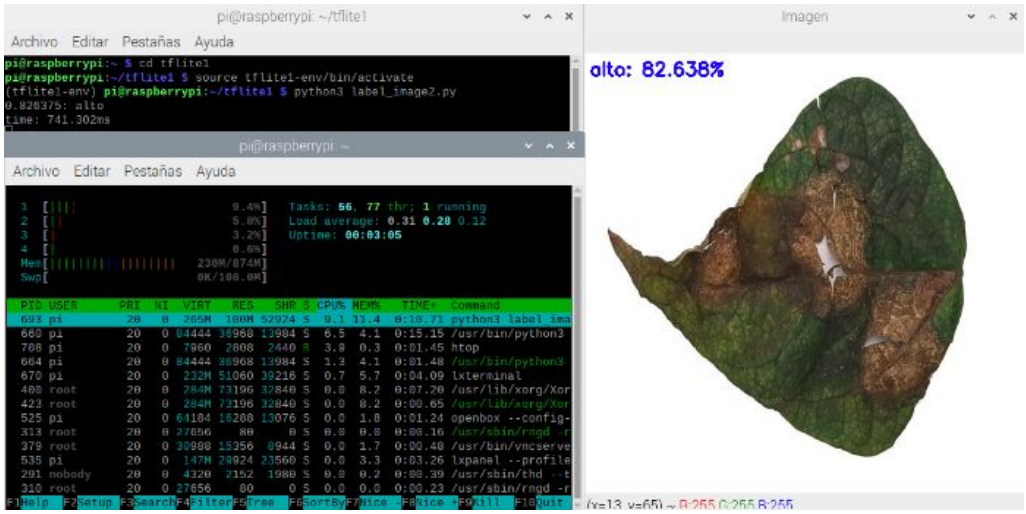
Estos resultados fueron obtenidos desde la ejecución en el entorno de *Google Colaboratory*, el tiempo promedio de ejecución de la clasificación de un foliolo fue de 3.55ms.

6.6.Resultados en la Raspberry Pi 2

Se implementó el modelo de red MobileNetV2 y el modelo con aXeleRate, puesto que fueron los que tuvieron un porcentaje mayor en la precisión de la clasificación de la mancha anillada en foliolos de frijol y su peso es soportado en este *hardware*. Para las pruebas en la Raspberry se usó el *dataset* utilizado para *test*, las imágenes que se evaluaron fueron adquiridas directamente desde la carpeta en la Raspberry, ya que en el momento de la implementación no se contó con hojas de frijol en físico. Con el entorno virtual creado se muestran los resultados; el foliolo evaluado, la etiqueta y porcentaje en el que fue clasificado, el tiempo que le tomó a la Raspberry la ejecución y se extrae más información para saber el porcentaje de memoria que se consume.

Se obtuvo un tiempo promedio de ejecución de 771.98ms la clasificación de un foliolo, un promedio de 11.47% de memoria utilizada y un consumo de 8.43% en CPU. Estos valores fueron calculados con los resultados de la clasificación de cada una de las categorías, como se muestran en las (Figuras 39, 40 y 41).

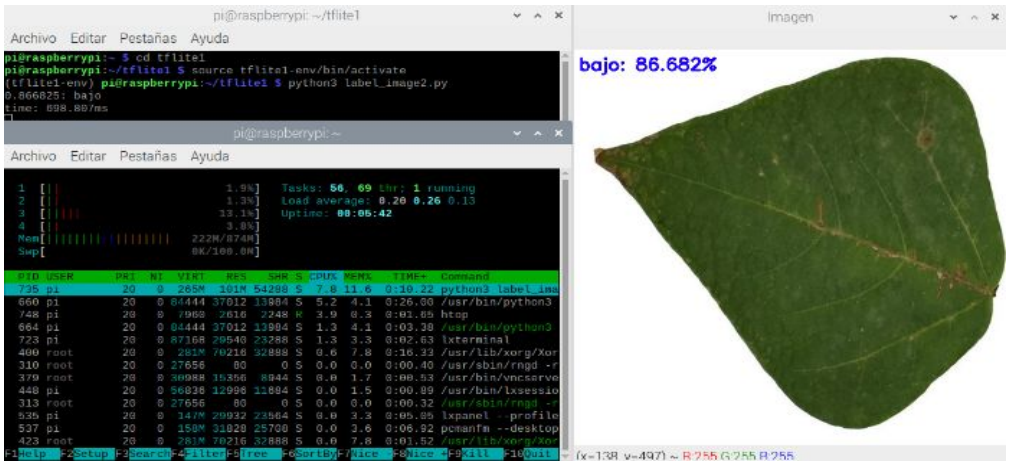
Figura 39. *Clasificación de la categoría alta en la Raspberry Pi*



Nota. Evidencia de clasificación de la categoría alta con la red Raspberry Pi.

