



Universidad Latina de Costa Rica

Facultad de Ingenierías, Tecnologías de la Información y Comunicación

Bachillerato en Ingeniería Telemática

Tema:

MACHINE LEARNING DE UN MODELO DE COLORIMETRÍA

Elaborado por:

Kenny Calderón H  
Manuel Cespedes  
Alois Tobal  
David Benavides

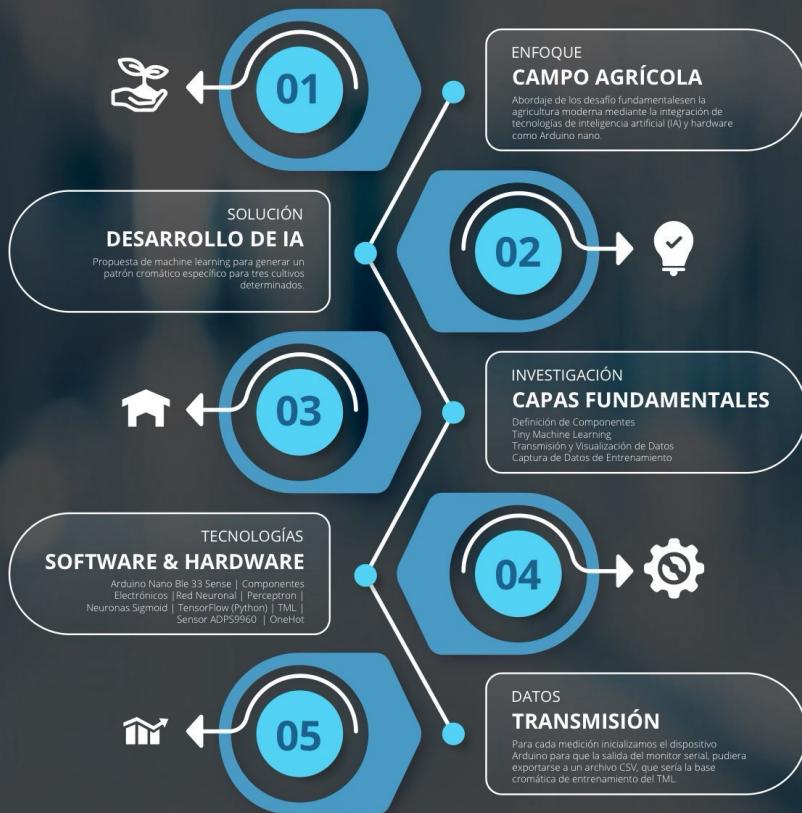
## Tabla de contenido

<b>Capítulo 1. Introducción .....</b>	<b>5</b>
<b>1.1 Generalidades.....</b>	<b>5</b>
<b>1.2 Antecedentes del Problema: El enfoque en el campo agrícola de este proyecto a determinado el alcance de los componentes y las herramientas a utilizar.....</b>	<b>6</b>
<b>1.3 Definición y Descripción del Problema.....</b>	<b>7</b>
<b>1.4 Justificación .....</b>	<b>8</b>
<b>1.5 Objetivos de investigación.....</b>	<b>9</b>
<b>1.5.1 Objetivo General.....</b>	<b>9</b>
<b>1.5.2 Objetivos Específicos.....</b>	<b>9</b>
<b>Capítulo 2. Marco Teórico o Conceptual.....</b>	<b>11</b>
<b>1.1 Definición de los componentes .....</b>	<b>11</b>
<b>1.1.1 Arduino .....</b>	<b>11</b>
<b>1.1.2 Red Neuronal.....</b>	<b>16</b>
<b>1.1.3 El Perceptrón .....</b>	<b>19</b>
<b>1.1.4 Neuronas Sigmoid.....</b>	<b>23</b>
<b>1.1.5 TensorFlow sobre Phyton .....</b>	<b>25</b>
<b>1.1.6 Tensores.....</b>	<b>26</b>
<b>1.2 TML en Microprocesadores.....</b>	<b>27</b>
<b>1.3 Transmisión y Visualización de los datos.....</b>	<b>32</b>
<b>1.3.1 Configurando el IDE de Arduino .....</b>	<b>32</b>
<b>1.4 Transmisión y Visualización de los datos.....</b>	<b>39</b>
<b>1.4.1 Transmisión de datos desde el monitor serial .....</b>	<b>39</b>
<b>1.4.2 Visualización grafica de la salida.....</b>	<b>40</b>
<b>1.5 Captura de Datos de Entrenamiento y Entrenamiento TML .....</b>	<b>40</b>
<b>1.5.1 Entrenamiento y modelo en Python con Tensorflow. ....</b>	<b>40</b>
<b>1.5.2 Entrenamiento y modelo en Python con Tensorflow. ....</b>	<b>46</b>
<b>1.5.3 Agregar Datos de prueba y ejecución. ....</b>	<b>49</b>
<b>1.5.4 Convertir el modelo entrenado en su versión a TensorFlow Lite. ....</b>	<b>52</b>
<b>1.6 Programar el modelo TensorFlow Lite Micro en la placa Arduino .....</b>	<b>53</b>

