Diseño e implementación de un sistema de procesamiento de imágenes para determinar la calidad del Limón Tahití.

Jeison Duvan Imbaña Portillo y William Fernando López Rueda

Trabajo de Grado para optar al título de Ingeniero Electrónico

Director

Jaime Guillermo Barrero Pérez

Magíster en Potencia Eléctrica

Universidad Industrial de Santander

Facultad de Ingenierías Fisicomecánicas

Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones

Bucaramanga

2021

Dedicatoria

Dedicado a mi madre, Olga Marlene Portillo Salazar, primero, por haberme regalado el privilegio de vivir. Además, por brindarme la oportunidad de crecer profesionalmente y darme de forma tan incondicional su tiempo, sus sabios consejos, su vasto apoyo, sus incansables esfuerzos, sus oraciones y su amor, ese infinito amor que me hace sentir día tras día, que sin duda fue fundamental para el proceso y culminación de esta etapa. A ella, por ser la preclara columna en la que se ha sustentado mi vida.

A mis hermanos Cristian Zambrano y Maicol Zambrano, y a mi Padrastro James Norvey Zambrano, este logro es en gran parte gracias a ustedes. A ellos por la infinita confianza y el constante acompañamiento, por ser la motivación que me impulsó en los momentos más decadentes y sobre todo por enseñarme que cualquiera te puede fallar, excepto la familia y tengo la certeza de que si desfallezco podré contar con ellos.

Jeison Imbaña

Dedico este logro inicialmente a Dios por permitirme llegar a donde estoy, a mi familia la cual siempre me apoyó de la mejor manera con lo que estuvo a su alcance; a los profesores, que con sus enseñanzas me han inculcado el conocimiento que requerí para cumplir con este proyecto y a mis compañeros que me orientaron y ayudaron a lo largo de mi estancia en la universidad.

William López

Agradecimientos

Agradecimientos a nuestras familias que nos han suscitado y apoyado tanto en cada paso de este proyecto de vida que hemos decidido emprender. A nuestros compañeros y amigos con los que compartimos tanto alegrías como tristezas y que hicieron esta etapa de la vida mucho más amena de lo que imaginamos. Y los profesores, pieza fundamental en nuestro desarrollo profesional, que siempre brindaron su tiempo, paciencia y conocimiento con la finalidad de formar no sólo excelentes profesionales, sino excelentes personas, principalmente a nuestro director Jaime Barrero que nos guió arduamente en el desarrollo y culminación de la tesis.

Tabla de Contenido

Lista de Figuras

Lista de Tablas

Resumen

Título: DISEÑO E IMPLEMENTACIÓN DE UN SISTEMA DE PROCESAMIENTO DE IMÁGENES PARA DETERMINAR LA CALIDAD DEL LIMÓN TAHITÍ. *

Autor: Jeison Duvan Imbaña Portillo, William Fernando López Rueda. **

Palabras Clave: Redes Neuronales Convolucionales (RNC), MobileNet, Maix Bit, Raspberry, kmodel, tflite.

Descripción: Teniendo en cuenta que el departamento de Santander ha incrementado su producción de limón Tahití y que los criterios de calidad para la exportación son cada vez mayores, es necesario aportar a estos procesos con tecnología que facilite la selección y clasificación que deben realizar los productores campesinos. Por lo anterior, el objetivo principal de este proyecto fue mejorar la selección de frutos para hacerla más autónoma y estandarizada, evitando así la subjetividad de las personas encargadas de esta labor. Para ello, se propuso el uso de **IA** (Inteligencia Artificial) para automatizar el proceso de identificación de características y clasificación del tipo de limón Tahití. En la solución se planteó el uso de redes neuronales convolucionales pre-entrenadas. De allí se desprenden varios requisitos siendo el principal, seleccionar la arquitectura de la red y el sistema embebido que la soporta. Adicionalmente fue importante la calidad y cantidad de datos utilizados para el entrenamiento de esta red neuronal, por esta razón se acondicionó una base de datos con imágenes de limones Tahití de diferentes características. La validación del modelo implementado en la tarjeta Maix Bit da como resultado un acierto del 94 % gracias a su acelerador de operaciones convolucionales, esto con un tiempo de predicción de aproximadamente 137 [ms].

---

* Trabajo de grado.

** Facultad de Ingeniería Fisicomecánicas. Escuela de Ingenierías Eléctrica, Electrónica y Telecomunicaciones. Director: Jaime Guillermo Barrero Pérez, Magíster en Potencia Eléctrica.

Abstract

Title: DESIGN AND IMPLEMENTATION OF AN IMAGE PROCESSING SYSTEM TO DETERMINE THE QUALITY OF TAHITI LEMON. *

Author: Jeison Duvan Imbaña Portillo, William Fernando López Rueda. **

Keywords: Convolutional Neural Networks (CNN), MobileNet, Maix Bit, Raspberry, kmodel, tflite.

Description: Taking into account that the department of Santander has increased its production of Tahitian lemons and that the quality criteria for export are increasingly higher, it is necessary to contribute to these processes with technology that facilitates the selection and classification that must be carried out by the peasant producers. Therefore, the main objective of this project was to improve fruit selection to make it more autonomous and standardized, thus avoiding the subjectivity of the people in charge of this task. For this purpose, the use of **IA** (Artificial Intelligence) to automate the process of identification of characteristics and classification of the type of Tahiti lemon. In the solution, the use of pre-trained convolutional neural networks was proposed. The main requirement was to select the architecture of the network and the embedded system that supports it. Additionally, the quality and quantity of data used for the training of this neural network was important; for this reason, a database with images of Tahiti lemons of different characteristics was conditioned. The validation of the model implemented in the Maix Bit card resulted in a 94 % accuracy thanks to its convolutional operations accelerator, with a prediction time of approximately 137 [ms].

---

* Bachelor Thesis.

** Faculty of Engineering and Physical Sciences. School of Electronic and Electrical Engineering. Director: Jaime Guillermo Barrero Pérez, Master in Electrical Power.

Introducción

El limón Tahití, también conocido como limón persa es de los cítricos que se abren espacio en Colombia gracias a la fuerte demanda internacional. De este tipo de lima ácida corresponden alrededor del 20 % de las plantaciones de cítricos sembradas en todo el territorio colombiano. Algunas de las características que lo hacen destacar es su falta de semillas, su color verde intenso y la forma ovalada que posee; razones por las cuales es un fruto con alta demanda en la industria gastronómica y coctelera. Adicionalmente y debido a los tratados de libre comercio que existen esta fruta no tiene ningún arancel, situación que ha incidido en el aumento de la producción de este cítrico en los últimos años, siendo los principales importadores Estados Unidos y los Países Bajos según lo indica el Sistema de Información de Gestión y Desempeño de las Organizaciones de Cadenas (SIOC, 2019).

Uno de los requerimientos que se deben satisfacer para entrar en el mercado internacional es cumplir con los estándares de calidad que exigen los países importadores, para ello es necesario hacer una selección de las frutas; esta es realizada manualmente cuando los productores son de medianas o pequeñas empresas y no se tienen los recursos para adquirir sistemas industrializados de selección. Es por ello que se planteó una solución a esta problemática a fin de evitar errores humanos y los costos agregados que acarrea este método de selección; para ello se usaron las técnicas de visión artificial tales como clasificación de imágenes mediante el uso de redes neuronales convolucionales y fue implementada con un sistema de procesamiento de imágenes el cual extrae las características más importantes al momento de la selección las cuales son: uniformidad en el

color de la piel, ausencia de manchas y ausencia de plagas.

Para lograr ello se realizó una base de datos desde cero, esta consta de fotografías de limones divididas en tres clases correspondientes a las características que se describieron anteriormente; naturalmente, es necesario contar con un sistema que pueda realizar la captura de las imágenes y su procesamiento; razón por la que se seleccionaron los sistemas embebidos Raspberry Pi 3 y la tarjeta Maix Bit que cumplen con estas características. Del mismo modo se seleccionó una arquitectura de red neuronal que cumpliera con ciertos requisitos como lo son: un bajo peso de modelo y un buen desempeño en cuanto a predicciones; una vez seleccionado este se realizó el entrenamiento y su posterior validación en los dos sistemas embebidos y se realizó un análisis de resultados donde se comparó su desempeño.

La estructura del presente trabajo corresponde a un capítulo inicial donde se describen los conceptos claves en el campo de la visión artificial y la clasificación de imágenes; en el segundo capítulo se habla acerca de la base de datos, la arquitectura de la red y como fue la implementación del modelo; en el tercer capítulo se aprecian los resultados obtenidos del entrenamiento y la validación para cada uno de los sistemas embebidos y finalmente un capítulo de conclusiones y recomendaciones para posteriores trabajos en este campo.

1. Objetivos

1.1. Objetivo general

Diseñar un sistema de procesamiento de imágenes que permita identificar la calidad del Limón Tahití.

1.2. Objetivos específicos

Acondicionar una base de datos de imágenes de limón Tahití para entrenar la red neuronal.

Seleccionar un sistema embebido con la capacidad de capturar imágenes y que permita realizar el procesamiento de las mismas.

Seleccionar una arquitectura de red neuronal que permita el procesamiento de imágenes de limones Tahití para detectar la calidad de estos.

Entrenar y validar la red neuronal aplicando técnicas de optimización, para reconocer la calidad de limones Tahití.

Implementar y realizar las pruebas correspondientes al funcionamiento de la red neuronal dentro del sistema embebido.

2. Marco teórico

2.1. Estado del arte

En los últimos años el uso de los algoritmos de inteligencia artificial como las redes neuronales ha venido aumentado, esto es debido a la nueva aparición de dispositivos capaces de realizar más operaciones al mismo tiempo tales como GPUs (*Graphics Processing Unit*) y TPUs (*Tensor Processing Unit*); es por esto que los problemas como la visión artificial, el reconocimiento de objetos y de voz han logrado destacar actualmente. De igual forma todo ello no sería posible si no se cuentan con sistemas que puedan aprender a reconocer estos patrones, entre ellos destacan el MobileNet, NASNet, EfficienceNet e Inception como modelos lo suficientemente versátiles para ser cuantizados y usados en aplicaciones móviles.

Debido a la capacidad de ser comprimidos estos modelos pueden implementarse en dispositivos cuyo *hardware* sea limitado, por ejemplo: teléfonos inteligentes y microcontroladores; estos últimos abren un gran campo de aplicación para el uso de las redes neuronales como control de calidad y seguridad tanto a nivel industrial como residencial.

Uno de los trabajos más recientes es el de Muñoz (2019) en el cual se hace uso de redes neuronales profundas para la clasificación de frutas y verduras todo esto implementado en un teléfono móvil, una Raspberry y un kit de desarrollo Jetson TX2.

Adicionalmente Deebul *et al.* (2020) realizó un estudio de microcontroladores para aplicaciones de visión por máquina en el cual puso a prueba el desempeño de tres diferentes dispositivos entre los cuales están: la Maix Bit, JeVois A33 y OpenMV H7. De este estudio se observó un

buen desempeño de la Maix Bit la cual cuenta con buenos índices de latencia y *throughput* en comparación con los otros dos sistemas.

2.2. Inteligencia Artificial

La inteligencia artificial (IA) es un campo de investigación que busca la forma en que una máquina pueda "pensar" con vistas a ir más allá de las instrucciones que se le puedan programar tal como indica Russell *et al.* (2004). Este campo tiene asociados sus orígenes usualmente a el matemático Alan Turing, quien en 1950 publicó un artículo académico llamado: "Computing Machinery and Intelligence"; en este, él se cuestiona acerca de la posibilidad de que las máquinas puedan razonar y para ello ideó *Imitation game* también conocido como el Test de Turing (Copeland, 2004). Con su invención él planteaba una situación en la que se ven envueltas tres partes, la primera de ellas es un interrogador (humano) que debe formular preguntas a las otras dos partes: una máquina y un humano para descubrir cual de los dos es la máquina, esto es debido a que este desconoce la identidad de las mismas. De esta forma, si la máquina logra hacer creer a su interlocutor que es humana, se le considerará como inteligente. Con ello se sientan las bases que permiten medir la inteligencia de las máquinas; este test aún hoy en día es usado bajo el nombre de "CAPTCHA" (acrónimo de: *Completely Automated Public Turing test to tell Computers and Humans Apart*) en la verificación de identidad por muchas páginas en línea.

2.3. Aprendizaje Automático

El aprendizaje automático (también conocido como *Machine Learning*), es una rama de la IA que busca mejorar el desempeño de algoritmos de procesamiento de datos a partir de la experiencia (Mitchell, 1997). Esto se realiza con diversos métodos de aprendizaje tales como:

- Arboles de decisión
- Redes neuronales
- Redes Bayesianas
- Algoritmos Genéticos
- Análisis de regresión

2.4. Redes Neuronales

Es una rama del Aprendizaje automático (*Maching Learning*) que logra aprender con base a su experiencia tal como se menciono anteriormente, utilizando una estructura que guarda cierta relación con las neuronas del sistema nervioso humano. “La neurona es una simple unidad procesadora que recibe y combina señales desde y hacia otras neuronas. Si la combinación de entradas es suficientemente fuerte la salida de la neurona se activa.” (Olabe (1998), p. 2-3). Todo este sistema biológico de conexiones entre neuronas se le conoce como redes neuronales biológicas.

“Una de las unidades analógicas mas básica de la neurona se denomina perceptrón, estos fueron desarrollados en las décadas de 1950 y 1960 por el científico Frank Rosenblatt , inspirado en trabajos anteriores de Warren McCulloch y Walter Pitts.” (Nielsen, 2015). La conexión de todas estas neuronas artificiales se les denominó en el subcampo de las ciencias de la computación, como redes neuronales (*neural networks*).