Elaboración propia

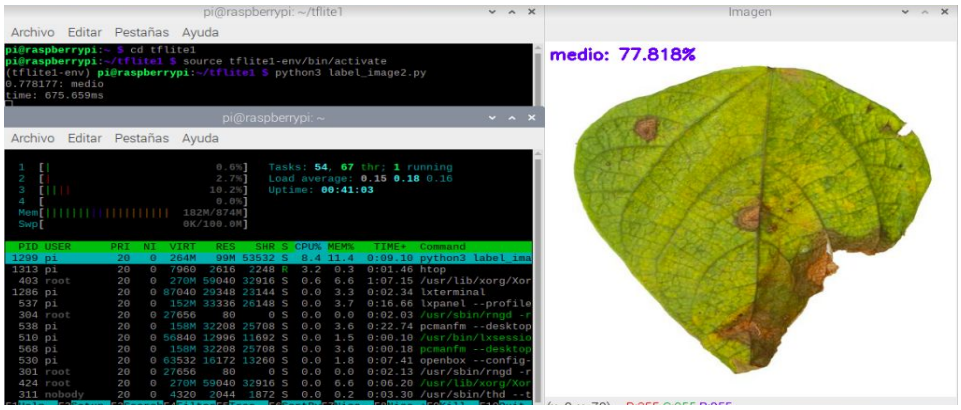
Figura 40. *Clasificación de la categoría baja en la Raspberry Pi*



Nota. Evidencia de clasificación de la categoría baja con la red Raspberry Pi.

Elaboración propia

Figura 41. *Clasificación de la categoría media en la Raspberry Pi*



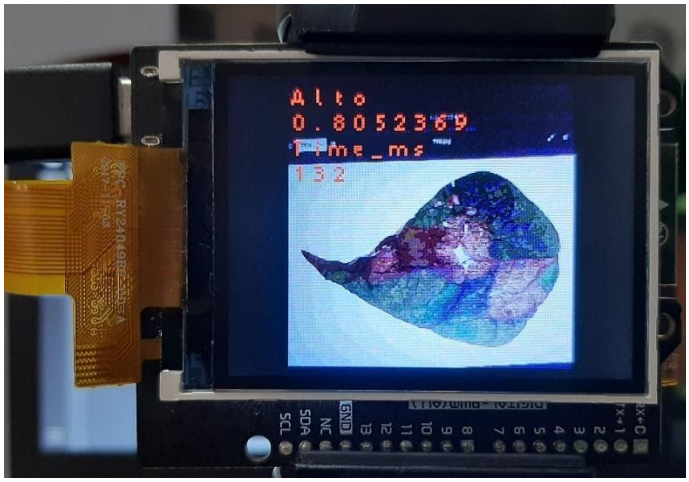
Nota. Evidencia de clasificación de la categoría media con la red Raspberry Pi. Elaboración propia

6.7.Resultados en Sipeed Maixduino

Se implementó el modelo de red obtenido mediante aXeleRate, este modelo presenta la mejor precisión y un peso viable para trabajar en el sistema embebido Maixduino y al igual que la Raspberry Pi, la prueba se llevó a cabo con el *dataset* de *test*. La manera en la que se captura la imagen a evaluar es a través de la cámara del kit de Sipeed (Figura 17), aunque en el momento en que se llevaron a cabo las pruebas en los sistemas embebidos no se tenían hojas en físico se usó la proyección de los archivos en la pantalla del computador.

Se obtuvo un tiempo promedio de ejecución de 134.37ms en la clasificación por foliolo, este valor fue calculado con los resultados de las clasificaciones de cada categoría como se muestran en las (Figuras 42, 43 y 44).