<i>Capítulo 3. Conclusiones y Recomendaciones .....</i>	<i>60</i>
<i>    3.    Conclusiones .....</i>	<i>60</i>
<i>    4.    Recomendaciones para su correcto uso y aplicación.....</i>	<i>61</i>
<i>Bibliografía .....</i>	<i>62</i>

# MACHINE LEARNING

## MODELO DE COLORIMETRÍA



## **Capítulo 1. Introducción**

### **1.1 Generalidades**

En la era contemporánea, las tecnologías de inteligencia artificial (IA) han emergido como catalizadores clave para la innovación y el progreso en una amplia gama de campos. Desde la atención médica hasta la manufactura, desde la agricultura hasta el transporte, la IA está transformando fundamentalmente la forma en que interactuamos con el mundo que nos rodea. En este contexto, el papel de la integración de la inteligencia artificial con plataformas de hardware como Arduino es cada vez más relevante y emocionante.

Arduino, una plataforma de hardware de código abierto ampliamente utilizada, ha democratizado la electrónica y la informática, permitiendo a los aficionados, estudiantes y profesionales crear una amplia variedad de dispositivos interactivos. Al combinar la versatilidad de Arduino con las capacidades de la inteligencia artificial, se abren nuevas fronteras en la aplicación práctica de estas tecnologías en la vida cotidiana.

Este trabajo de investigación explora la importancia de las tecnologías de inteligencia artificial y cómo su integración con plataformas como Arduino puede llevar a su aplicación en procesos de la vida real. Específicamente, nos enfocaremos en la introducción de entradas a los nodos de una red neuronal utilizando la información de los sensores de un Arduino. Esta sinergia entre IA y Arduino no solo permite la creación de dispositivos inteligentes, sino que también amplía el alcance de la automatización y la toma de decisiones basada en datos en una variedad de contextos.

A lo largo de este trabajo, exploraremos un ejercicio concreto de cómo la integración de IA y Arduino está siendo utilizada en aplicaciones del mundo real para la detección de colores a través de los sensores, desde sistemas de monitoreo ambiental hasta dispositivos médicos inteligentes. Al comprender las posibilidades y los desafíos de esta convergencia tecnológica, podemos vislumbrar un futuro en el que la inteligencia artificial se convierta en una parte integral de nuestro entorno físico, mejorando la eficiencia, la precisión y la autonomía en una variedad de industrias y áreas de la vida cotidiana.

**1.2 Antecedentes del Problema:** El enfoque en el campo agrícola de este proyecto a determinado el alcance de los componentes y las herramientas a utilizar.

La agricultura es uno de los pilares fundamentales de la civilización humana, pero enfrenta desafíos significativos en un mundo cada vez más poblado y cambiante. Para garantizar la seguridad alimentaria y la sostenibilidad ambiental, es crucial adoptar tecnologías innovadoras que optimicen los procesos agrícolas y maximicen el rendimiento de los cultivos. En este contexto, la integración de tecnologías de inteligencia artificial y hardware como Arduino nano (para disminuir el consumo energético y la dependencia de fuentes externas de energía) se presenta como una solución prometedora para abordar las necesidades específicas de la agricultura moderna.

Uno de los desafíos clave en la agricultura es la identificación y clasificación precisa de los cultivos y sus condiciones de crecimiento. Tradicionalmente, esta tarea ha sido realizada manualmente por agricultores expertos, pero con el crecimiento de la demanda de alimentos y la escasez de mano de obra, se requiere un enfoque más eficiente y automatizado. Aquí es donde entra en juego la combinación de sensores avanzados como el APDS9960 (sensor de proximidad, luz, color y gestos) y tecnologías de IA como TensorFlow.

El sensor APDS9960, con su capacidad para detectar colores y la proximidad, ofrece una herramienta valiosa para la identificación y clasificación de frutos en la agricultura. Al integrar este sensor con una plataforma Arduino, se puede recopilar datos precisos sobre la madurez, el tamaño y la calidad de los frutos en tiempo real. Estos datos pueden alimentar una red neuronal entrenada con TensorFlow, que puede aprender a identificar diferentes tipos de plantas basadas en las características de sus frutos.