2.4.1. Perceptrón.

Es una representación artificial, muy básica, de una neurona biológica. Nielsen (2015) propone que su composición consta de varias entradas, x1, x2, ... xn; además de pesos, w1, w2, ... wn; que expresan la importancia de las respectivas entradas, posteriormente se produce una única salida binaria que se determinada por si la suma ponderada de la estradas es menor o mayor a un umbral establecido. Este umbral al final se cambia añadiendo un sesgo (b) al perceptrón entonces la salida de este se puede escribir como:

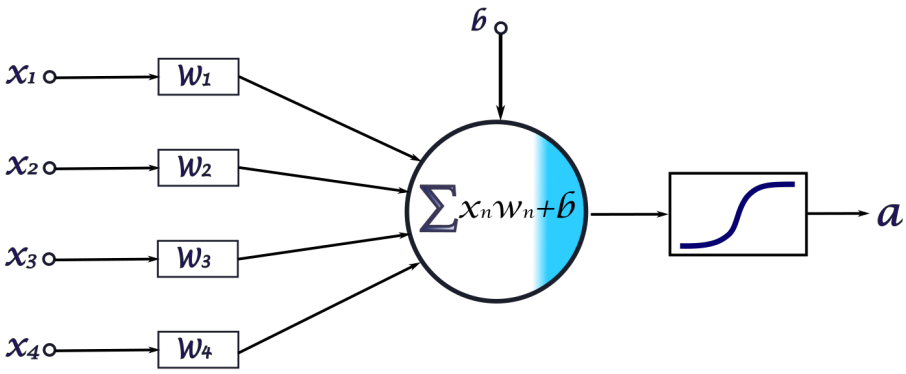
$$z = \sum_{i=1}^{n} w_i * x_i + b$$

$$\text{f(z)} = \left\{ \begin{array}{l} 0 \mapsto si \;\; z < 0 \\ \\ 1 \mapsto si \;\; z \geq 0 \end{array} \right\}$$

Un perceptrón mas complejo se logra al cambiar la función con la que se guía su salida, se quiere una salida que al modificar sus pesos o entradas se modifique proporcionalmente su salida pero con la función actual no cumple ese requerimiento ya que solo proporciona 0 o 1, por ende esta es la principal modificación que sufre.

La forma completa de la neurona artificial queda con las entradas x1, x2, ... xn, tomando valores entre 0 y 1, esto ayuda a que el entrenamiento sea mas rápido, estas se multiplican por sus respectivos pesos, w1, w2, ... wn; se suman todos los valores para luego agregar el sesgo (b) y el valor final pasa por una función denominada función de activación que ayuda a satisfacer alguna especificación que la neurona este intentado resolver, como su linealidad, aprendizaje etc. Los valores finales dependen por ende de esta función de activación. (Nielsen, 2015).

Figura 1. Diagrama de una neurona artificial.

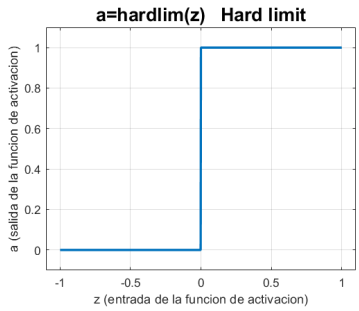
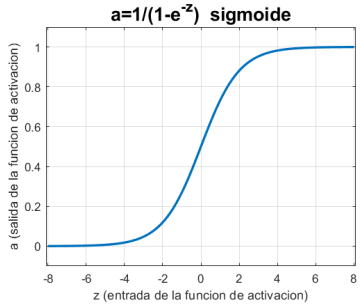
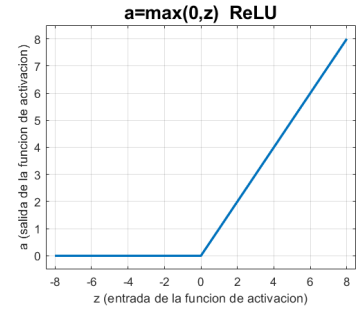


Nota. Esta figura muestra la relación en la que las entradas y salidas operan dentro de una red neuronal.

2.4.2. Función de activación.

Las funciones de activación se adaptan según el problema a resolver y se buscan de forma tal que las derivadas sean simples; esto para minimizar el coste computacional de su aprendizaje. (Demuth *et al.*, 2014). En la siguiente tabla se pueden apreciar las más populares:

Tabla 1
Funciones de activación.

Función de activación	Fórmula	Gráfica
Hard limit o Escalón	$f(z) = \begin{Bmatrix} 0 \mapsto si \;\; z < 0 \\ 1 \mapsto si \;\; z \geq 0 \end{Bmatrix}.$	
Sigmoid o Sigmoide	$f(z) = \frac{1}{1-e^{-z}}.$	
ReLU (*Rectified Linear Unit*)	$\begin{Bmatrix} 0 \mapsto si \;\; z < 0 \\ z \mapsto si \;\; z \geq 0 \end{Bmatrix}.$	
Softmax	$f(z)j = \frac{e^{zj}}{\sum_{k=1}^{k} e^{zk}}$ para j = 1, . . . , K.	La salida de la función softmax corresponde a una distribución de probabilidad sobre K posibles salidas donde z es el valor de la red antes de pasar por la función de activación.

2.4.3. Arquitectura de redes neuronales.

Según Soria y Blanco (2001) indica que las redes neurales se puede formar de diferentes estructuras o modelos y se clasifican según el numero de capas, tipo y grado de conexión.

2.4.3.1. Clasificación según el número de capas.

Redes neuronales monocapa: Como su nombre lo indica posee una sola capa, una capa es una organización de perceptrones los cuales solo poseen entradas y sus respectivas salidas, por lo general a la capa de entrada no se le tiene en cuenta para el número total de capas.

Redes neuronales multicapa: a veces solo una capa no es suficiente por lo que es necesario agregar más capas en una especie de conexión en paralelo, con estas conexiones y algoritmos como el *backpropagation* logramos entrenar de forma mas robusta la red. Se denomina capa profunda a todas las capas diferentes a la entrada y salida, de ahí el término *Deep Learning*.

2.4.3.2. Clasificación según el tipo de conexión.

Redes neuronales recurrentes: Este tipo de red permite tener realimentación por medio de lazos que se pueden conectar entre neuronas de diferente capa, neuronas de la misma capa e incluso entre la misma neurona.

2.4.3.3. Clasificación según el grado de conexión.

Redes neuronales parcialmente o totalmente conectadas: Cuando son totalmente conectadas, todas las neuronas de una capa van conectadas a la capa siguiente, cuando no se conectan

todas las salidas de una red a la siguiente capa se denomina parcialmente conectadas (Soria y Blanco, 2001).

Todas las estructuras anteriores se pueden combinar entre si dependiendo de la necesidad o aplicación donde se vaya a utilizar la red neuronal.

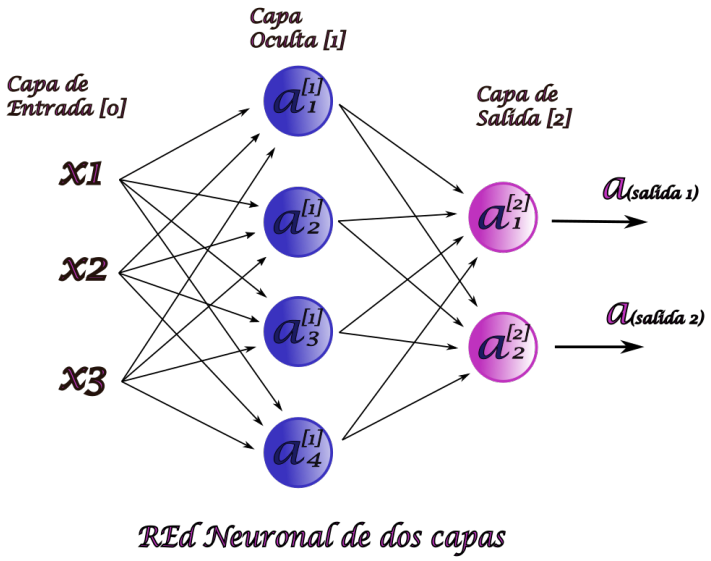
2.4.4. Funcionamiento.

El funcionamiento de las redes neuronales que se desprenden del perceptrón son muy similares, pero su entrenamiento varia dependiendo de si es un sistema supervisado o no supervisado. Según Rivas Asanza y Mazón Olivo (2018), “Las redes neuronales están conformadas por neuronas artificiales interconectadas entre si y distribuidas por capas donde para que pueda fluir la información las neuronas de cada capa se conectan con las neuronas de las capas siguientes”.

Toda neurona de cada capa posee su respectivos pesos (W), un sesgo (b) y su respectiva función de activación (σ), la salida de estas se conectan con las capas siguientes y así se distribuye la información, cuando el aprendizaje es supervisado se hace necesario tener los valores de las salidas para supervisar la exactitud de la red y disminuir su error para mejorar su aprendizaje.

Los datos que se proporcionarán serán teniendo como ejemplo una red neural de dos capas con aprendizaje supervisado, la capa oculta y la de salida, las entradas no se consideran una capa.

Figura 2. Ejemplo de red neuronal.



Nota. Esta figura muestra la función visualmente una red de 1 sola capa oculta en la cual hay 4 neuronas.

2.4.5. Algoritmo de entrenamiento.

Las redes que utilizan *backpropagation* usan el aprendizaje supervisado el cual consiste en suministrar a la red la información de entrada y salida de forma que esta descubra patrones en los datos que le permitan asociar las salidas a las entradas disminuyendo de esta forma el error. Este algoritmo se constituye de dos fases: propagación hacia adelante y la propagación hacia atrás; esto teniendo en cuenta que para cada sesión de entrenamiento se deben realizar las dos fases. (Olabe, 1998).

2.4.5.1. Propagación hacia adelante (*foward*). La propagación hacia adelante se divide en 2 pasos:

Primero, las entradas pasan por los pesos de las neuronas, se hace una suma ponderada antes de agregar el sesgo.

$$Z=W.t *X + b$$

Luego este valor (Z) pasa por una función de activación y su resultado se convierte en una nueva entrada para la siguiente capa.

$$a = \sigma(z)$$

El valor final que sale de la red neuronal se compara con los valores reales y se cuantifica el error que existe con la función de costo, pero este último paso de cuantificar solo se realiza para re-entrenar la red.

2.4.5.2. Propagación hacia atrás (*backward*). Cuando se tiene la función de costo se busca disminuir el error de la salida, para lo cual se necesita modificar los valores de pesos y sesgos de la red. Para lograr esto, se inicia con el descenso de gradiente, que en resumidas cuentas es un cálculo de las derivadas de la función de costo con respecto a los parámetros de peso y el sesgo, luego se modifican los parámetros con una tasa de aprendizaje extra.

$$W = W - \alpha*dW$$

$$b = b - \alpha*db$$

dw = derivada de la función de costo con respecto al peso.

db = derivada de la función de costo con respecto al sesgo.

α = tasa de aprendizaje.

Tal como nos indica Nielsen (2015) en su libro, se define la función de costo como una ecuación que nos permite comparar la salida de la red neuronal con el valor real, con esta comparación se cuantifica que tan bien está haciendo la predicción la red neuronal. La **función de costo** más conocida es la función de costo o error cuadrático medio, aunque existen muchas otras.

$$C(W,b) = \sum \left(\frac{1}{2m} * |y - a|^2 \right)$$

m = numero de ejemplos utilizados.

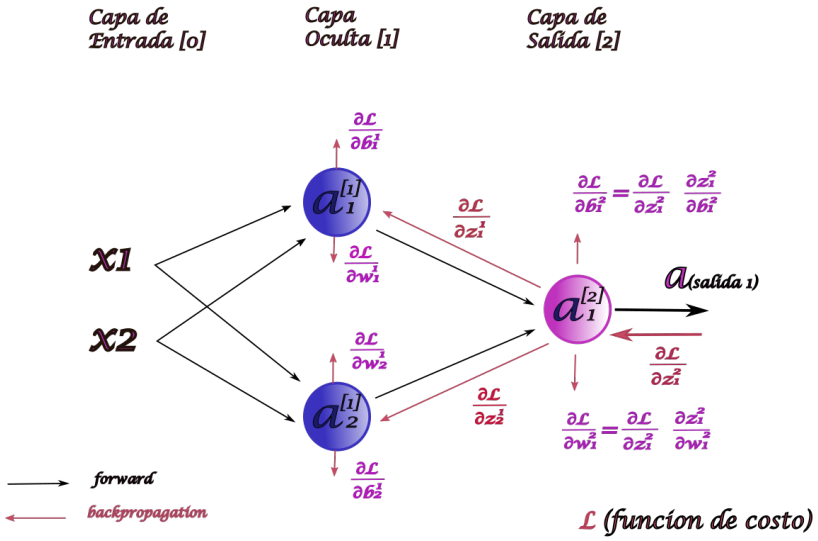
y = valor real.

a = valor que la red proporciona.

Por lo que en resumen, el objetivo del algoritmo de entrenamiento es minimizar el costo de C(W,b) en función de los pesos (W) y los sesgos(b) de las neuronas utilizando el algoritmo conocido como descenso del gradiente (Nielsen, 2015).

Algoritmo *backpropagation*: Esta es una técnica que ha aportado la solución para entrenar redes con bastantes capas ocultas ya que facilita encontrar la derivada parcial de cada neurona y ver que tanto aportó en el error del sistema en general sin afectar a las neuronas que no se encuentran involucradas. Para ello se utilizan los resultados obtenidos previamente en la propagación hacia adelante.

Figura 3. Algoritmo *backpropagation* en una red neuronal.



Nota. Esta figura muestra como recorre el algoritmo *backpropagation* una red neuronal.

$$\frac{\partial L}{\partial z_1^2} = \frac{\partial L}{\partial a_1^2} * \frac{\partial a_1^2}{\partial z_1^2}$$

$$\frac{\partial L}{\partial w_1^2} = dW = \frac{\partial £}{\partial z_1^2} * \frac{\partial z_1^2}{\partial w_1^2}$$

$$\frac{\partial L}{\partial b_1^2} = db = \frac{\partial L}{\partial z_1^2} * \frac{\partial z_1^2}{\partial b_1^2}$$

$\frac{\partial L}{\partial a_1^2}$ (varía según la función de costo utilizada)

W = W - α*dW

b = b - α*db

α = tasa de aprendizaje.

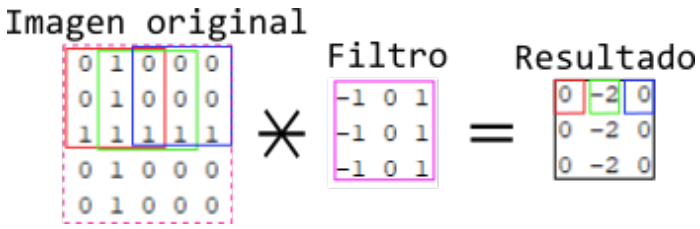
2.5. Redes neuronales convolucionales

A diferencia de las redes completamente conectadas, las redes convolucionales se presentan como una forma más acertada de abordar la clasificación de imágenes debido a que tienen en cuenta las posiciones relativas que tienen los píxeles con respecto a los demás en la imagen; de está forma se extraen las características más relevantes; esto se hace mediante la convolución de la imagen con diferentes filtros a lo largo de la red; donde cada uno de estos está compuesto a su vez de unidades neuronales dispuestas en forma de matriz cuadrada. Como no esta la necesidad de que cada píxel de la imagen este asociado a una unidad neuronal se reduce el número de parámetros (W y b) que debe aprender la red. (Bishop, 2006; Nielsen, 2015).