Figura 42.*Clasificación de la categoría alta en Sipeed Maixduino*

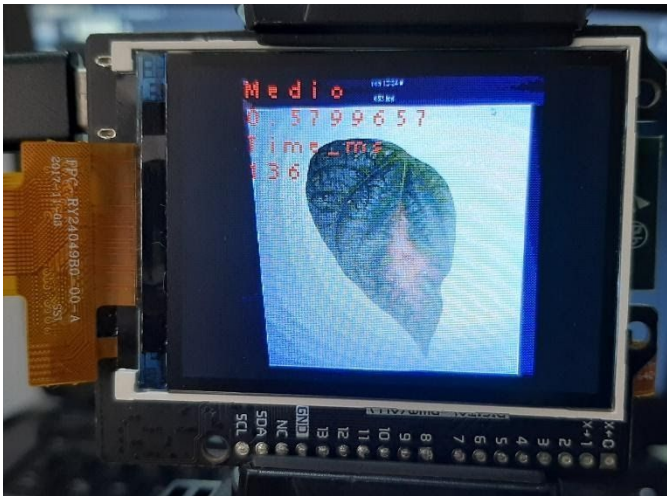


Nota. Imagen de clasificación de la categoría alta en Sipeed Maixduino Elaboración propia

Figura 43. *Clasificación de la categoría baja en Sipeed Maixduino*

Nota. Imagen de clasificación de la categoría baja en Sipeed Maixduino Elaboración propia.

Figura 44. *Clasificación de la categoría media en Sipeed Maixduino*



Nota. Imagen de clasificación de la categoría media en Sipeed Maixduino. Elaboración propia

En la (Tabla 7) se presentan las características de peso y tiempo de ejecución de los dos modelos que presentaron mejores resultados, en los tres sistemas de ejecución presentados en el capítulo 4, con el fin de hacer una comparativa del rendimiento de las redes entrenadas para clasificar el grado de severidad en las categorías de alto, bajo y medio en un foliolo de frijol.

Tabla 25. *Características de la ejecución de los dos mejores modelos de en los tres sistemas*

Sistema de ejecución	**Características**	**Modelo de red implementado**	
		MobileNetV2	**aXeleRate**
Google Colab	Peso	9.11 MB	7.9 MB
	Tiempo de predicción	3.42 ms	3.58 ms
Raspberry Pi 2 Model B V1.1	Peso	8.67 MB	7.6 MB
	Tiempo de predicción	761.3 ms	780.2 ms
Sipeed Maixduino	Peso	7.25 MB	2.2 MB
	Tiempo de predicción	------	134.37 ms

Nota. Elaboración propia

7. Conclusiones y recomendaciones

Para obtener un buen resultado en la clasificación de imágenes mediante redes neuronales es necesario contar con una base de datos amplia y bien clasificada. Cuando se trabajó inicialmente con el conjunto de imágenes aportadas por el trabajo de investigación previo de la Bióloga Miranda (Miranda, 2018), se obtuvo un porcentaje de precisión en validación inferior al 70%, posterior al aumento logrado bajo las condiciones de pandemia y temporada de cultivos se logró aumentar a un 91%. Se puede inferir que aumentando más de tres veces esta base de datos tal vez se logre clasificar los 9 niveles de severidad de la mancha anillada en foliolos de frijol como se muestra en la escala diagramática de la (Figura 3).

Mediante las matrices de confusión resultantes a las pruebas con las diferentes redes utilizadas, se puede notar que la categoría con las que más hubo equivocación fue la categoría media, por lo cual fue necesario rectificar la evaluación manual que se realiza en ImageJ y aumentar un poco el *dataset* perteneciente a esa categoría.

La arquitectura de red neuronal convolucional es ideal cuando se tiene imágenes a la entrada, ya que está diseñada para trabajar con entradas con más de 2 dimensiones. Gracias a las convoluciones que se llevan a cabo a lo largo del entrenamiento de la red, se logra extraer características o patrones fundamentales para la clasificación, en el caso de estudio la cantidad de manchas anilladas presentes en unos foliolos.

Para tener una red capaz de clasificar los foliolos en las tres categorías de severidad de la mancha anillada, se creó una red convolucional sencilla, pero los resultados no fueron los deseados y tenía demasiados parámetros lo cual la hace muy pesada para implementar en un sistema embebido no tan costoso. Por tanto, aplicar la técnica de *transfer learning* es una buena opción, ya que reduce el tiempo de entrenamiento y la cantidad de parámetros. El tiempo de

respuesta con esta técnica es acorde a lo esperado, puesto que, a menores parámetros, menores cálculos son necesarios.