Una vez que la red neuronal ha sido entrenada con suficiente precisión, se puede implementar en el campo para realizar tareas específicas, como el monitoreo del crecimiento de los cultivos, la detección de enfermedades o plagas, y la gestión de la irrigación y fertilización. Por ejemplo, la red neuronal puede ejecutar instrucciones específicas para ajustar los niveles de riego, la cantidad de fertilizante o la exposición a la luz solar, según las necesidades de cada tipo de planta identificado.

En resumen, la combinación de tecnologías de IA, hardware como Arduino y sensores avanzados ofrece un enfoque integral y eficiente para abordar las necesidades y desafíos de la agricultura moderna. Al permitir una gestión más precisa y automatizada de los cultivos, estas tecnologías tienen el potencial de aumentar la productividad, reducir los costos y promover la sostenibilidad en la industria agrícola.

### **1.3 Definición y Descripción del Problema**

El proyecto en cuestión plantea la implementación de una IA en un modelo de red neuronal, para realizar la identificación automática de frutos a través de sus colores, para la toma proactiva de decisiones en términos de riego, nutrientes y otros factores que fortalezca el crecimiento productivo del cultivo. La densidad tonalidades en los colores de los frutos de los cultivos y la semejanza cromática de los mismo para diversas especies plantea un reto en la identificación ya que no se puede generalizar un patrón RGB para determinar dicho cultivo. Es por esta razón que se ha determinado entregar una IA a través de machine learning para generar un patrón cromático específico para 03 determinados cultivos y que la misma puede generar una certeza probable a nivel porcentual del origen y el tipo de cultivo.

#### **Definición del Problema:**

La diversidad de tonalidades en los colores de los frutos y la semejanza cromática entre diferentes especies dificultan la identificación manual de los cultivos en la agricultura. No se puede generalizar un patrón RGB para determinar un cultivo específico, lo que implica la necesidad de una solución más avanzada para la identificación automatizada.

#### **Propuesta de Solución del Problema:**

Se propone desarrollar una IA basada en machine learning para generar un patrón cromático específico para tres cultivos determinados. La IA utilizará un modelo de red neuronal entrenado con TensorFlow para proporcionar una certeza probable a nivel porcentual sobre el origen y el tipo de cultivo.

#### **Capas funcionales del Proyecto:**

1. **Definición de Componentes:** Investigación y selección de hardware y software necesarios para el proyecto, incluyendo la plataforma Arduino, sensores como el APDS9960, y herramientas de desarrollo como TensorFlow.
2. **Transmisión y Visualización de Datos:** Implementación de la comunicación entre la placa Arduino y un sistema de visualización para monitorizar los datos del sensor en tiempo real.
3. **Captura de Datos de Entrenamiento:** Recopilación de muestras de colores y proximidad de los frutos para el entrenamiento del modelo de red neuronal.

4. **Cuantificación y Entrenamiento de Red Neuronal:** Desarrollo y entrenamiento de una red neuronal utilizando TensorFlow para clasificar los datos de color y proximidad de los frutos.
5. **Tiny Machine Learning (TML):** Investigación y aplicación de técnicas de machine learning en microcontroladores para el desarrollo de la IA en dispositivos de bajo consumo.
6. **Clasificación de Datos:** Implementación de algoritmos de clasificación para identificar los cultivos en base a los datos capturados por los sensores.
7. **Aplicación y Carga del Modelo:** Integración del modelo de red neuronal en la placa Arduino para el reconocimiento en tiempo real de los cultivos en el campo.
8. **Conclusiones:** Evaluación de la efectividad y las limitaciones del sistema implementado, así como posibles mejoras y aplicaciones futuras.

#### 1.4 Justificación

La implementación de una inteligencia artificial (IA) para la identificación automática de frutos en la agricultura mediante redes neuronales está justificada por varias razones fundamentales:

1. **Optimización de Procesos Agrícolas:** La identificación precisa de los cultivos es esencial para tomar decisiones informadas sobre la gestión del cultivo, incluyendo el riego, la aplicación de nutrientes y otros factores ambientales. La IA proporcionará una solución automatizada y eficiente para este proceso, mejorando la productividad y la calidad de los cultivos.
2. **Reducción de Costos y Tiempo:** La automatización de la identificación de cultivos mediante IA reducirá la necesidad de mano de obra manual y el tiempo dedicado a esta tarea. Esto conducirá a una reducción de costos operativos y una mayor eficiencia en la gestión agrícola.
3. **Precisión y Fiabilidad:** La IA basada en redes neuronales entrenadas con TensorFlow ofrecerá una mayor precisión y fiabilidad en la identificación de cultivos en comparación con métodos manuales o sistemas basados en reglas predefinidas. Esto garantizará una toma de decisiones más confiable y oportuna para los agricultores.
4. **Adaptabilidad a Diversas Condiciones:** La IA permitirá la adaptación a diferentes condiciones ambientales y tipos de cultivo, lo que la convierte en una solución versátil y escalable para una amplia gama de aplicaciones agrícolas.