De una forma más técnica una red neuronal convolucional es un sistema que toma como entrada una imagen y la modifica mediante filtros que se encuentran en cada una de las capas que lo conforman; a medida que la imagen atraviesa el sistema, se modifican sus dimensiones de forma que termina por convertirse en una matriz tridimensional de área más pequeña que la original, pero con una profundidad mayor. Finalmente se debe pasar a una (o unas) capa densa o completamente conectada, en ellas se decide a que clase corresponde la imagen en cuestión.

Convolución en imágenes: Esta operación se hace moviendo un filtro por pequeñas regiones de la imagen, este realiza una multiplicación punto a punto con los píxeles de la imagen y posteriormente suma cada uno de estos valores para obtener un resultado asociado a la cada región de la imagen. En la figura 4 se aprecia la operación.

Figura 4. Ejemplo del operador convolución en imágenes.



Nota. Esta figura muestra el operador convolución en acción.

Según Aggarwal (2018); Rungta (2019) la forma en la que se modifican las dimensiones a la salida de cada una de las capas del sistema depende de tres parámetros importantes:

- *Padding* (P): Esto un " ajuste " que se hace a la imagen (matriz) de entrada con el fin de hacer que la matriz a la salida tenga unas dimensiones definidas; por ejemplo, se puede ajustar este parámetro para que las dimensiones de la imagen de salida sean las mismas que la de entrada (a esto se le denomina " *same padding* "). Para lograr esto se agrega un " borde " con nuevos píxeles a la imagen, el grosor (en píxeles) de este depende del tamaño del filtro que se vaya a usar.

- *Filter* (F): Este es el filtro en sí, contiene las unidades neuronales que se va a usar para extraer las características de la imagen. Usualmente es de dimensiones impares. (ej. 3x3, 5x5, 7x7, etc).

- *Stride* (S): Este parámetro define el " paso " o número de píxeles que se mueve el filtro a través de la imagen. En otras palabras, cada cuantos píxeles se va a poner nuevamente el filtro.

- Número de filtros (*filters*): Es el número de filtros con los cuales se hace la convolución; este modifica la cantidad de canales de la imagen de salida. En términos más convencionales, este define la profundidad de la imagen (matriz) de salida.

Estos parámetros modifican las dimensiones de la imagen de salida tal que: si se tiene una imagen de dimensiones $[N_H x N_W x N_C]$ (donde N_H es el alto en píxeles, N_W el ancho en píxeles y N_C es el numero de canales de la imagen; si la imagen es RGB $N_W = 3$), las dimensiones de salida serán:

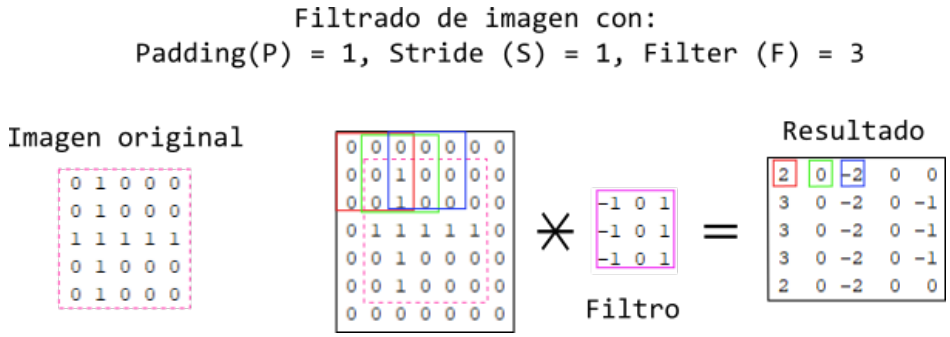
$$N_H new = floor((N_H + 2P - F)/S + 1)$$

$$N_W new = floor((N_W + 2P - F)/S + 1)$$

$$N_C new = filters$$

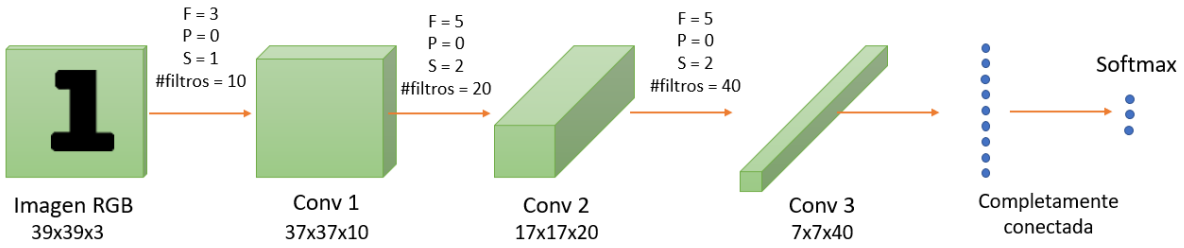
Como aclaración, $floor$ es la operación de redondeo inferior.

Figura 5. Padding, Stride y *Filter* en una imagen 5x5x1.



Nota. Esta figura muestra el filtrado de una imagen mediante el uso de distintos parámetros y como quedan sus dimensiones de salida.

Figura 6. Imagen a través de una red convolucional.



Nota. Esta figura muestra la transformación de una imagen al atravesar una red convulucional.

2.5.1. Tipos de capas en redes neuronales convolucionales.

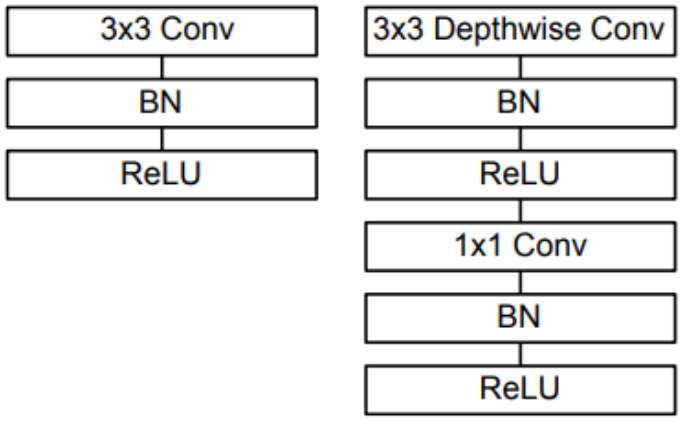
Las redes neuronales convolucionales (RNC) tienen tres tipos de capas que se usan para procesar las imágenes, el primer tipo: **capa convolucional** es la más importante y en ella, como se explica arriba; se realizan las convoluciones con los filtros y se aprenden un número de parámetros equivalente a las dimensiones del filtro en la etapa de entrenamiento. El segundo tipo: **capa *pooling*** es una capa que permite reducir el tamaño de la matriz de entrada debido a que extrae un valor máximo (*max pooling*) o un valor promedio (*average pooling*) por determinadas regiones de la matriz (estas regiones van sujetas al paso (*stride*) y el tamaño del filtro (*filter*) con el que se define la capa); debido a que estas capas no necesitan ninguna función de activación no tienen que aprender ningún parámetro en la etapa de entrenamiento y, estas capas son usadas normalmente después de una capa convolucional para condensar la información. (Nielsen, 2015; Rungta, 2019). Finalmente se tiene la **capa *fully connected*** en la cual se realiza la identificación de la clase a la que corresponde la imagen de entrada; estas capas son usadas al final de la red, y, el número de

parámetros que deben aprender depende de las dimensiones de la matriz de entrada.

2.6. Red MobileNet-V1

Esta red se basa en convoluciones separables profundas o *depthwise separable convolution* por su nombre en inglés, estos son bloques constituidos por: una **convolución profunda** (ocurre cuando para cada canal de entrada (N_C) existe solamente un filtro, por lo tanto $N_C NEW = N_C$) seguidos de una **convolución 1x1** la cual sirve para combinar los canales de entrada en un nuevo bloque con diferente número de canales de salida; esto ayuda a reducir el número de operaciones que se deben realizar. (Howard *et al.*, 2017). En la imagen 7 se pueden ver un bloque de convolución normal y el bloque que usa la red MobileNet (*depthwise separable convolution*).

Figura 7. Tipos de convolución de la red MobileNet-V1.



Nota. A la derecha la convolución estándar y a la izquierda la convolución profunda separable. *Fuente*. Imagen recuperada de: `https://arxiv.org/pdf/1704.04861.pdf`

Adicionalmente esta red cuenta con dos operadores o *hyper-parameters*, estos le permiten modificar la cantidad de parámetros que debe aprender y por consiguiente el peso del modelo. El

primero de ellos es el multiplicador de ancho (α) este regula el número de bloques convolucionales totales de la red; el segundo es el multiplicador de resolución (ρ) que como su nombre indica, modifica la resolución de entrada. α varía entre 0 y 1 mientras que ρ entre 128 y 224 píxeles, pero sus valores efectivos son:

Tabla 2
MobileNet-V1 hyper-parameters.

α	ρ
1	224
0.75	192
0.5	160
0.25	128

Note. Valores efectivos para α y ρ.

Para el caso del multiplicador de ancho en la siguiente tabla se puede apreciar como es la reducción de parámetros; de igual forma si se compara la versión completa que tiene 4.2 millones con redes como la VGG16 que cuenta con 138 millones se nota una diferencia abismal que repercute directamente en el tiempo de computo.

Tabla 3
MobileNet-V1 con diferente multiplicador de ancho.

	Millones	
Modelo	Mult-Add	Parametros B
MobilNet 1_0	569	4.2
MobilNet 0_75	325	2.6
MobilNet 0_50	149	1.3
MobilNet 0_25	41	0.5

Figura 8. Arquitectura de la red MobileNet-V1.

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. Acá se presenta el modelo de capas de la red MobileNet-V1. *Fuente*. Imagen recuperada de: `https://arxiv.org/pdf/1704.04861.pdf`

Para más información acerca de esta red ver: (Howard *et al.*, 2017)

2.7. Transferencia de conocimiento

También conocido por su nombre en ingles como: *Transfer-Learning*, es una estrategia usada ampliamente en la actualidad que hace uso de modelos pre-entrenados; la principal ventaja de esto es que las redes ya han aprendido a identificar patrones para una gran cantidad de clases esto es porque su entrenamiento se hace con bases de datos de 1000 clases como por ejemplo: Image-

Net. Algunas de las redes más populares que han sido entrenadas con estas bases son: MobileNet, ResNet, VGG16, VGG19, NASNet, Inception. Para hacer uso de estas redes se debe hacer un re-entrenamiento en el cual se bloquean todas sus capas a excepción de las capas finales; estas se modifican para la aplicación de interés. Por ejemplo si se desea re-entrenar un modelo con una base de datos de 3 clases, se cambia la ultima capa por una de 3 neuronas. Al tener que entrenar solamente unas pocas neuronas se reduce el tiempo de computo necesario para entrenar un modelo desde cero y se aprovechan las características que han aprendido modelos previos.

Otra técnica que ayuda a mejorar el modelo de predicción es *data augmentation* que consiste en aumentar tamaño del conjunto de datos realizando pequeñas perturbaciones a los datos originales, por ejemplo, en el área de imágenes se logra con modificaciones como rotaciones, zoom, cambio de brillo entre otras esto ayuda a que el conjunto de entrenamiento sea más robusto a variaciones.

2.8. Limón Tahití

La lima ácida o limón Tahití (también conocido como Limón Persa) pertenece a la familia botánica *Rutaceae* y al género *citrus*; fue introducida a Colombia a partir de la conquista y se caracteriza por producir frutos sin semilla debido a que es un triploide y no cuenta con polen viable; es un fruto de color verde, de forma semirredonda ovalada y peso de entre 50 y 100 gramos. Se han establecido en las diferentes regiones del país desde los cero metros hasta los 2.000 metros sobre el nivel del mar (msnm), sin embargo, el mejor comportamiento en la producción para la comercialización se logra máximo a alturas entre los 1.500 y los 1.600 msnm. Estos cítricos son cultivos de zonas subtropicales, que se han adaptado muy bien a las condiciones tropicales presentando un

buen comportamiento productivo. Departamento Administrativo Nacional de Estadística (DANE, 2015).

Figura 9. Limón Tahití

Nota. Esta figura muestra el fruto del limón persa también conocido como limón Tahití. *Fuente*. Imagen tomada de `http://limon-persico-2014.blogspot.com/2014/`

A nivel internacional, los principales mercados son EEUU, Alemania, Paises Bajos y Francia que concentran el 35,32 % de las importaciones mudiales, siendo EEUU el principal importador mundial de limón y el cual se abastece principalmente de países geográficamente cercanos como México, Guatemala y Colombia (Arias y Suarez, 2017). Existen unas características mínimas que el limón debe cumplir para ser clasificado como tipo exportación; según el Centro para la Promo-

ción de Importes de los Países Bajos (CBI, 2018) estas son:

- Intactos.
- Sin moretones y/o extensos cortes curados.
- Sano; excluyendo los productos afectados por la putrefacción o el deterioro que los haga no aptos para el consumo.
- Limpio, prácticamente libre de cualquier materia extraña visible.
- Prácticamente libre de plagas.
- Esfericidad.
- Libre de daños causados por plagas que afectan a la pulpa.
- Libre de signos de marchitamiento y deshidratación.
- Libre de daños causados por bajas temperaturas o heladas.
- Libre de humedad externa anormal.
- Libre de cualquier olor y/o sabor extraño.

Según la ficha técnica que presenta Segura (2008), las cosechas se deben realizar en horas frescas del día, a mano o utilizando guantes y tijeras. Se clasifican en 5 tipos diferentes de maduración siendo la 1 y 2 de exportación, también se separan por diámetro logrando cuantificar los limones en cada caja según las libras requeridas.

Figura 10. Niveles de maduración

Nota. Esta figura se muestran los niveles de maduración para el limón Tahití que proporciona (Segura, 2008). *Fuente*. Imagen tomada de `http://www.colexagro.com/fichas/Ficha-tecnica-LIMON.pdf`

Dentro de todos los estándares que se exigen a nivel nacional e internacional se decide agruparlos en tres clases que traten la mayor parte de las características visuales. La primer clase corresponde a los **Buenos**: limones con nivel de maduración 1 y 2, libres de daño en cáscara o la parte externa, sin manchas, ni signos de plaga, moretones, etc. La segunda clase corresponde a los **Manchados**: limones que poseen manchas amarillas y aquellos que tengan una maduración entre niveles 3, 4, 5 o superiores. Finalmente la clase de **Ácaros**: limones con daños en su cascara debido a plagas o daños que puedan afectar la calidad del fruto como moretones, marchitamiento, putrefacción entre otros. En la siguiente imagen se muestra de forma visual las clasificaciones descritas:

Figura 11. Tipos de limones

Nota. Clasificación de limones según las características mas relevantes en este proyecto.