La red MobileNetV2 arrojó buenos resultados en cuanto a la precisión en la fase de validación, además está hecha para tener un buen desempeño en dispositivos móviles y sistemas embebidos como la Raspberry Pi, aunque por tamaño se presentan limitaciones de RAM para ser implementada en la Maixduino, ya que el peso del modelo ocupa casi las 8 MB disponibles.

Usando el *framework* aXeleRate y un tipo de red MobileNet permitió obtener un modelo con precisión alta y peso óptimo para ser implementada en los dos sistemas embebidos.

A pesar de que la Raspberry Pi no tiene como uso específico implementar aplicaciones de *deep learning*, sino su uso es muy general, se lograron tiempos de clasificación menores a 800 ms por foliolo. En comparación a la tarjeta Maixduino que sí está diseñada para inteligencia artificial, pues se nota una disminución de tiempo alcanzando menos de 135 ms, por ello se recomienda implementar un acelerador de IA en la Raspberry Pi.

Cuando se compara los tiempos de ejecución en el entorno virtual de *Google Colaboratory*, la Raspberry Pi y la Maixduino, se puede notar que en el entorno es mucho más rápido con un promedio de clasificación por foliolo de 3.55 ms ya que se utilizan los recursos de una GPU robusta y potente como las establecidas en *Google Colab*.

Trabajar con herramientas como *Google Colaboratory* permite ejecutar modelos de redes con una gran cantidad de datos y parámetros sin problemas, pero se presentan limitaciones en el rendimiento cuando se pretenden ejecutar en sistemas embebidos. Por ello se recomienda a los desarrolladores de Sipeed aumentar la capacidad de memoria RAM.

Referencias Bibliográficas

American Physical society, Membership. (s.f). Epidemiología de las Enfermedades de las Plantas, El Progreso de la Enfermedad. https://www.apsnet.org/edcenter/disimpactmngmnt/topc/Epidemiologia/Pages/ProgresoEnfermedad.aspx

Benítez Iglesias, R., Escudero Bakx, G., Kanaan Izquierdo, S., & Masip Rodó, D. (2013). Intel· ligència artificial avançada, febrer 2013

Bórquez Ramírez, P., & Villanueva Ramos, J. (2006). Predicción de signo a tres semanas de la acción caterpillar con redes neuronales.

Carneiro, T., Da Nóbrega, R. V. M., Nepomuceno, T., Bian, G. B., De Albuquerque, V. H. C., & Reboucas Filho, P. P. (2018). Performance analysis of google colaboratory as a tool for accelerating deep learning applications. IEEE Access, 6, 61677-61685.

Chung, C. C., Chen, W. T., & Chang, Y. C. (2020, February). Using Quantization-Aware Training Technique with Post-Training Fine-Tuning Quantization to Implement a MobileNet Hardware Accelerator. In 2020 Indo–Taiwan 2nd International Conference on Computing, Analytics and Networks (Indo-Taiwan ICAN) (pp. 28-32). IEEE.

Culfaz, F. (2018, 6 de noviembre) Transfer Learning using Mobilenet and Keras. Blog. https://towardsdatascience.com/transfer-learning-using-mobilenet-and-keras-c75daf7ff299

Debouck, D. G., & Hidalgo, R. (1985). Morfología de la planta de frijol común. Programa de las Naciones Unidas (PNUD).

Fraile, M. E., García-Suárez, D., Martínez-Bernal, A., & Slomianski, R. (2007). Nutritivas y apetecibles: conozca de leguminosas comestibles. Parte I: hojas, vainas y semillas. Contactos, 66, 27-35.

Harveson, R. M., Schwartz, H. F., Urrea, C. A., & Yonts, C. D. (2015, 22 de octubre). Bacterial wilt of dry-edible beans in the central high plains of the US: past, present, and future. Plant Disease, 99(12), 1665-1677. Link. https://apsjournals.apsnet.org/doi/full/10.1094/PDIS-03-15-0299-FE

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., ... & Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861.