5. **Promoción de la Sostenibilidad:** Al mejorar la eficiencia en la gestión de los cultivos, la IA contribuirá a la reducción del uso de recursos naturales como agua y fertilizantes, promoviendo así la sostenibilidad ambiental en la agricultura.

La implementación de una IA para la identificación automática de frutos en la agricultura mediante redes neuronales es una medida justificada que ofrece beneficios significativos en términos de optimización de procesos, reducción de costos, precisión y sostenibilidad. Este proyecto tiene el potencial de transformar la forma en que se gestionan los cultivos y contribuir al avance de la agricultura moderna hacia prácticas más eficientes y sostenibles.

## **1.5 Objetivos de investigación**

### ***1.5.1 Objetivo General***

Desarrollar e implementar un sistema basado en inteligencia artificial para la identificación automática de frutos en la agricultura, utilizando redes neuronales entrenadas con TensorFlow, con el fin de mejorar la gestión y optimizar los procesos de cultivo.

### ***1.5.2 Objetivos Específicos***

- 1 Investigar y Seleccionar Componentes de Hardware y Software:
  - 1.1 Realizar un estudio exhaustivo de los componentes disponibles en el mercado para determinar la mejor combinación de hardware y software que se adapte a las necesidades del proyecto.
  - 1.2 Evaluar las características técnicas, la compatibilidad y la facilidad de integración de cada componente, incluyendo la placa Arduino, el sensor APDS9960 y la herramienta de desarrollo TensorFlow.
  - 1.3 Seleccionar los componentes más adecuados en función de los criterios establecidos y documentar las razones de la elección.
- 2 Desarrollar e Implementar un Modelo de "Tiny Machine Learning":
  - 2.1 Investigar y comprender los conceptos fundamentales de machine learning y su implementación en microcontroladores.
  - 2.2 Adaptar y configurar el entorno de desarrollo para permitir la implementación de algoritmos de aprendizaje automático en microcontroladores Arduino.

- 2.3** Desarrollar y probar un modelo de "Tiny Machine Learning" que sea capaz de ejecutar algoritmos de IA de manera eficiente en dispositivos de bajo consumo y recursos limitados.
- 3 Establecer Método de Transmisión de Datos:
- 3.1** Investigar y evaluar diferentes métodos de comunicación entre la placa Arduino y un sistema de visualización externo.
- 3.2** Seleccionar el método de transmisión más adecuado en función de la velocidad, la fiabilidad y la facilidad de implementación.
- 3.3** Configurar y establecer la comunicación entre la placa Arduino y el sistema de visualización para permitir la monitorización y análisis en tiempo real de los datos capturados por el sensor.
- 4 Recolectar Datos de Entrenamiento:
- 4.1** Diseñar un protocolo de recolección de datos para capturar muestras de colores y proximidad de los frutos de los cultivos.
- 4.2** Realizar experimentos de campo para recopilar una cantidad suficiente de datos de entrenamiento que representen la diversidad de colores y características de los frutos.
- 4.3** Registrar y etiquetar correctamente los datos recolectados para su posterior uso en el entrenamiento del modelo de red neuronal.
- 5 Desarrollar y Entrenar una Red Neuronal con TensorFlow:
- 5.1** Diseñar la arquitectura de la red neuronal, incluyendo el número de capas, el tipo de neuronas y las funciones de activación.
- 5.2** Preprocesar y normalizar los datos de entrenamiento para garantizar un entrenamiento eficaz y una buena generalización del modelo.
- 5.3** Entrenar la red neuronal utilizando TensorFlow, ajustando los hiperparámetros y realizando pruebas iterativas para mejorar la precisión y la eficiencia del modelo.

## Capítulo 2. Marco Teórico o Conceptual

### 1.1 Definición de los componentes

#### 1.1.1 Arduino

Dada la naturaleza del proyecto, los componentes del entorno físico que tendrán contacto con los cultivos deben de cumplir los siguientes requerimientos funcionales:

- 1 **Función:** Buscar un dispositivo inteligente actúe de forma rápida y local (independiente de Internet) esto por las condiciones de uso final en granjas o fincas agrícolas.
- 2 **Costo:** lograr esto con hardware simple y de menor costo. Pero lo suficientemente flexible y expandible mediante módulos o la integración de otros sensores para aplicaciones futuras y fuera del alcance de este proyecto.
- 3 **Privacidad:** Para evitar en los exteriores agrícolas la multiplicidad de componentes y que estos puedan generar un riesgo en las lecturas y mediciones, se desea no compartir todos los datos de los sensores externamente. Para lo cual los mismo deben estar lo más condensados o formar parte de la placa base.
- 4 **Eficiencia:** Se debe buscar el factor de forma de dispositivo más pequeño, es primordialmente para el consumo de energía y que la función pueda ser suplida por una batería comercial de bajo amperaje.