2.9. Sistemas embebidos

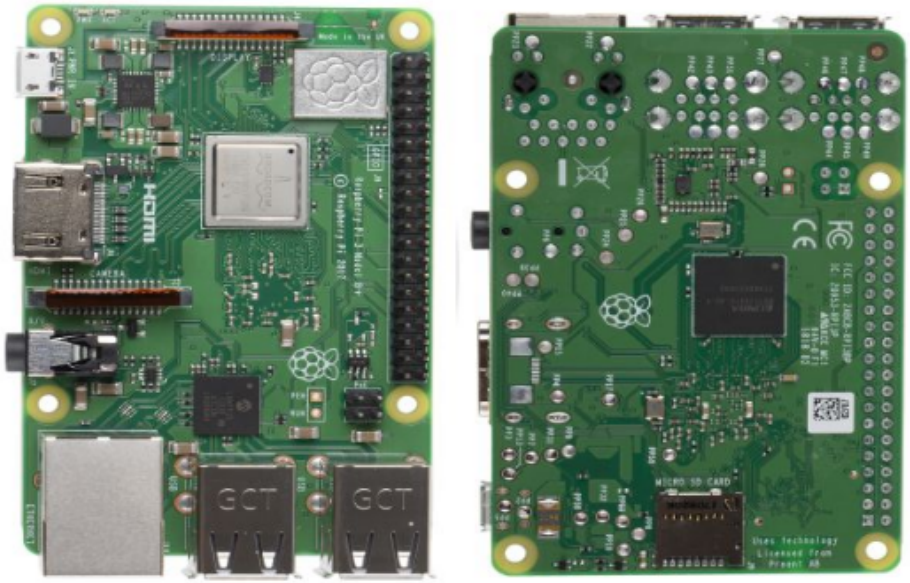
Se conoce como sistema embebido a un circuito electrónico que esta diseñado para cumplir una labor especifica cumpliendo una o algunas funciones en un producto y la mayoría de sus componentes se encuentra en una placa base. (Galeano, 2009).

En este proyecto se usaron dos sistemas embebidos, la Raspberry pi 3 b+ y MaixBit con la finalidad de hacer una comparación del desempeño de estos realizando la tarea de clasificación de imágenes en tiempo real.

2.9.1. Raspberry.

La Raspberry Pi es una computadora construida en una sola placa de circuito impreso, esta es compatible con varios componentes externos que se pueden conectar en sus puertos como lo son: cargador, pantalla, cámara, teclado, ratón (*mouse*), memoria de almacenamiento micro SD y pantalla LCD. Como toda computadora necesita de un sistema operativo para realizar el control de sus operaciones, siendo el más popular de estos Raspbian. Este se basa en Debian Linux y está diseñado a medida para este sistema. (Halfacree, 2018).

Figura 12. Raspberry pi 3 model B+



Nota. Placa de desarrollo Raspberry con vista frontal y posterior.

En concreto, el módulo Raspberry Pi 3 b+ es una versión avanzada de las Raspberry pi y tiene las siguientes especificaciones de hardware:

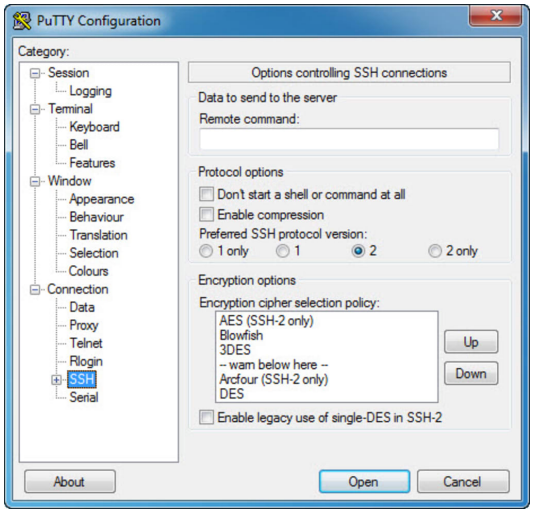
Tabla 4
Especificaciones técnicas Raspberry Pi.

CPU	ARM Cortex A53 de cuatro núcleos 64bit
Frecuencia	1.4GHz
Overclock	Si
RAM	1GB LPDDR2 SDRAM
WIFI	2.4 GHz and 5 GHz IEEE 802.11.b/g/n/ac wireless LAN
Bluetooth	4.2
Ethernet	Gigabit Ethernet over USB 2.0
Puertos	HDMI, 4 x USB 2.0, Micro USB, PoE, CSI (cámara), DSI (pantalla tácil), Micro SD conector 3.5 mm (auriculares)
Cámara	5MP, con foco ajustable
GPIO	40 pines
Tamaño	85mm x 53mm
Sistema Operativo	Linux / Unix
Tensión	5V
Corriente	2.5A
Precio	$ 39,45
Precio cámara	$ 8

2.9.1.1. Protocolo SSH. Este protocolo facilita la comunicación entre dos sistemas usando una arquitectura cliente/servidor y que permite a los usurarios conectarse a un servidor (*host*) remotamente. (Moore *et al.*, 2005). Se utiliza este tipo de conexión entre el computador y la Raspberry Pi para intercambiar información como: modelos entrenados, imágenes, código, etc, además de utilizar los periféricos del computador como ratón, teclado y pantalla.

Para esta conexión se utilizó el software de licencia libre PuTTy, que permite conectar a servidores remotos con solo iniciar sesión en ellos, también permite ejecutar comandos en el servidor una vez iniciada la sesión.

Figura 13. PuTTy



Nota. Interfaz gráfica del programa PuTTy. *Fuente*. Imagen tomada de: `https://putty.softonic.com/`

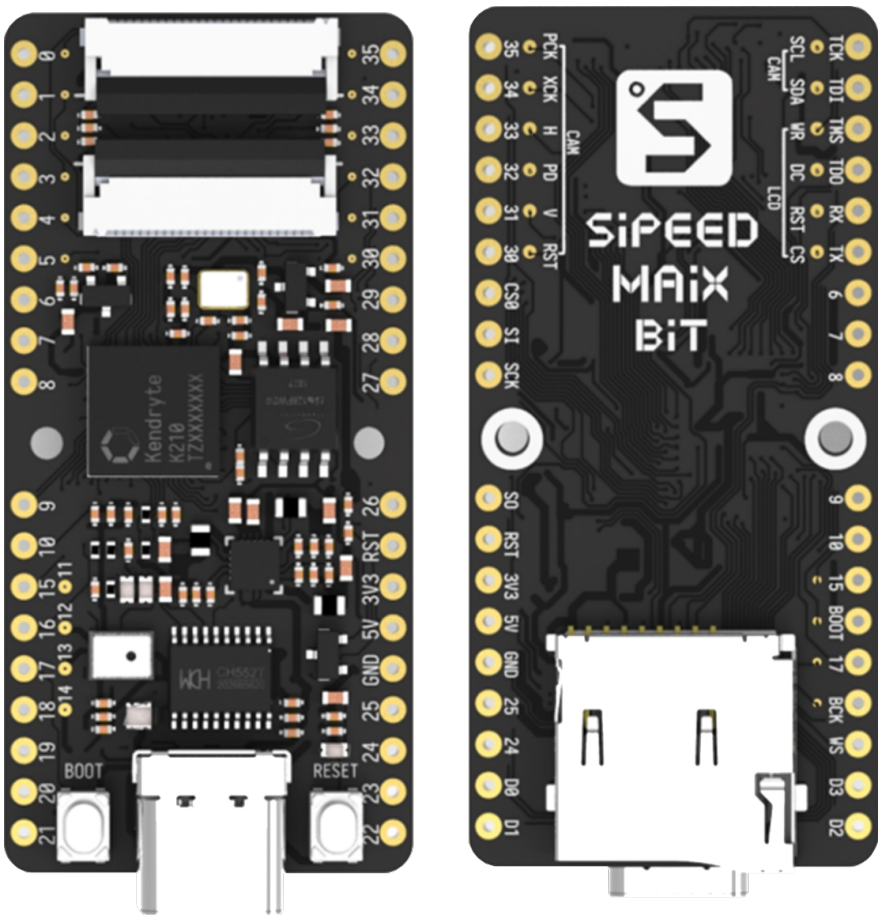
2.9.2. Maix Bit.

Esta es una tarjeta desarrollada por la compañía Seed-Studio cuyo objetivo principal es el desarrollo de hardware para internet de las cosas (IoT por sus siglas en ingles), como es de suponer; esta tarjeta es uno de estos dispositivos. Con un tamaño de reducido y un bajo consumo de potencia, esta tarjeta esta pensada para aplicaciones industriales como:

- Mantenimiento preventivo
- Detección de anomalías
- Visión artificial
- Robótica

- Reconocimiento de voz

Figura 14. Tarjeta MaixBit.



Nota. Placa de desarrollo Sipeed Maix Bit. *Fuente*. Imagen tomada de: `https://www.seeedstudio.com/Sipeed-MAix-BiT-for-RISC-V-AI-IoT-1-p-2873.html`

Las características técnicas de esta tarjeta se pueden ven en la tabla 5:

Tabla 5
Especificaciones técnicas Maix Bit.

CPU	RISC-V Dual Core 64bit
Frecuencia	400Mhz-500Mhz
Overclock	SI
RAM	8MB de alta velocidad SRAM
WIFI	NO
Bluetooth	NO
Ethernet	NO
Puertos	USB tipo C, 2x 24P FPC conector(Cámara y LCD) Micro SD
Cámara	OV2640 2MP con foco ajustable
GPIO	36 pines
Tamaño	53.3mm x 25.4mm
Software	Micro Python, FreRTOS y C
Micrófono	SI
Tensión	4.8V ~ 5.2V
Corriente	600mA
Precio	$ 12.9
Precio cámara	$ 7.6

En la parte de software posee soporte para FreeRTOS SDK, es compatible con MicroPython y con Arduino; en cuanto a *Deep-Learning* el acelerador K210 cuenta con un compilador:**nncase** (Kendryte, sf). También cuenta con soporte para tiny-yolo (Detector de objetos), MobileNet-V1 y la mayor parte de los modelos de TFLITE (TensorFlow Lite).

3. Desarrollo del proyecto

El entorno de desarrollo elegido para programar fue Google Colab + Google Drive, este entorno basado en la nube facilita compartir y editar *scripts* en tiempo real por diferentes miembros del equipo y también proporciona el hardware tanto en GPU o TPU totalmente gratis.

3.1. Base de datos

Las imágenes se tomaron con diferentes fondos (para que la red pueda identificar los limones en distintos lugares), cuadradas sin especificar un tamaño, en ambiente no controlado (iluminación natural y diferentes distancias) y en el espacio de color RGB. Se emplearon diferentes tipos de cámaras: Cámara de celular y sistema embebido de la tarjeta Maix Bit con la cámara OV2640. Para aumentar la cantidad de datos se tomaron fotos del mismo limón en diferentes fondos y se capturaban distintas zonas de este.

Figura 15. Tipos de limones

Nota. Clasificación de limones según las características mas relevantes en este proyecto.

El total de las imágenes obtenidas fue superior a 2500 con la posibilidad de aumentar este banco de imágenes realizando un pre-procesamiento o modificación de estas mediante rotación, reflexión, simetría axial o variaciones del brillo a la imagen. Esta base de datos se separa en dos grupos que corresponden al grupo de entrenamiento, que tiene aproximadamente el 80 % de las imágenes totales, y al grupo de validación, que tiene un 20 % del total y se subdivide en: Validación 15 % y Pruebas 5 %. Además cada grupo debe tener clasificadas y separadas sus 3 clases de limones con la misma cantidad de imágenes ya que se trabaja con una base de datos balanceada.

Tabla 6
Banco de imágenes completo.

Tipo de imagen	**Datos totales**
Ácaros	642
Buenos	1295
Manchados	758
Total	2695

Tabla 7
Banco de imágenes balanceado

Tipo de imagen	**Datos Train**	**Datos Validacion**	**Datos Prueba**	**Datos totales**
Acaros	513	96	32	641
Buenos	513	96	32	641
Manchados	513	96	32	641

Ya teniendo todas la imágenes clasificadas correctamente se remodeló su matriz RBG para que cada píxel de la imagen sea una entrada de la red neuronal convolución (CNN), primero se ponen todas las imágenes del mismo tamaño (224x224x3), luego se convierte en un vector y se junta con otros vectores, entre 32 a 512 vectores, para formar pequeñas matrices llamadas *mini-batch* o *batches*, esto mejora la optimización del entrenamiento. Por ultimo se normalizan sus valores lo que ayuda a un mejor funcionamiento en su función de costo.

Todo lo anterior se puede hacer con la librería numpy de python, para obtener los array **.np**, pero ya existen funciones en librerías dedicadas a redes neuronales que hacen esto, para lo cual se utiliza la función: **image_dataset_from_directory** y la clase: **ImageDataGenerator** de la biblioteca Keras.

Figura 16. Función ImageDataGenerator.

```
tf.keras.preprocessing.image.ImageDataGenerator(
    featurewise_center=False, samplewise_center=False,
    featurewise_std_normalization=False, samplewise_std_normalization=False,
    zca_whitening=False, zca_epsilon=1e-06, rotation_range=0, width_shift_range=0.0,
    height_shift_range=0.0, brightness_range=None, shear_range=0.0, zoom_range=0.0,
    channel_shift_range=0.0, fill_mode='nearest', cval=0.0, horizontal_flip=False,
    vertical_flip=False, rescale=None, preprocessing_function=None,
    data_format=None, validation_split=0.0, dtype=None
)
```

Nota. Código que genera lotes de datos de tensores de las imágenes con aumento de datos en tiempo real.