Intro to TensorFlow for Deep Learning Hyttsten, M., Delgado, J., & Bailey, P., (Cursado en el 2020) Link. https://www.udacity.com/course/intro-to-tensorflow-for-deep-learning--ud187

Neural Networks and Deep Learning. Ng, Andrew. (Cursado en el 2020) Link. https://www.coursera.org/learn/neural-networks-deep-learning

Introducción práctica con Keras Torres, J. Deep Learning: Link. https://torres.ai/deeplearning-inteligencia-artificial-keras/

Jara, C., y Giraldo, D. (2016). Manejo agronómico de frijol. Cartilla 1. Cali, Colombia: Centro Internacional de Agricultura Tropical (CIAT).

Khandelwal, R. (2019, 3 de febrero). Overview of different Optimizers for neural networks. Accessed on: Oct, 7. Blog. https://medium.com/datadriveninvestor/overview-of-different-optimizers-for-neural-networks-e0ed119440c3

Kohonen, T. (2012). Self-organization and associative memory (Vol. 8). Springer Science & Business Media.

López, W. Nuñez, L. (2020, 12 agosto). Sipeed Maix. Blog. https://docs.google.com/document/d/1JvX6oVMLQ8HlnLWLlir7miyEY3x57Z48kYExdhmUsXI/edit?usp=sharing

Maslov, D. (s.f). (en prensa). Github. Blog. https://github.com/AIWintermuteAI/aXeleRate

Mathworks, N. (2017). MA, USA. 2017. MATLAB. Blog. (no disponible actualmente)

Miranda Montero, Y. (2018). Distribución, Incidencia Y Severidad De La Mancha Anillada (Boeremia Spp.) Del Frijol En Los Departamentos De Antioquia, Tolima Y Huila, Colombia (Doctoral dissertation, Universidad Industrial de Santander, Escuela De Biología).

Miranda, Y. (2018). Distribución, Incidencia Y Severidad De La Mancha Anillada (Boeremia Spp.) Del Frijol En Los Departamentos De Antioquia, Tolima Y Huila, Colombia. [Fotografía].

Núcleo Ambiental S.A.S. Manual del frijol (2015). Cámara de comercio de Bogotá. https://bibliotecadigital.ccb.org.co/bitstream/handle/11520/14313/Frijol.pdf?sequence=1&isAllowed=y

Nuñez, L. (2021, 7 de agosto). Maixduino, Redneuronal.py. Blog. https://github.com/LisbethLorena/Maixduino/blob/main/RedNeuronal.py

Pajankar, A., Kakkar, A., Poole, M., & Grimmett, R. (2016). Raspberry Pi: Amazing Projects from Scratch. Packt Publishing Ltd.

Pastor Corrales, M. A. (1985). Enfermedades del frijol causadas por hongos. Programa de las Naciones Unidas (PNUD).p. 170.

Pattanayak, S. (2017). Pro deep learning with TensorFlow: a mathematical approach to advanced artificial intelligence in Python. Apress.

Raji, C. G., Vinish, A., Gopakumar, G., & Shahil, K. (2019, November). Implementation of Bitcoin Mining using Raspberry Pi. In 2019 International Conference on Smart Systems and Inventive Technology (ICSSIT) (pp. 1087-1092). IEEE.

Reyes, S., (2021, 7 de Agosto). Raspberry. Blog. https://github.com/samir0713/raspberry

Russell, S. J., & Norvig, P. (2004). Inteligencia Artificial: un enfoque moderno (No. 04; Q335, R8y 2004.). Pretince Hall.

Saha, S. (2018). A comprehensive guide to convolutional neural networks—the ELI5 way. Towards Data Science, 15.

Schmit, V. y Baudoin, JP (1992). Detección de resistencia al tizón de Ascochyta en poblaciones de Phaseolus coccineus L. y P. polyanthus Greenman. Investigación de cultivos de campo, 30 (1-2), 155-165.