Tomando en cuenta lo anterior se ha determinado utilizar la placa Arduino: **Arduino Nano 33 BLE Sense<sup>1</sup>**, el cual incluye dentro de su placa base los sensores de Colorímetro y sensor de proximidad para clasificar objetos.

Pero empecemos por definir que es Arduino? Arduino es una plataforma de hardware de código abierto diseñada para facilitar la creación de proyectos electrónicos interactivos. Consiste en una placa de circuito impreso con un microcontrolador y un entorno de desarrollo integrado (IDE) que permite programar y cargar código en la placa. Arduino se destaca por su simplicidad y

<sup>1</sup> Para información de la Placa: [https://store.arduino.cc/products/arduino-nano-33-ble-sense?gl=1%2Aa466d%2A\\_ga%2AMTcwMTc4ODc2My4xNzA4MjY3MjM0%2A\\_ga\\_NEXN8H46L5%2AMTcxMjQxODM0NC4xNS4xLjE3MTI0MTk4MjMuMC4wLjE0NjMzMzkzODk.%2A\\_fplc%2ANCUyQkFJRVFma2xEd2FYazhibjBGN1k3T2J1dTIOYIQ0JTJCc3dIcEcIMklzTGZ3WVFSZGpSWtaZCUyQmQzYzVUMHVQd0JyMTVIZzBIWEYxQko3VzxODV0MEtHTG80U0tSV3Nud1hkU0pkdGxzOFll0WnckZFMXpLzNzNzQnF0c1Q0V3clM0QlM0Q.&utm\\_campaign=ai&utm\\_content=fruit\\_idenification&utm\\_source=blog](https://store.arduino.cc/products/arduino-nano-33-ble-sense?gl=1%2Aa466d%2A_ga%2AMTcwMTc4ODc2My4xNzA4MjY3MjM0%2A_ga_NEXN8H46L5%2AMTcxMjQxODM0NC4xNS4xLjE3MTI0MTk4MjMuMC4wLjE0NjMzMzkzODk.%2A_fplc%2ANCUyQkFJRVFma2xEd2FYazhibjBGN1k3T2J1dTIOYIQ0JTJCc3dIcEcIMklzTGZ3WVFSZGpSWtaZCUyQmQzYzVUMHVQd0JyMTVIZzBIWEYxQko3VzxODV0MEtHTG80U0tSV3Nud1hkU0pkdGxzOFll0WnckZFMXpLzNzNzQnF0c1Q0V3clM0QlM0Q.&utm_campaign=ai&utm_content=fruit_idenification&utm_source=blog)

accesibilidad, lo que lo hace accesible para profesionales en campos como la electrónica, la robótica, la domótica y la Internet de las cosas (IoT).

La placa Arduino está compuesta por un microcontrolador principal, como los de la familia AVR de Atmel o los de la serie SAMD de Microchip, que ejecuta el código programado en su interfaz de usuario. Además del microcontrolador, la placa incluye una variedad de puertos de entrada y salida (E/S) que permiten conectar sensores, actuadores y otros componentes electrónicos. En nuestro caso, la placa seleccionada incluye sensores integrados que describiremos adelante.

El entorno de desarrollo Arduino proporciona un lenguaje de programación basado en Wiring, una versión simplificada de C/C++, y una serie de bibliotecas de software que facilitan la interacción con los componentes conectados a la placa. La interacción del código con la placa se hace directamente en el IDE de Arduino, compilarlo y cargarlo en la placa a través de un cable USB, lo que permite la ejecución de programas autónomos en tiempo real.

El Nano 33 BLE Sense, es la placa Arduino habilitada para IA su funcionamiento a diferencia de modelos anteriores y de ahí su nombre de 33BLE es que opera a: 3,3 V y tiene el factor de forma pequeño disponible: 45 x 18 mm. Así como lo mencionábamos anteriormente cuenta con una serie de sensores integrados: sensor inercial de 9 ejes, sensor de humedad y temperatura, sensor barométrico, micrófono, sensor de gestos, proximidad, color de luz e intensidad de luz.