Figura 17. Función image_dataset_from_directory.

```
tf.keras.preprocessing.image_dataset_from_directory(
    directory, labels='inferred', label_mode='int', class_names=None,
    color_mode='rgb', batch_size=32, image_size=(256, 256), shuffle=True, seed=None,
    validation_split=None, subset=None, interpolation='bilinear', follow_links=False
)
```

Nota. Genera tf.data.Dataset desde archivos de imagen en un directorio

Uniendo estas dos funciones anteriores se puede crear un conjunto de datos para ingresar a la red neuronal; el código con el que se genera es el siguiente:

```
TRAINING_DIR = "/DatosPrueba/Datos_Train" #Data_train


training_datagen = ImageDataGenerator(
    rescale = 1./255,
    rotation_range=40,
```

```
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest')

train_generator = training_datagen.flow_from_directory(
    TRAINING_DIR,
    target_size=(224,224),
    class_mode='categorical',
    shuffle=True,
    batch_size=32
)
```

3.2. Arquitectura de red neuronal

Para la arquitectura de red existen múltiples estructuras ya creadas y entrenadas que tienen resultados sobresalientes como por ejemplo, VGG16, VGG19, ResNet, DenseNet, Xception y más. Debido a que nuestra aplicación es móvil se decide hacer *Transfer-Learning* con la red **MobileNet-V1**, esta ya ha sido entrenada para clasificación de imágenes con las bases de datos de ImageNet. TensorFlow Lite (una versión de TensorFlow diseñada para aplicaciones de IoT y para dispositivos móviles) tiene a disposición modelos ya entrenados para los diferente multiplicadores de ancho y resolución (α y ρ).

Para hacer la adecuación de la red esta se invoca desde la librerías de Tensorflow con sus

pesos ya entrenados con la base de datos ImageNet y se procede a modificar sus etapas de salida para crear un modelo capaz de reconocer solo 3 clases.

```
def modelo():
  base_model = tf.keras.applications.MobileNet(input_shape=(224,224,3),alpha
   =1.0,include_top=False, weights='imagenet')
  base_model.trainable = False
  model = tf.keras.Sequential([
    base_model,
    tf.keras.layers.Conv2D(32, 3, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(3, activation='softmax')
  ])
  return model
```

El anterior código es una función que invoca el modelo MobileNet sin incluir la capa de salida de este (include_top); con los pesos ya entrenados sacados desde ImageNet y con un tamaño de la imagen especificado. Luego se deshabilita su capacidad de entrenamiento para las capas principales, ya que no se quiere que se vuelvan a entrenar, y se procede a añadir las capas que no fueron incluidas en la invocación del modelo, esta son las capas finales donde se configura cuantas clases se tiene para su clasificación y serán están las capas que se entrenen. Luego se configura el modelo para el entrenamiento con el nuevo conjunto de datos utilizando funciones de Tensorflow con Keras.

Figura 18. Función compile de TensorFlow

```
compile(
    optimizer='rmsprop', loss=None, metrics=None, loss_weights=None,
    weighted_metrics=None, run_eagerly=None, **kwargs
)
```

Nota. Configura el modelo para su entrenamiento.

Figura 19. Función fit de TensorFlow.

```
fit(
    x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None,
    validation_split=0.0, validation_data=None, shuffle=True, class_weight=None,
    sample_weight=None, initial_epoch=0, steps_per_epoch=None,
    validation_steps=None, validation_batch_size=None, validation_freq=1,
    max_queue_size=10, workers=1, use_multiprocessing=False
)
```

Nota. Entrena el modelo para un número fijo de épocas (iteraciones en un conjunto de datos).

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

# valores para pasos en epocas
steps_per_epoch = np.ceil(train_generator.samples/train_generator.batch_size)
val_steps_per_epoch = np.ceil(validation_generator.samples/
    validation_generator.batch_size)
```

```
7
8  history=model.fit(
9      generator=train_generator,
10     steps_per_epoch=steps_per_epoch,
11     epochs=30,
12     verbose = 1,
13     validation_data = validation_generator,
14     validation_steps=16)
```

Como se vio en las figuras 18 y 19 existen varios parámetros que se pueden modificar en la configuración de las funciones (compile) y (fit) del modelo, para nuestro proyecto los párametros de interes son: En la configuración se elige tipo de descenso del gradiente y función de costo. En el entrenamiento es importante el número de épocas y el paso dentro de las épocas, se puede elegir este parámetro dependiendo del tamaño y cantidad en *batch* del conjunto de datos, esto ayudará a un mejor entrenamiento, también se debe tener los datos de entrenamiento y validación (train_generator, validation_generator).

```
1  converter = lite.TFLiteConverter.from_keras_model(model)
2  tflite_model = converter.convert()
3
4  open("modelo\_limones.tflite", 'wb').write(tflite_model)
```

Luego del entrenamiento se procede a guardar el modelo en formato .tflite, y .kmodel debido a que se implementa en la Raspberry pi y Maix Bit, y se buscan modelos optimizados para sistemas embebidos, o cualquier otro formato deseado. Para convertir y guardar el modelo en tflite

se dispone de funciones de la librería TensorFlow **tf.lite.TFLiteConverter** como se muestra en el código.

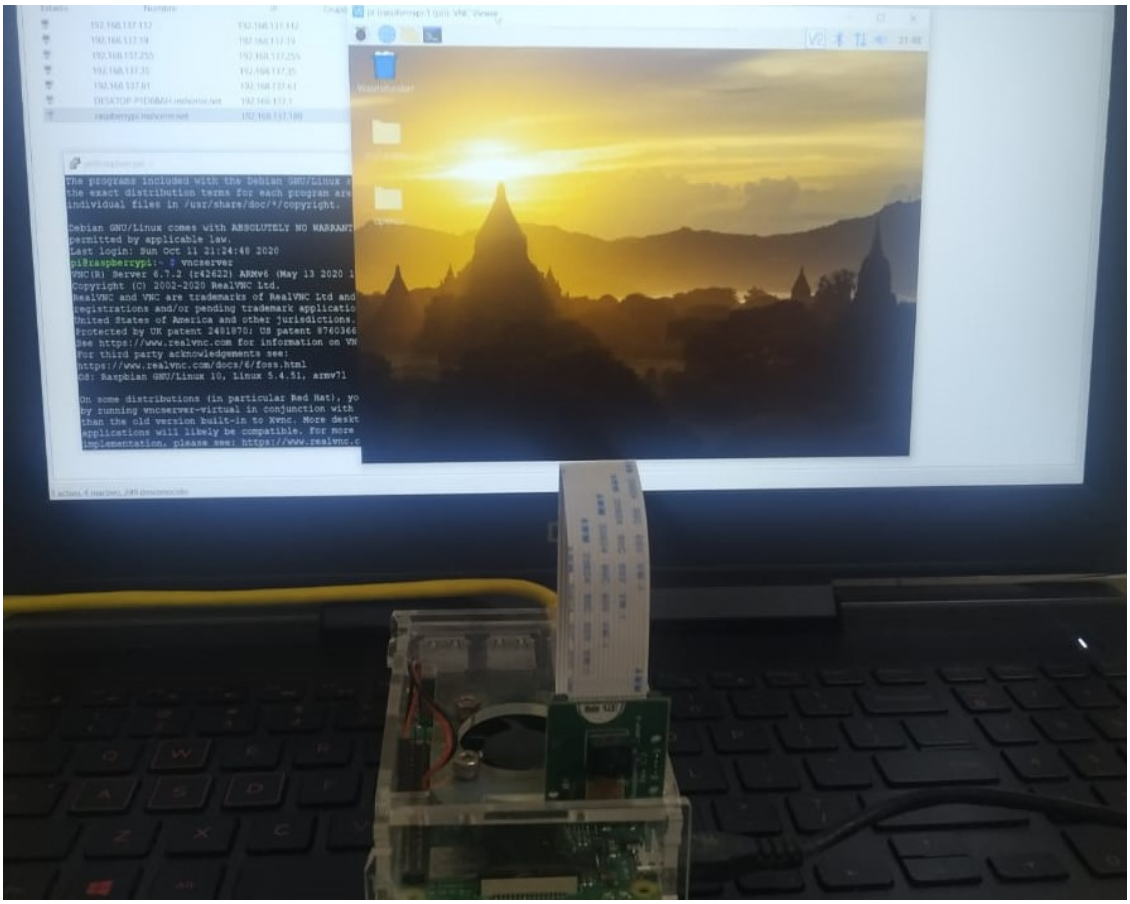
En la práctica se hizo uso del *framework* aXeleRate diseñado por Maslov el cual esta basado en Keras y permite realizar el entrenamiento de varios modelos de RNC conocidos como: MobileNet, SqueezeNet, NASNetMobile, DenseNet121, ResNet50 entre otros. La principal ventaja que tiene este *framework* (y por la cual se hizo uso de él) es que tiene integrado el compilador *nncase* (Kendryte, sf) el cual es fundamental para la generación del .kmodel; de igual forma este *framework* es capaz también de generar el .tflite que se usa en la Raspberry por lo que nos permite realizar una mejor comparación en el desempeño de los dos dispositivos al tener entrenado el modelo con exactamente el mismo *framework*.

3.3. Implementación del modelo

Para la implementación primero se configura el sistema embebido donde se va hacer inferencia del Tensorflow Lite, en caso de la Raspberry Pi, el término " inferencia se refiere al proceso de ejecutar un modelo de TensorFlow Lite en el dispositivo para hacer predicciones basadas en datos de entrada" (Tensorflow, sf). Se configura la tarjeta Raspberry pi grabando el sistema operativo Raspbian en la tarjeta SD utilizando cualquier software libre como Win32 Disk Imager, balenaEtcher, rufus entre otros, se habilita el acceso remoto a la Raspberry con SSH, para evitar conectar teclado y monitor, para esto se crear un archivo ssh en el directorio de arranque de la tarjeta SD, luego se configura su acceso a Windows con el programa PuTTY, al cual se le agrega solo la dirección IP por donde se esta conectando la Raspberry pi y se pulsa el botón "abrir" , esto nos abre la terminal para trabajar con Raspberry, si se desea ver de forma grafica su pantalla se puede utilizar

VNC viewer, es un programa de software libre basado en una estructura cliente-servidor que permite observar las acciones del ordenador servidor remotamente a través de un ordenador cliente. Al final de este proceso podremos conectar la raspberry pi a nuestro computador a través de solo un cable Ethernet.

Figura 20. Conexión de la Raspberry pi al computador.

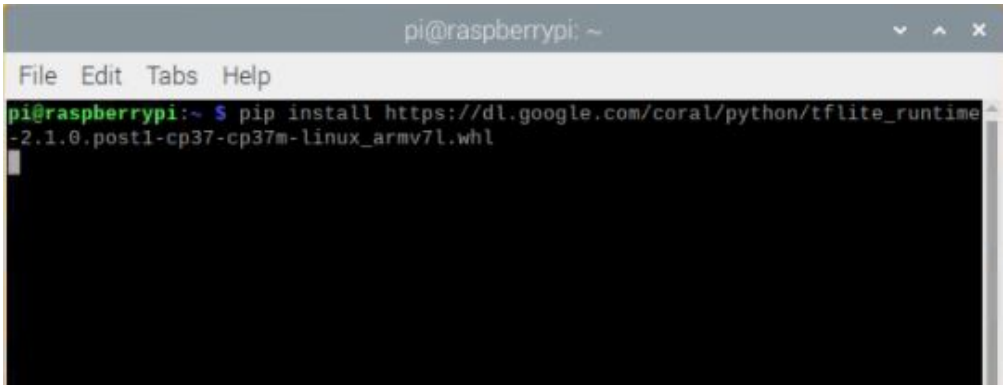


Nota. Raspberry pi 3 b+ conectada al computador por medio de cable Ethernet mediante el protocolo SSH.

Luego se instala la API de Tensorflow lite con python ya que es ideal para dispositivos integrados basados en Linux. Para ejecutar rápidamente modelos de TensorFlow Lite con Python, pue-

de instalar solo el intérprete de TensorFlow Lite, este paquete posee solo la clase **tf.lite.Interpreter** el cual posee el código necesario para correr modelos tflite. Para instalar, se ejecuta *pip3 install* con la URL apropiada que se encuentra fácilmente en la pagina guía de TensorFlow Lite: inicio rápido con python.

Figura 21. Instalación de TensorFlow Lite.



Nota. Instalación del interprete de TensorFlow Lite en la terminal de Raspbian mediante pip install y URL proporcionado por TensorFlow.

La siguiente imagen muestra el código de como usar el interprete de tflite con python para cargar el modelo y ejecutar las inferencias con los datos de entrada.

Figura 22. Código para ejecutar el interprete tf.lite.

```python
import numpy as np
import tensorflow as tf

# Load the TFLite model and allocate tensors.
interpreter = tf.lite.Interpreter(model_path="converted_model.tflite")
interpreter.allocate_tensors()

# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

# Test the model on random input data.
input_shape = input_details[0]['shape']
input_data = np.array(np.random.random_sample(input_shape), dtype=np.float32)
interpreter.set_tensor(input_details[0]['index'], input_data)

interpreter.invoke()

# The function `get_tensor()` returns a copy of the tensor data.
# Use `tensor()` in order to get a pointer to the tensor.
output_data = interpreter.get_tensor(output_details[0]['index'])
print(output_data)
```

Nota. Código necesario para ejecutar el interprete de modelos tflite.

Además del código anterior se debe programar también la captura de las imágenes utilizando la pi-Camera V2, para esto se utiliza la librería cv2 como también funciones de la biblioteca piCamera (continuous_capture), luego de capturar los datos y pasarlos por el interprete de TensorFlow Lite se debe mostrarlos en pantalla junto a otros posibles datos de interés como: tiempo que se demorar procesando cada imagen, predicción del limón y su porcentaje. A continuación se observa el código para capturar las imágenes, procesarlas para entrar a la red, tomar el tiempo de clasificación y mostrar los resultados en la imagen. El código completo para correr la red neuronal será la combinación en un .py del interprete de tflite y el código de captura de imágenes, el cual se puede ver completamente en el siguiente repositorio de GitHub:

https://github.com/Imbana/ClasificacionLimonesCNN.git

```
camera = PiCamera()
camera.resolution = (640, 480)
camera.framerate = 30
rawCapture = PiRGBArray(camera, size=(640, 480))
stream = io.BytesIO()
for frame in camera.capture_continuous(rawCapture, format="bgr",
    use_video_port=True):
    image = frame.array
    frame=cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    #procesa la imagen para entrar a la red
    Imagen_normed = frame/ 255
    Imagen_show = cv2.resize(image, (600, 600))
    Imagen_resized = cv2.resize(Imagen_normed, (224, 224))
    Imagen_espnaded=np.expand_dims(Imagen_resized, axis=0)

    #clasificar la imagen, llama a una funcion creada del interprete de tflite
    y calcula el tiempo
    start_time = time.time()
    results = clasificar_imagen(interpreter, Imagen_espnaded)
    elapsed_ms = round((time.time() - start_time) * 1000,2)
    label_id, prob = results[0]