Seem, R. C. (1984, 14 de octubre). Disease incidence and severity relationships. Vol. 22:133-150. Blog. https://www.annualreviews.org/doi/10.1146/annurev.py.22.090184.001025

Srivastava, N. Hinton, G. Krizhevky, A. Sutskever, I. Salakhuntdinov, R. (2014, enero) Dropout: a simple way to prevent neural networks from overfitting. Vol. 15, No. 1. The Journal of Machine Learning Research. Blog. https://dl.acm.org/doi/abs/10.5555/2627435.2670313

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research, 15(1), 1929-1958. Blog. https://dl.acm.org/doi/abs/10.5555/2627435.2670313

Thakur. K. (2019, 13 de mayo) Jupyter and Google Colab for AI startups. Blog. https://www.linkedin.com/pulse/jupyter-google-colab-ai-startups-kamal-thakur/

Treviño, C., & Rosas, R. (2013). El frijol común: factores que merman su producción. Revista de Divulgación Científica y Tecnológica de la Universidad Veracruzana, 26(1), 1-6.

Ulloa, J. A., Rosas Ulloa, P., Ramírez Ramírez, J. C., & Ulloa Rangel, B. E. (2011). El frijol (Phaseolus vulgaris): su importancia nutricional y como fuente de fitoquímicos. CONACYT.

Yani, M. (2019, May). Application of transfer learning using convolutional neural network method for early detection of terry's nail. In Journal of Physics: Conference Series (Vol. 1201, No. 1, p. 012052). IOP Publishing.

Apéndices

Apéndice A. *Código de programación para ejecutar una red neuronal entrenada en Raspberry Pi.*

```
# Importar librerías
import os
import argparse
import cv2
import time
import numpy as np
import sys
import glob
from PIL import Image
# Definir y analizar argumentos de entrada
parser = argparse.ArgumentParser()
parser.add_argument('--modeldir', help='Folder the .tflite file is
located in',
                    default='INTENTO2')
parser.add_argument('--graph', help='Name of the .tflite file, if
different than detect.tflite',
                    default='converted_model.tflite')
parser.add_argument('--labels', help='Name of the labelmap file, if
different than labelmap.txt',
                   default='/home/pi/tflite1/INTENTO2/labelmap.txt')
parser.add_argument('--threshold', help='Minimum confidence
threshold for displaying detected objects',
                    default=0.5)
parser.add_argument('--image', help='Name of the single image to
perform detection on. To run detection on multiple images, use
--imagedir',
                    default=None)
parser.add_argument('--imagedir', help='Name of the folder
containing images to perform detection on. Folder must contain only
images.',
                    default='FOTOSPRUEBA')
parser.add_argument('--edgetpu', help='Use Coral Edge TPU
Accelerator to speed up detection',
                    action='store_true')
def load_labels(filename):
  with open(filename, 'r') as f:
```

```
    return [line.strip() for line in f.readlines()]
args = parser.parse_args()
MODEL_NAME = args.modeldir
GRAPH_NAME = args.graph
LABELMAP_NAME = args.labels
min_conf_threshold = float(argsthreshold)
use_TPU = args.edgetpu
# Analizar el nombre y el directorio de la imagen de entrada
IM_NAME = args.image
IM_DIR = args.imagedir
# Si se especifican tanto una imagen como una carpeta, arroja un
error
if (IM_NAME and IM_DIR):
    print('por favor solo especificar una imagen o carpeta')
    sys.exit()
# Si no se especifica ni una imagen ni una carpeta, utilice de forma
predeterminada 'test1.jpg' para el nombre de la imagen
if (not IM_NAME and not IM_DIR):
    IM_NAME = '403.jpg'
# Importar bibliotecas de TensorFlow
# Si tflite_runtime está instalado, importe el intérprete de
tflite_runtime, de lo contrario, importe de tensorflow regular
pkg = importlib.util.find_spec('tflite_runtime')
if pkg:
    from tflite_runtime.interpreter import Interprete
else:
    from tensorflowlite.python.interpreter import Interprete

# Obtener la ruta al directorio de trabajo actual
CWD_PATH = os.getcwd()
# Definir la ruta a las imágenes y tome todos los nombres de archivo
de imagen
if IM_DIR:
    PATH_TO_IMAGES = os.path.join(CWD_PATH,IM_DIR)
    images = glob.glob(PATH_TO_IMAGES + '/*')
elif IM_NAME:
    PATH_TO_IMAGES = os.path.join(CWD_PATH,IM_NAME)
    images = glob.glob(PATH_TO_IMAGES)
# Ruta al archivo .tflite, que contiene el modelo que se utiliza
para la detección de objetos
```

```
PATH_TO_CKPT = os.path.join(CWD_PATH,MODEL_NAME,GRAPH_NAME)
# Ruta al archivo de etiquetas
PATH_TO_LABELS = os.path.join(CWD_PATH,MODEL_NAME,LABELMAP_NAME)
# Cargar etiquetas
with open(PATH_TO_LABELS, 'r') as f:
    labels = [line.strip() for line in f.readlines()]
# Cargar el modelo de Tensorflow Lite.
# Si usa Edge TPU, use el argumento especial load_delegate
interpreter = Interpreter(modelpath=PATH_TO_CKPT)
interpreterallocate_tensors()
# Obtener detalles del modelo
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
height = input_details[0]['shape'][1]
width = input_details[0]['shape'][2]
floating_model = (input_details[0]['dtype'] == np.float32)
input_mean = 255
input_std = 255
# Recorrer cada imagen y realice la detección
for image_path in images:
    # Cargar la imagen y cambie el tamaño a la forma esperada
[1xHxWx3]
    image = cv2.imread(image_path)
    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    image_resized = cv2.resize(image_rgb, (height, width))
    input_data = np.expanddims(image_resized, axis=0)
# Realizar la detección real ejecutando el modelo con la imagen como
entrada
    interpreter.set_tensor(input_details[0]['index'],input_data)
    start_time = time.time()
    interpreter.invoke()
    stop_time = time.time()
    output_data = interpreter.get_tensor(output_details[0]['index'])
    results = np.squeeze(output_data)
    top_k = results.argsort()[:-1][:1]
    labels = load_labels(args.labels)
    for i in top_k:
      if floating_model:
        print('{:08.6f} {}'.format(float(results[i]*100),
labels[i]))
```

```
      else:
        print('{:08.6f}%: {}'.format(float(results[i] / 255.0),
labels[i]))
    print('time: {:.3f}ms'.format((stop_time - start_time) * 1000))
    label = '%s: %.3f%%' % (labels[i], float(results[i]*100))
    cv2.putText(image, label, (5, 25),
    cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 0, 25), 2)
    cv2.imshow("Imagen", image)
    cv2.waitKey(0)
    # Presionar cualquier tecla para continuar con la siguiente
imagen, o presione 'q' para salir
    if cv2.waitKey(0) == ord('q'):
        break
# Limpiar
cv2.destroyAllWindows()
```