#### Diagrama PIN-OUT del Componente

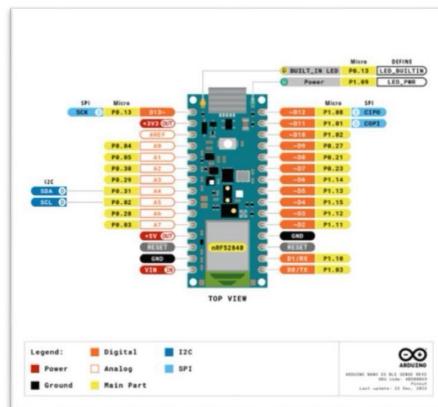


Figura 1. Diagrama de Pines de la estructura física del 33 BLE

## Componente Real del Proyecto



Figura 2. Arduino NANO 33 SENSE BLE

El nombre “Arduino Nano 33 BLE Sense” brinda información importante. Se llama “Nano” porque las dimensiones, el pinout y el factor de forma son muy similares al clásico Arduino Nano , en realidad está planeado para usarse como reemplazo de Arduino Nano en sus proyectos existentes, pero el problema es que este nuevo módulo **funciona en 3.3V** mientras que el clásico Nano funciona con 5V. “33”, indica que la placa funciona con 3.3V, “BLE” indica que el módulo es compatible con **Bluetooth Low Energy (BLE5.0)** y la palabra “SENSE” indica que tiene sensores integrados como acelerómetro, giroscopio, magnetómetro, sensor de temperatura y humedad, sensor de presión, sensor de proximidad, sensor de color, sensor de gestos e incluso un micrófono incorporado.

En una revisión preliminar superior de la placa, se pueden observar muchos componentes apiñados, la mayoría de los cuales son sensores. Detrás de la carcasa de metal en el lado derecho, se encuentra su unidad de procesamiento central: **procesador Nordic nRF52840** que contiene un Cortex M4F y el módulo **NINA B306** para comunicación BLE y Bluetooth 5. Esto permite que la placa funcione con muy poca energía y se comunique usando Bluetooth 5, lo que la hace ideal para **aplicaciones de red de malla de baja potencia** en domótica y otros proyectos conectados.

Además, el procesador **nRF52840** es compatible con el sistema operativo **ARM Mbed**,

### Arduino NANO BLE 33 SENSE

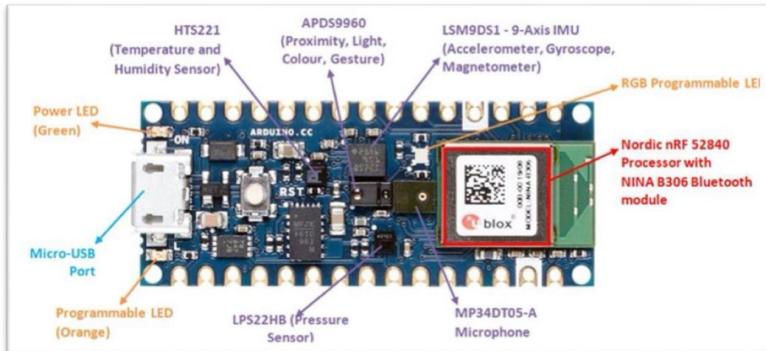


Figura 3. Estructura de componentes NANO 33 BLE SENSE

Los detalles del sensor en la placa de detección Arduino Nano 33 BLE junto con sus bibliotecas requeridas se tabulan a continuación:

### Arduino NANO BLE 33 SENSE, sensores

Nombre del sensor	Parámetros	Enlaces
<i>LSM9DS1 – ST Microelectronics</i>	Acelerómetro, giroscopio, magnetómetro	Hoja de datos de LSMDSI Biblioteca Arduino_LSM9DS1
<i>LPS22HB – ST Microelectronics</i>	Presión	LPS22HB Hoja de datos Arduino_LPS22HB Biblioteca
<i>HTS221 – Microelectrónica ST</i>	Temperatura y humedad	LPS22HB Hoja de datos Arduino_HTS221 Biblioteca
<i>APDS9960 – Avago Tech.</i>	Proximidad, Luz, Color, Gesto	LPS22HB Hoja de datos Arduino_APDS9960 Biblioteca
<i>MP34DT05 – ST Microelectronics</i>	Micrófono	MP34DT05 Hoja de datos Inbuilt-PDM Library

Tabla1. Detalle de sensores y bibliotecas dentro del núcleo de IDE

La mayoría de estos sensores son de **ST Microelectronics** y admiten operaciones de baja potencia, lo que los hace ideales para diseños con baterías. Es de vital importancia para el proyecto en cuestión el **sensor APDS9960** ya que será con el cual se hagan las mediciones para el entrenamiento y donde se utilice para realizar la captura de información. Para la utilización de los sensores por parte del sistema Arduino IDE o su versión CloudWEB, se debe de asegurar que las librerías de los componentes estén agregadas a la biblioteca y de manera provista para la placa IDE Arduino que se va a utilizar. Para agregar una biblioteca, se puede usar el enlace dado para acceder a la página respectiva de GitHub y descargar el archivo ZIP, luego crear un “Sketch” (que es un archivo único compilable), proceder a “Incluir biblioteca |Aregar biblioteca ZIP” o sino también se puede usar el administrador de la biblioteca en Arduino IDE y agregar esas bibliotecas.