    #Agregar texto a la imagen con los resultados obtenidos
```

```
    print(labels[label_id]+"    "+str(elapsed_ms))
    font = cv2.FONT_HERSHEY_SIMPLEX
    color=(14,129,60)
    images = cv2.putText(Imagen_show,str(elapsed_ms)+"ms  "+labels[label_id]+"
    "+str(prob) , (00,100), font, 1,color,2, cv2.LINE_AA)

    # Mostrar la imagen
    cv2.imshow("Frame", images)

    # Salir si se preciona "q"
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

    # Borra la imagen y comenzar de cero
    rawCapture.truncate(0)

cv2.destroyAllWindows()
```

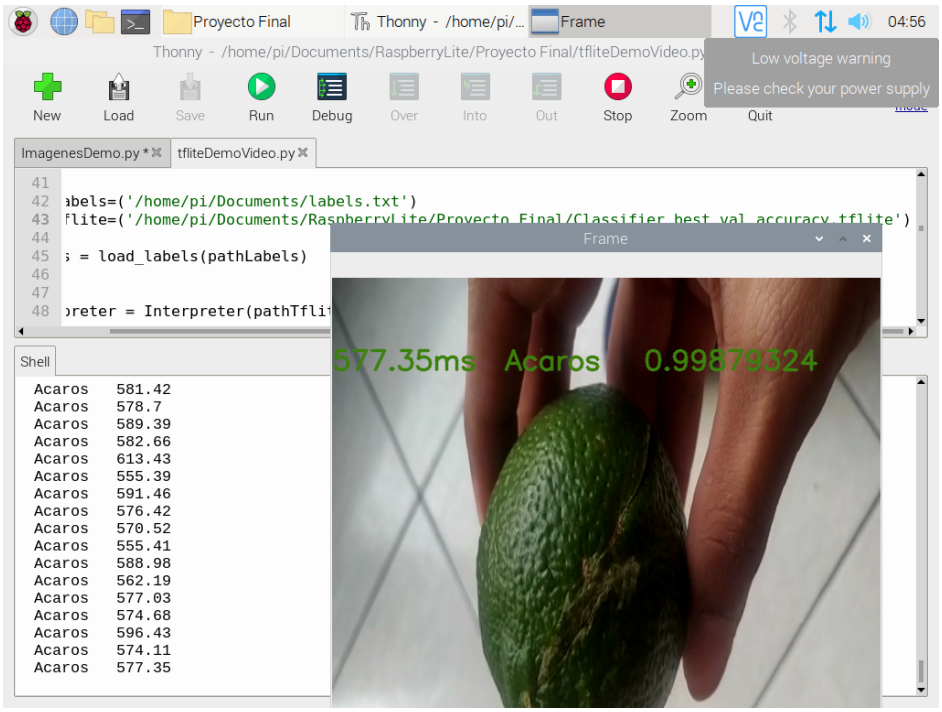
Finalmente para ejecutar el .py donde se hará la clasificación de las imágenes, el cual estará conformado por los códigos descritos anteriormente, se necesita tanto el modelo entrenado y las etiquetas en formato texto. Se puede ejecutar desde la terminal de la siguiente forma, donde las rutas del *model* y *labels* corresponden a donde se encuentran estos dos documentos o directamente se puede poner las rutas en en el .py, para el proyecto se utilizó la terminal.

```
python3 tensorflowDemo.py --model /home/pi/Documents/ultimo.tflite --labels
```

```
/home/pi/Documents/labels.txt
```

En la figura 23 se observa un ejemplo de la clasificación por medio de un vídeo, el porcentaje superior al 90% de precisión y en la terminal se observa el tiempo que va desde unos 500 a 600 en milisegundos, que cada imagen tarda en ser clasificada .

Figura 23. Clasificación utilizando la Raspberry.



*Nota.*Clasificación usando la Raspberry pi 3 b+ con cámara.

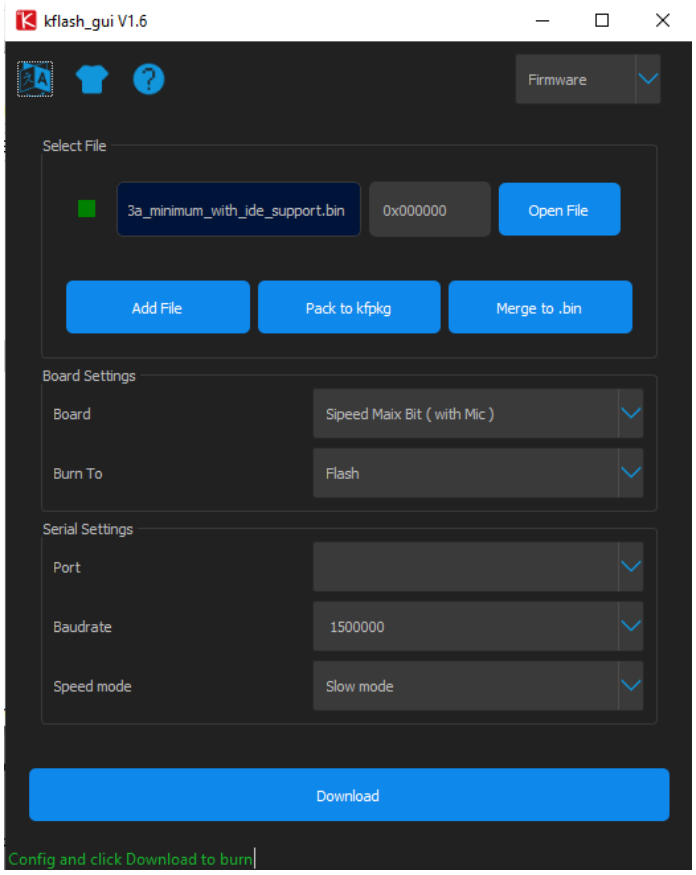
En el caso de la tarjeta **Maix Bit** la implementación se divide en las siguientes actividades:

- Carga del firmware
- Carga del modelo

- Descripción y carga del código
- Prueba y validación

Inicialmente se debe realizar la instalación o **carga del firmware** controlador de la tarjeta, el cual se puede descargar desde el repositorio oficial de Sipeed en GitHub; una vez se tiene la versión deseada del firmware es necesario cargarlo dentro de la tarjeta, esto se hace por medio de la herramienta kflash; tal como se muestra a continuación:

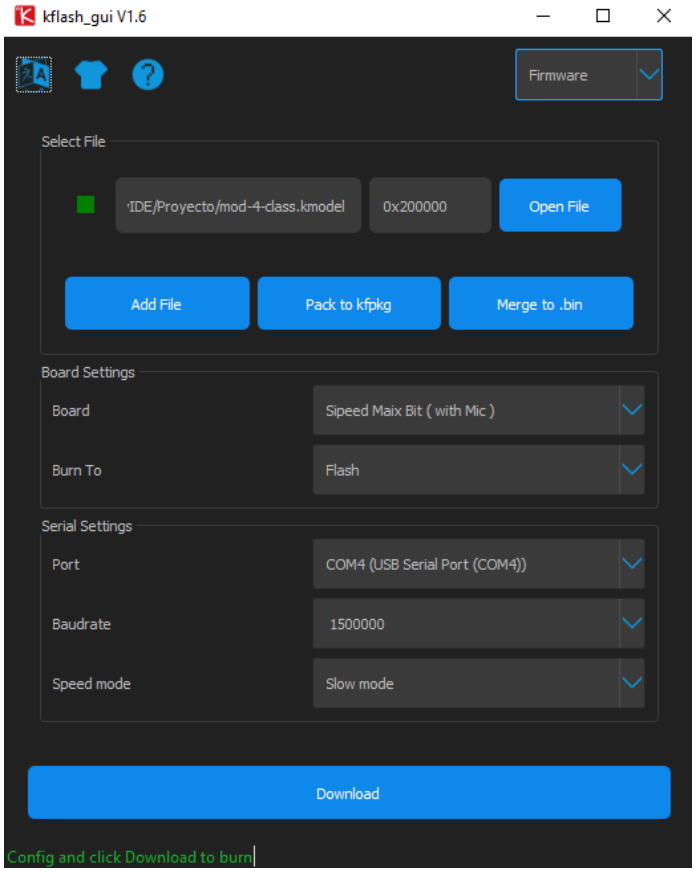
Figura 24. Herramienta Kflash para carga del firmware.



Nota. Configuración del Kflash para la carga del firmware en la tarjeta Maix Bit.

Para la **carga del modelo** se debe tener una tarjeta SD a la cual previamente se le debe copiar el modelo o en caso de no contar con una de estas memorias, se puede instalar mediante el uso de kflash teniendo en cuenta que el modelo debe tener como máximo un peso de 2Mbytes y se debe asignar en otra dirección de memoria RAM; tal que:

Figura 25. Herramienta Kflash para carga del modelo.



Nota. Configuración del Kflash para la carga del .kmodel.

Posteriormente debe realizarse la **descripción del código** la cual consta de 5 etapas; en la primera se realiza el importe de las librerías necesarias para el manejo de la pantalla, las imá-

genes, las redes neuronales y la cámara. En la segunda se describe la configuración inicial de la cámara y la pantalla; allí se indica cual es el tamaño de las imágenes a capturar y el espacio de color a usar. En la tercera se describen las variables correspondientes al modelo, se indican las etiquetas del mismo, cual es la ubicación en memoria del modelo así como también las dimensiones de la capa de salida del mismo. La cuarta es la ejecución del modelo, esto se hace dentro de un bucle *while* capturando una imagen y pasándola a la red, de este modo la red extrae en una lista las probabilidades correspondientes a cada una de las clases. La ultima etapa consiste en dibujar en la pantalla las predicciones del modelo, esto se hace escogiendo la clase con la mayor probabilidad y poniendo su etiqueta y probabilidad en la pantalla. El código usado se puede observar a continuación y tambien se puede acceder a él en el siguiente repositorio de GitHub: `https://github.com/Imbana/ClasificacionLimonesCNN.git`:

```
# MicroPython v0.5.0-29-g97fad3a on 2020-03-13; Sipeed_M1 with kendryte-k210

# Importe de librerias
import sensor, image, lcd, time, utime
import KPU as kpu

# Configuracion inicial de la pantalla LCD y la camara OV2640
lcd.init() # Inicializa la pantalla
sensor.reset() # Inicializa la camara
sensor.set_pixformat(sensor.RGB565) # Define el formato de color de la imagen
sensor.set_framesize(sensor.QVGA) # Establece la captura de imagen como QVGA
    (320x240)
```

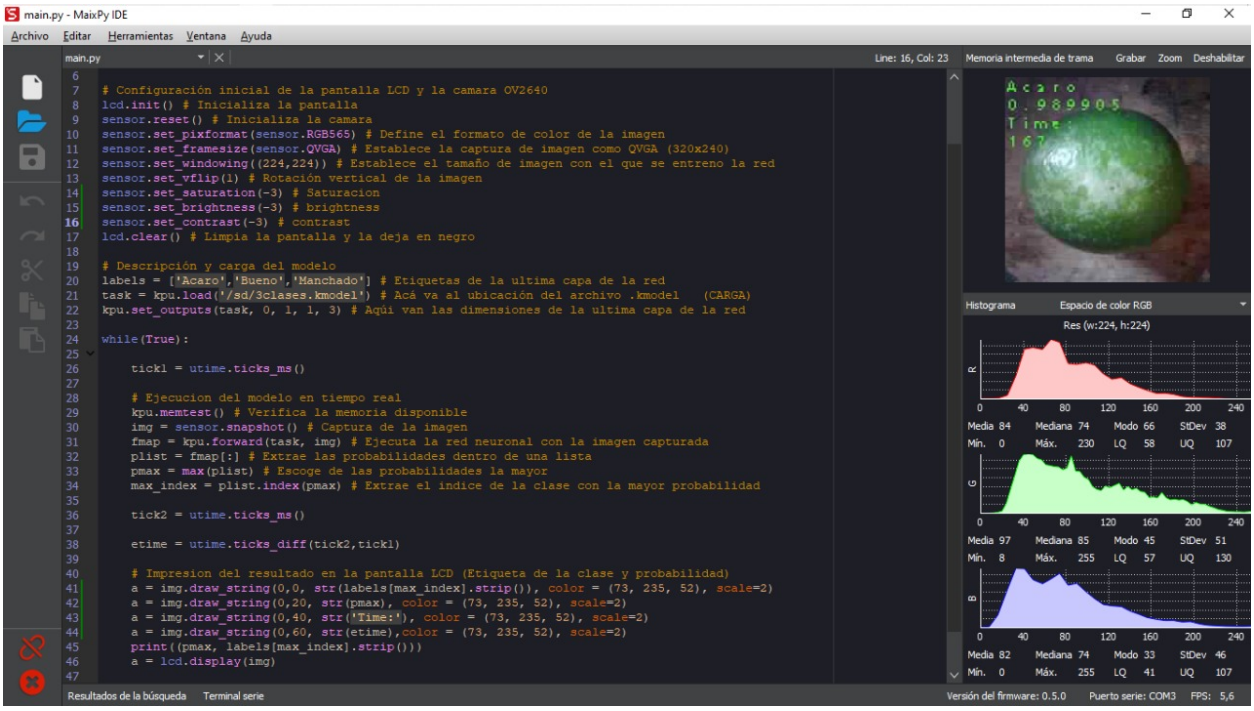
```
sensor.set_windowing((224,224)) # Establece el tamano de imagen con el que se entreno la red
sensor.set_vflip(1) # Rotacion vertical de la imagen
sensor.set_saturation(-3) # Saturacion
sensor.set_brightness(-3) # brightness
sensor.set_contrast(-3) # contrast
lcd.clear() # Limpia la pantalla y la deja en negro

# Descripcion y carga del modelo
labels = ['Acaro','Bueno','Manchado'] # Etiquetas de la ultima capa de la red
task = kpu.load('/sd/3clases.kmodel') # Aca va al ubicacion del archivo .kmodel (CARGA)
kpu.set_outputs(task, 0, 1, 1, 3) # Aqui van las dimensiones de la ultima capa de la red

while(True):

    tick1 = utime.ticks_ms()

    # Ejecucion del modelo en tiempo real
    kpu.memtest() # Verifica la memoria disponible
    img = sensor.snapshot() # Captura de la imagen
    fmap = kpu.forward(task, img) # Ejecuta la red neuronal con la imagen capturada
    plist = fmap[:] # Extrae las probabilidades dentro de una lista
```