Apéndice B. *Código de programación para ejecutar una red neuronal entrenada en Sipeed Maixduino*

```
import sensor, image, time, lcd # Importar librerías
import KPU as kpu # Librería para realizar operaciones convoluciones
con imágenes
lcd.init()
sensor.reset()
sensor.set_pixformat(sensor.RGB565) # Formato de imagen
sensor.set_framesize(sensor.QVGA) # Resolución 320x240
sensor.set_windowing((224,224)) # Tamaño de img de entrenamiento
sensor.set_hmirror(0) # Espejo horizontal
lcd.clear()
labels = ['Alto', 'Bajo', 'Medio'] # Etiquetas con el mismo orden de
entrenamiento
task = kpu.load(0x200000) # Cargar el modelo desde la flash
kpu.set_outputs(task, 0, 1, 1, 3) # Última capa de modelo
while(True):
   ban1 = time.ticks_ms() # Bandera de inicio de tiempo
   kpu.memtest() # Test de memoria para las operaciones
   img = sensor.snapshot() # Captura de imagen
   fmap = kpu.forward(task, img) # enviar la img al modelo
   plist = fmap[:] # Vector salidas de predicción
   pmax = max(plist) # Max valor de probabilidad
```

```
   max_index = plist.index(pmax) # Asociación para etiqueta
   ban2 = time.ticks_ms() # Bandera de fin de tiempo
   tim = time.ticks_diff(ban2, ban1) # Tiempo de predicción
   a = img.draw_string(0,0, str(labels[max_index].strip()), color=
(255,0,0), scale=2) # Imprimir etiqueta
   a = img.draw_string(0,20, str(pmax), color= (255,0,0),scale=2) #
Porcentaje de predicción
   a = img.draw_string(0,40, str('Time_ms'), color= (255,0,0),
scale=2)
   a = img.draw_string(0,60, str(tim), color= (255,0,0), scale=2) #
Tiempo de predicción
   print((pmax, labels[max_index].strip()))
   a = lcd.display(img) # Mostrar resultados
a = kpu.deinit(task)
```