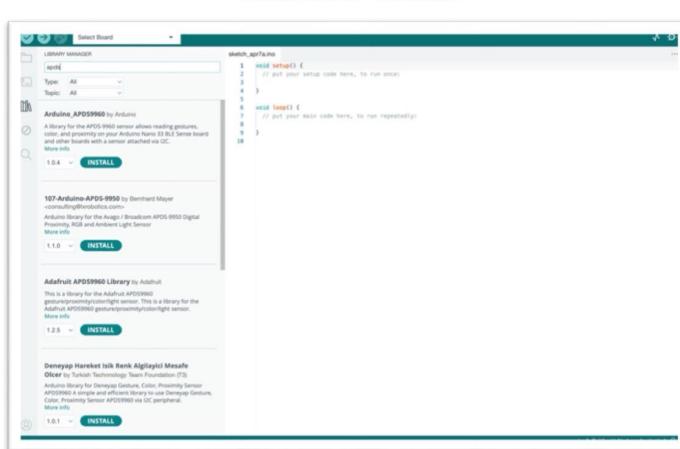


Figura 4. Administrador de Librerías en Arduino IDE, en relación al sensor APDS9960

Como hemos dicho anteriormente, el **nRF52840** se puede programar utilizando ARM Mbed OS, esto significa que la placa Arduino Nano 33 es compatible con el sistema operativo en tiempo real (RTOS). Con la programación de Mbed OS podemos ejecutar múltiples hilos al mismo tiempo en el programa para realizar tareas múltiples. Además, el consumo de energía de la placa se reducirá considerablemente, cada vez que llamemos a la función de retraso, la placa entrará en **modo hold**

durante el tiempo de retraso para ahorrar energía y volvería a funcionar una vez que el retraso haya terminado. Se informa que esta operación consumirá 4.5uA menos que una operación de retardo Arduino normal.

Finalmente Arduino Nano 33 BLE se podría utilizar para multiples aplicaciones portátiles que utilicen inteligencia artificial para reconocer movimientos y esto a su capacidad de ejecutar aplicaciones Edge Computing (AI) utilizando TinyML (TML), que veremos más adelante en este documento. Algunas aplicaciones adicionales que se podría aplicar a futuro y tomando en cuenta la inversión en el componente podemos mencionar:

- 1 Monitorear la temperatura ambiente que pueda sugerir o modificar cambios en un termostato o tomar acciones en cuanto a iniciar ventiladores u otros dispositivos.
- 2 Desarrollar un dispositivo de reconocimiento de voz o gestos utilizando el micrófono o el sensor de gestos junto con las capacidades de IA de la placa.

### 1.1.2 Red Neuronal

Una red neuronal es un modelo computacional inspirado en el funcionamiento del cerebro humano, diseñado para procesar información y realizar tareas específicas, como reconocimiento de patrones, clasificación, predicción, entre otros. Está compuesta por un conjunto de nodos interconectados llamados neuronas, organizados en capas.

Las redes neuronales artificiales están básicamente formadas por 03 capas :

- 3 **Capa de entrada:** Esta capa recibe los datos de entrada y transmite esta información a la siguiente capa.
- 4 **Capas ocultas:** Estas capas intermedias procesan la información recibida de la capa anterior y aplican transformaciones no lineales a los datos. Cuantas más capas ocultas tenga la red neuronal, más complejas pueden ser las relaciones que puede aprender.
- 5 **Capa de salida:** Esta capa produce los resultados finales de la red neuronal después de procesar la información a través de las capas ocultas. Dependiendo de la tarea que la red esté diseñada para realizar, la capa de salida puede tener una o varias neuronas.

Cada conexión entre las neuronas está asociada con un peso que determina la importancia relativa de la señal transmitida. Durante el entrenamiento de la red neuronal, estos pesos se ajustan

iterativamente mediante algoritmos de optimización para minimizar una función de pérdida, lo que permite que la red aprenda a realizar la tarea deseada de manera más precisa.

Una de las características clave de las redes neuronales es su capacidad para aprender y generalizar a partir de datos, lo que las hace útiles en una amplia variedad de aplicaciones, como reconocimiento de imágenes, procesamiento de lenguaje natural, diagnóstico médico, control de procesos industriales, entre otros. Las redes neuronales profundas, que tienen múltiples capas ocultas, han demostrado ser especialmente efectivas en la resolución de problemas complejos y en el manejo de grandes conjuntos de datos.