```
    pmax = max(plist) # Escoge de las probabilidades la mayor
    max_index = plist.index(pmax) # Extrae el indice de la clase con la mayor probabilidad

    tick2 = utime.ticks_ms()

    etime = utime.ticks_diff(tick2,tick1)

    # Impresion del resultado en la pantalla LCD (Etiqueta de la clase y probabilidad)
    a = img.draw_string(0,0, str(labels[max_index].strip()), color = (73, 235, 52), scale=2)
    a = img.draw_string(0,20, str(pmax), color = (73, 235, 52), scale=2)
    a = img.draw_string(0,40, str('Time:'), color = (73, 235, 52), scale=2)
    a = img.draw_string(0,60, str(etime),color = (73, 235, 52), scale=2)
    print((pmax, labels[max_index].strip()))
    a = lcd.display(img)

a = kpu.deinit(task)
```

Finalmente la **validación** se realiza ejecutando el código mediante el uso de la MaixPy IDE y un cable USB tipo C conectado a la tarjeta; cuando el sistema este conectado y listo para usar se ponen limones frente a la cámara y se observa la predicción en la pantalla conectada a la tarjeta o en la proporcionada por la IDE; del mismo modo también se muestra el tiempo de ejecución del

bucle. En este punto se resalta que cuando el código se ejecuta con la IDE el tiempo de respuesta aumenta a diferencia de cuando este se ejecuta desde la tarjeta

Figura 26. Clasificación usando la tarjeta Maix Bit.



Nota.Clasificación usando la Maix Bit con cámara y mostrando el código utilizado

4. Resultados

En este capítulo se presentan y analizan los resultados de las 4 versiones de la red neuronal *MobileNet-V1* usadas en la fase de entrenamiento (estas corresponden a los multiplicadores de ancho (α) 1, 0.75, 0.5 y 0.25), inicialmente se muestran los gráficos correspondientes a la exactitud y la perdida que se logró con el entrenamiento; adicionalmente se pone en una matriz de confusión la efectividad real del modelo con las imágenes de prueba y finalmente se realiza una comparación en cuanto al desempeño del modelo en los dos sistemas embebidos utilizados: Maix Bit y Raspberry Pi 3 b+.

4.1. Entrenamiento

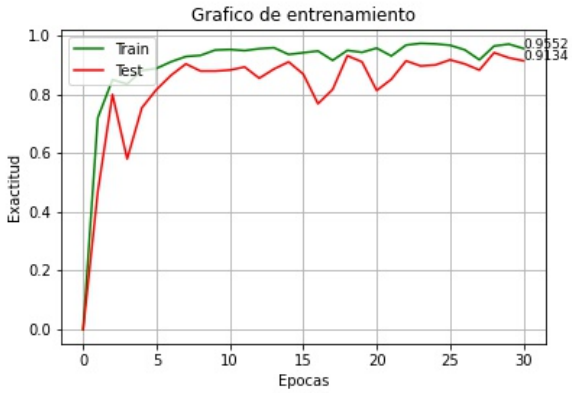
Cada modelo se entrenó con el mismo conjunto de datos por un total de 30 épocas tomando en cada época información de su exactitud en la predicción y el error en la función de coste como se observa en las figuras 27 y 28; de estas figuras se resalta que el modelo con el mejor desempeño es el que tiene todas sus capas *MobileNet 1_0* esto debido a que logra un 99 % de exactitud con el entrenamiento y un 92 % con la validación. La tasa de aprendizaje usada fue de 0,001 y se puede observar que a mayor número de épocas mejor es su rendimiento pero puede llegar a la situación en la que la red deja de aprender: sobre-juste (*overfitting*).

Las imágenes se procesaron con *batches* de 32, un tamaño de 224x224x3 (Imágenes RGB).

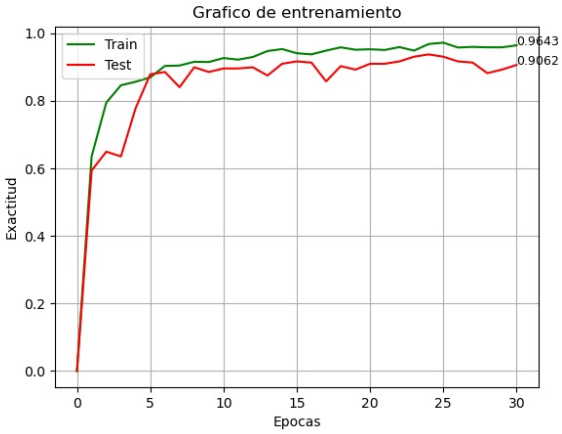
Figura 27. Gráficas de entrenamiento según su predicción.

(a) MobilNet 1_0

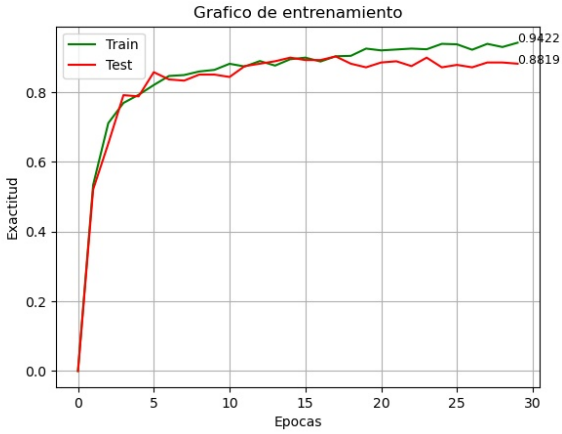
(b) MobilNet 0_75



(c) MobilNet 0_50



(d) MobilNet 0_25



Nota. Información de exactitud para modelos MobileNet V1 con diferente multiplicador de ancho (*alpha*).

Figura 28. Gráficas de perdida según la función de coste.

(a) MobilNet 1_0

(b) MobilNet 0_75

(c) MobilNet 0_50

(d) MobilNet 0_25

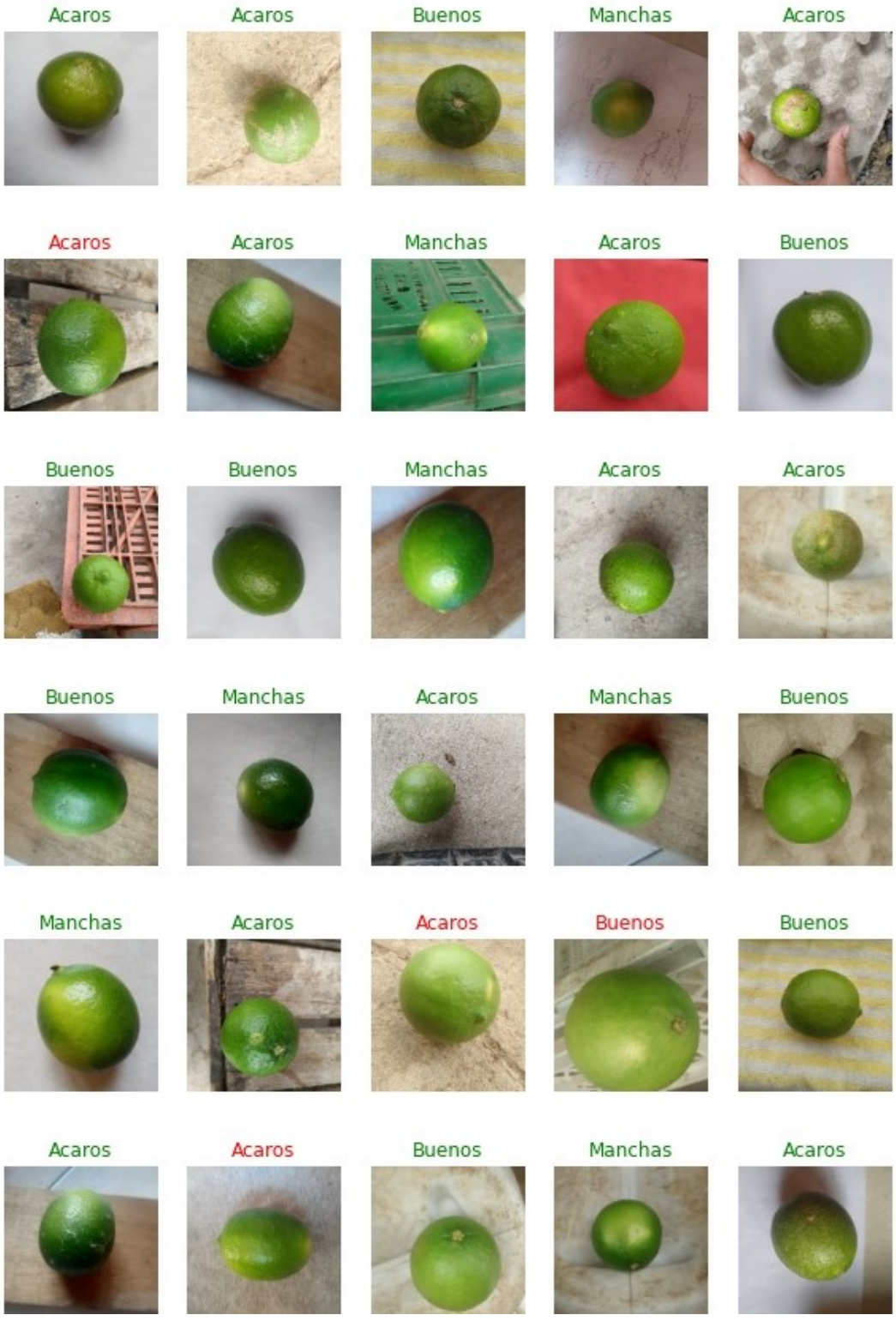
Nota. Información del error en la función de coste para modelos MobileNet V1 con diferente multiplicador de ancho (*alpha*).

Tabla 8
Rendimiento de los modelos.

Modelo	**Tipo**	**Exactitud**	**Perdida**
MobilNet 1_0	Train	99.03 %	0.033
	Validacion	92.36 %	0.14
MobilNet 0_75	Train	95.52 %	0.144
	Validacion	91.34 %	0.410
MobilNet 0_50	Train	96.43 %	0.0949
	Validacion	90.62 %	0.0151
MobilNet 0_25	Train	94.22 %	0.1733
	Validacion	88.19 %	0.1678

En la siguiente gráfica (29) se observa la clasificación de un conjunto de imágenes con el modelo MobilNet 1_0, el cual tiene la mejor tasa de exactitud con un 92 % de aciertos, poniendo en color verde los de correcta clasificación y roja las incorrectas, se observa que existe una clara diferenciación de las características del limón pero con cierto margen de error.

Figura 29. Clasificación gráfica de limones Tahití.



Nota. Ejemplo de clasificación de limones utilizando el conjunto de datos de validación.

4.2. Matriz de confusión

Para este apartado se utiliza las imágenes de la sección de testeo, un 5 % del total de las imágenes denominado *dataset* de **prueba**, las cuales no se han utilizado hasta el momento, es un conjunto de datos de 96 imágenes para realizar las pruebas de campo y ver que tan bien responde nuestro modelo en la clasificación e identificación de características con respecto a lo esperado.

Tabla 9
Matriz de confusión con los datos de prueba.

		Valor Predicho		
		Ácaros	**Buenos**	**Manchas**
Valor Real	**Ácaros**	32	0	0
	Buenos	1	31	0
	Manchas	5	0	27

En la anterior matriz de confusión de observa que existe mayor exactitud en las predicciones con respecto a la clase **Ácaros**, ya el modelo logra clasificarlos todos correctamente. Los errores de las otras categorías se inclinan hacia la clase **Ácaros**, donde el mayor error ocurre con la clase **Manchas** esto se observa en la situación en la que existan limones con defectos de **Manchas** y **Ácaros** lo que hace que el modelo le de una mayor importancia al defecto que resalte más, en este caso es el de **Ácaros**. Finalmente se observa que la clase de **Buenos** esta bien definida ya que presenta muy poco error y, además los errores de las otras categorías no cayeron o se confundieron por buenos, esto nos proporciona un gran modelo para la clasificación de los limones por ejemplo en el campo de la exportación donde se deben cumplir los estos parámetros de forma estricta.

4.3. Validación en los sistemas embebidos

Comparación de los dos sistemas embebidos:

Tabla 10
Especificaciones principales de los sistemas embebidos.

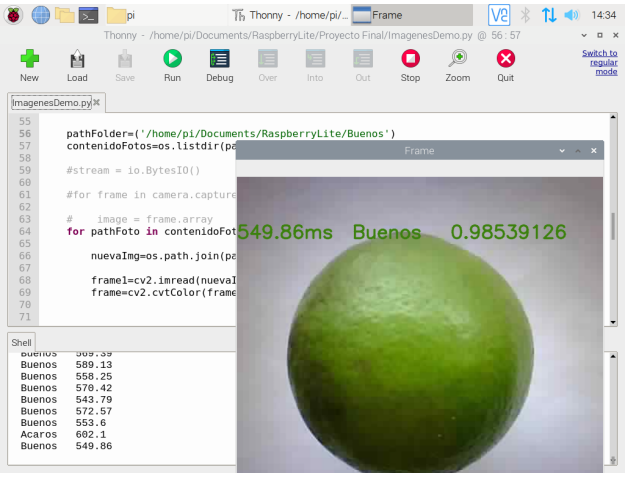
Parámetros	**Maix BiT**	**Raspberry pi 3 b+**
Alimentación	4.8V a 5.2 V, 600mA	5V, 2.5 A
Procesador	Kendryte K210, 64bit, 400MHz	Broadcom, 64 bits, 1,4 GHz
COM	USB	USB, Ethernet,
Formato del modelo	.kmodel	.tflite
RAM	8 MB	1 GB
Precio	$ 12.9	$ 39,45
Precio Cámara	$ 7.6	$ 8
Dimensiones [mm]	54 x 26 x 13	82 x 56 x 19,5
Firmware	MaixPy	Linux
Lenguaje	MicroPython, C	Python, Java, otros

4.3.1. Raspberry.

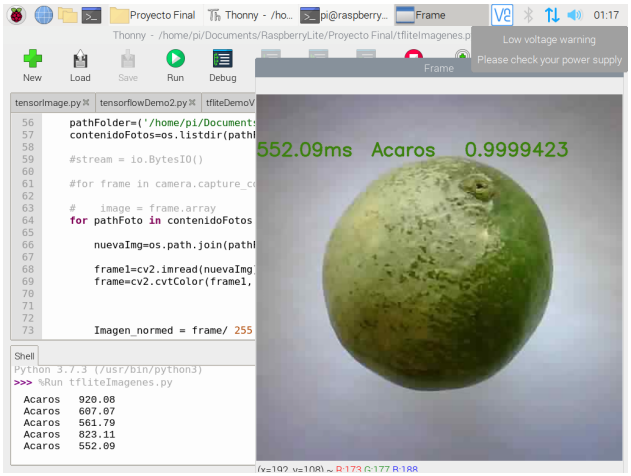
Implementando el modelo en la Raspberry se puede ver que el sistema funciona correctamente en su clasificación, sin embargo, también se puede añadir una mejora que consiste en proporcionar un vídeo en el cual se captura la mayor parte de la corteza del limón lo cual aumenta la exactitud de la predicción. Esto se observa en la terminal que se muestra en la imagen 30 donde se nota que unas pocas predicciones eran incorrectas pero su mayoría acertaba.

Figura 30. Clasificaciones en tiempo real usanto la Raspberry Pi

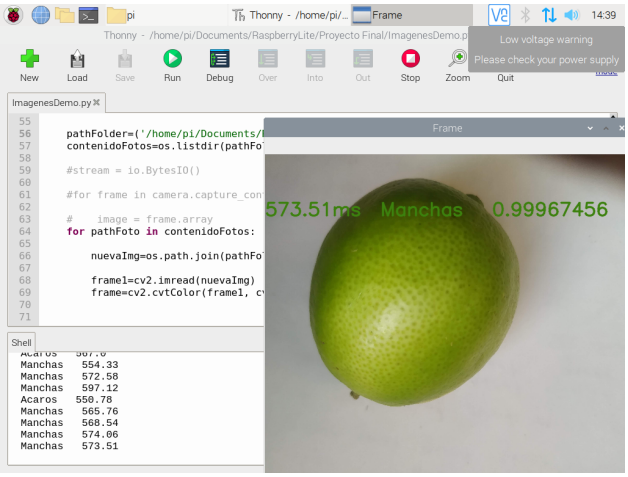
(a) Ácaros



(b) Buenos



(c) Manchados



Nota. Las etiquetas en verde muestran las clasificaciones de limones con tiempo [ms], el tipo de limón y el porcentaje de precisión.

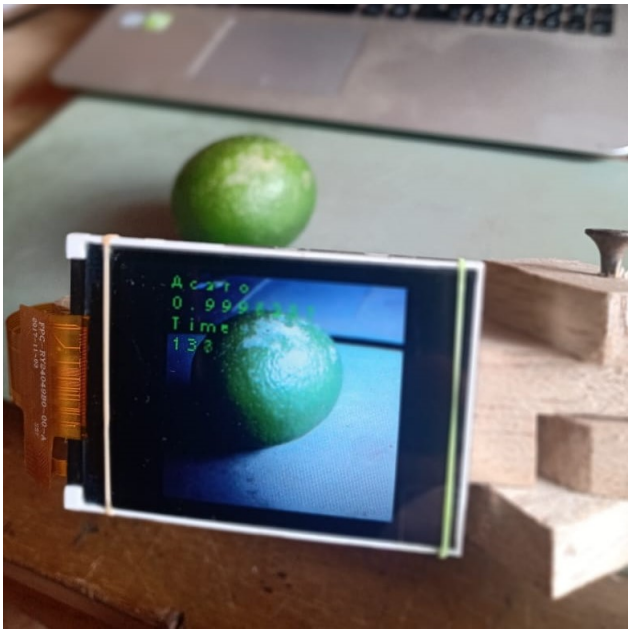
El tiempo de predicción aproximado es de 600 [ms], este tiempo varia entre los modelos entrenados ya que sus tamaños son diferentes pero se elige el mismo modelo para tener una buena comparación con otros sistemas embebidos, además de que este es el de mayor precisión.

4.3.2. Maix Bit.

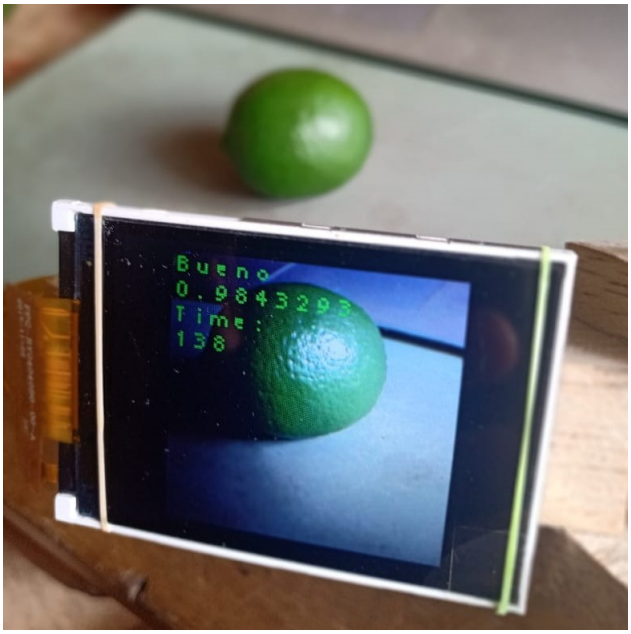
En cuanto a la Maix Bit el tiempo de predicción oscila entre los 160 y 170 ms cuando se carga el programa mediante la IDE; sin embargo, cuando se guarda el modelo dentro de la tarjeta SD el desempeño mejora y el tiempo de ejecución baja a un rango de entre 130 a 140 [ms]. En la imagen 31 se observa la evaluación del modelo en tiempo real con la tarjeta Maix Bit; en las imágenes la tarjeta no está conectada a la IDE.

Figura 31. Clasificaciones en tiempo real usando la Maix Bit.

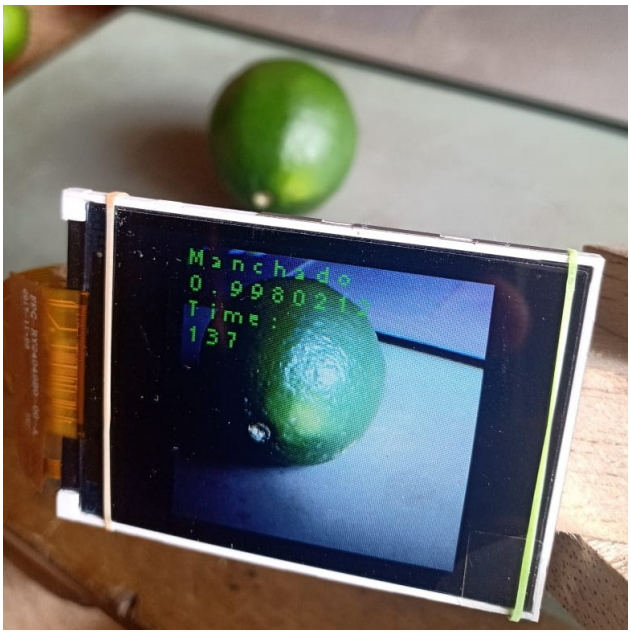
(a) Ácaros



(b) Buenos



(c) Manchados



Nota. Las etiquetas en verde corresponden a la clasificación del limón, el porcentaje de precisión y el tiempo de ejecución en [ms].

Tabla 11
Características de la implementación en los sistemas del modelo MobileNet 1_0.

Características	**Raspberry pi 3 b+**	**Maix Bit**
Peso del modelo	12,2 MB	3,14 MB
Tiempo de predicción	600 ms	137ms
Exactitud de predicción	93 %	94 %

Como aclaración sobre el peso del modelo para la Maix Bit la documentación oficial de la tarjeta indica que no debe exceder de 2Mbytes de peso (Kendryte, sf); sin embargo, esto solo es cierto cuando se desea poner a funcionar el modelo dentro de la memoria RAM de la tarjeta; en nuestro caso no fue necesario debido a que contamos con una tarjeta SD.

5. Conclusiones

Este trabajo de investigación estuvo enfocado en la clasificación de imágenes, en concreto del Limón Tahití con el fin de extraer las características más relevantes de acuerdo a las necesidades planteadas, y que; aún cuando el tema de este proyecto fue concreto, lo que se encontró puede aplicarse a cualquier otro campo o industria. Con los resultados obtenidos en el presente trabajo y tomando como referencia los objetivos planteados se redactan las siguientes conclusiones:

Acondicionar una base de datos es el proceso más importante en cualquier proyecto que haga uso de las redes neuronales, hoy en día existen diversos bancos de imágenes que pueden servir de punto de partida al momento de realizar una búsqueda de la información de interés; sin embargo, es posible que la información encontrada no se adecúe a la aplicación y por ende la mejor opción sea realizar la construcción de la base de datos.

Los sistemas embebidos que permiten realizar captura y procesamiento de imágenes son abundantes pero pocos cuentan con aceleradores convolucionales como el de la tarjeta Maix Bit; este acelerador permite que las aplicaciones de visión artificial presenten un buen desempeño con un reducido tiempo de respuesta.

Las mayoría de arquitecturas para la clasificación de imágenes puede abordar la clasificación de limones; sin embargo se debe tener especial cuidado al momento de su selección porque el peso de estas es decisivo si se desea implementar en dispositivos de *hardware* limitado lo que nos lleva a hacer uso de arquitecturas que se puedan cuantizar como las de Tensorflow Lite.

Al hacer uso del *Transfer-Learning* al momento de la elección del modelo se observó que la

cantidad de tiempo de entrenamiento disminuye y al mismo tiempo la precisión de la red aumenta en comparación a los modelos sin un entrenamiento previo.

De la implementación del modelo entrenado se observó el desempeño en dos dispositivos de los cuales la Maix Bit fue notoriamente mejor debido a su bajo tiempo de respuesta y bajo coste; al compararlo con la Raspberry se aprecia que tener un núcleo dedicado a acelerar las operaciones convolucionales le da la ventaja con respecto a sistemas que no cuentan con este, aun cuando tienen un mejor procesador y más almacenamiento.

Para garantizar un aprendizaje óptimo propiedades como luz, fondos o ángulos no deben ser controlados, pero si es aceptable tomar todos los datos en el entorno donde finalmente se piensa implementar el modelo y de ser posible con el tipo de cámara (sensor) que finalmente se piensa utilizar.

Adicionalmente el presente documento brinda información acerca del uso de la placa Maix Bit en el campo de clasificación de imágenes, este sirve para ampliar la documentación de la placa de desarrollo la cual, debido a ser un proyecto nuevo no cuenta con la suficiente información.

6. Recomendaciones

Con base en los resultados y las conclusiones del proyecto se considera necesario que para un futuro trabajo se puede tener en cuenta las siguientes recomendaciones en vista de mejorar los resultados obtenidos

Base de datos: Tener en cuenta factores como la distancia al momento de tomar las fotos puede ayudar a calcular el tamaño del fruto; de igual forma tener una cuadrícula guía sobre la cual poner el limón serviría como referencia para calcular el área del fruto mediante técnicas de procesamiento de imágenes.

Clasificador automatizado: Para lograr hacer un sistema automático de selección de frutos se podría optar por pasar de un clasificador de imágenes a un detector de objetos ya que con esta técnica se pueden hacer la clasificación de más objetos al mismo tiempo.

Referencias Bibliográficas

Aggarwal, C. C. (2018). *Neural networks and deep learning*. Springer.

Arias, F. & Suarez, E. (2017). Comportamiento de las exportaciones de limón persa (citrus latifolia tanaka) al mercado de los estados unidos. *Journal of Agriculture and Animal Sciences*, 2(2).

Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer.

CBI (2018). Exporting fresh limes to europe. `https://www.cbi.eu/market-information/fresh-fruit-vegetables/limes`.

Copeland, B. J. (2004). *The essential turing*. Clarendon Press.

DANE (2015). Cultivo del limón o lima tahití (citrus latifolia tanaka) frente a los efectos de las condiciones climáticas adversas. Boletín mensual 41.

Deebul, N., Amirhossein, P., & Paul, G. P. (2020). Performance evaluation of low-cost machine vision cameras for image-based grasp verification. `https://arxiv.org/pdf/2003.10167.pdf`.

Demuth, H. B., Beale, M. H., De Jess, O., & Hagan, M. T. (2014). *Neural Network Design*. Martin Hagan, Stillwater, OK, USA, 2nd edición.

Galeano, G. (2009). *SISTEMAS EMBEBIDOS en C*. Alfaomega.

Halfacree, G. (2018). *THE OFFICIAL Raspberry Pi Beginner's Guide How to use your new computer*. Raspberry Pi Press.

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

Kendryte (s.f). kendryte/nncase. `https://github.com/kendryte/nncase`.

Maslov, D. (s.f). axelerate, keras-based framework for ai on the edge. `https://github.com/AIWintermuteAI/aXeleRate`.

Mitchell, T. (1997). *Machine Learning*. McGraw-Hill.

Moore, S., Ha, J., Bailey, E., Wade, K., Benokraitis, A., Kennedy, P., Johnson, M., & Goldin, M. (2005). *Red Hat Enterprise Linux 4 Manual de referencia*. Red Hat.

Muñoz, R. (2019). Aprendizaje profundo en dispositivo portable para el reconocimiento de frutas y verduras. B.S. thesis, Universidad Autónoma de Occidente.

Nielsen, M. A. (2015). *Neural networks and deep learning*. Determination Press, San Francisco.

Olabe, X. B. (1998). Redes neuronales artificiales y sus aplicaciones. *Publicaciones de la Escuela de Ingenieros*.

Rivas Asanza, W. & Mazón Olivo, B. (2018). *Redes neuronales artificiales aplicadas al reconocimiento de patrones*. Utmach, 1ra edición.

Rungta, K. (2019). *TensorFlow in 1 Day: Make your own Neural Network*. Amazon Digital Services LLC - Kdp Print Us.

Russell, S., Norvig, P., & Rodríguez, J. (2004). *Inteligencia artificial: un enfoque moderno*. Pearson Educación.

Seed-Studio (2019). Sipeed maix bit for risc-v ai iot. `https://www.seeedstudio.com/Sipeed-MAix-BiT-for-RISC-V-AI-IoT-p-2872.html`.

Segura, M. (2008). Ficha técnica de manejo cosecha y poscosecha del limón tahiti. Ficha técnica.

SIOC (2019). Cadena de cítricos, indicadores e instrumentos diciembre 2019. Boletín cuatrimestral 4.

Sipeed (s.f). Sipeed - github. `https://github.com/sipeed`.

Soria, E. & Blanco, A. (2001). Redes neuronales artificiales. *Asociación de Autores Científico-Técnicos y Académicos*.

Tensorflow (s.f). Inferencia de tensorflow lite. `https://www.tensorflow.org/lite/guide/inference?hl=es-419`.