Para facilitar su entendimiento vamos a seguir el proceso de una red neuronal para el reconocimiento por parte de un sistema de unos dígitos escritos a mano, aprovechando el trabajo de: *Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015.*

El sistema visual humano es una de las maravillas del mundo, su complejidad y precisión son asombrosas, para el ejemplo vamos a considerar la siguiente secuencia escrita a mano:

#### Escritura de Números a Mano

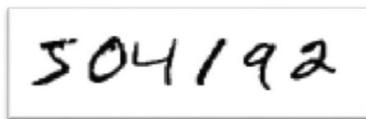


Figura 5. Escritura a mano para el reconocimiento Artificial

La mayoría de la gente reconoce fácilmente esos dígitos como 504192. Esa facilidad es engañosa. En cada hemisferio de nuestro cerebro, los humanos tenemos una corteza visual primaria, también conocida como V1, que contiene 140 millones de neuronas, con decenas de miles de millones de conexiones entre ellas.

Por otra parte la visión humana implica no sólo el campo V1, sino toda una serie de cortezas visuales (V2, V3, V4 y V5) que realizan un procesamiento de imágenes cada vez más complejo. Llevamos en la cabeza una supercomputadora, adaptada por la evolución a lo largo de cientos de millones de años y magníficamente adaptada para comprender el mundo visual.

Reconocer dígitos escritos a mano no es fácil, los humanos somos muy buenos para darle sentido a lo que nos muestran nuestros ojos. Pero casi todo ese trabajo se hace de forma inconsciente.

La dificultad del reconocimiento de patrones visuales se vuelve evidente si intenta escribir un programa de computadora para reconocer dígitos como los anteriores. Lo que parece fácil cuando lo hacemos nosotros mismos, de repente se vuelve extremadamente difícil. Intuiciones simples sobre cómo reconocemos las formas - "un 9 tiene un bucle en la parte superior y un trazo vertical en la parte inferior derecha" - resultan no ser tan simples de expresar algorítmicamente. Cuando se intenta hacer que dichas reglas sean precisas, rápidamente se pierde en un mar de excepciones, advertencias y casos especiales.

Las redes neuronales abordan el problema de una manera diferente. La idea es tomar una gran cantidad de dígitos escritos a mano, conocidos como ejemplos de entrenamiento, como se muestran en la figura #6.

**Tabla de entrenamiento de Escritura de números.**

0	4	1	9	2	1	3	1	4	3
5	3	6	1	7	2	8	6	9	4
0	9	1	1	2	4	3	2	7	3
8	6	9	0	5	6	0	7	6	1
8	7	9	3	9	8	5	9	3	3
0	7	4	9	8	0	9	4	1	4
4	6	0	4	5	6	1	0	0	1
7	1	6	3	0	2	1	1	7	9
0	2	6	7	8	3	9	0	4	6
7	4	6	8	0	7	8	3	1	5

Figura 6. Tabla de entrenamiento de 100 entradas para el reconocimiento Artificial

Para luego desarrollar un sistema que pueda aprender de esos ejemplos de capacitación. La red neuronal utiliza los ejemplos para inferir automáticamente reglas para reconocer dígitos escritos a mano. Además, al aumentar el número de ejemplos de formación, la red puede aprender más sobre escritura a mano y así mejorar su precisión. Entonces, en la figura #6 solo hay 100 dígitos de

entrenamiento, tal vez podríamos construir un mejor reconocedor de escritura usando miles o incluso millones o miles de millones de ejemplos de entrenamiento.

Por supuesto, un aspecto clave de las redes neuronales, incluyen dos tipos importantes de neuronas artificiales (**el perceptrón y la neurona sigmoidea**) y el algoritmo de aprendizaje estándar para redes neuronales, conocido como **descenso de gradiente estocástico**. Los cuales son utilizados en nuestro proyecto agrícola.

### 1.1.3 El Perceptrón

Los perceptrones fueron desarrollados en las décadas de 1950 y 1960 por el científico Frank Rosenblatt, inspirado en trabajos anteriores de Warren McCulloch y Walter Pitts. Un perceptrón toma varias entradas binarias,  $x_1, x_2, \dots$ , y produce una sola salida binaria:

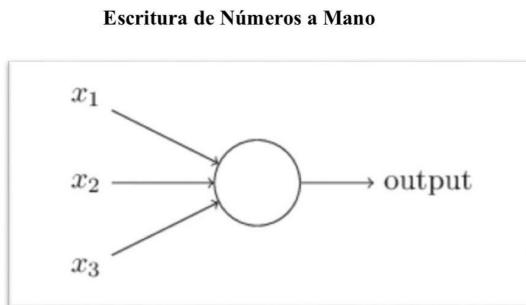


Figura 7. Escritura a mano para el reconocimiento Artificial

En el ejemplo mostrado, el perceptrón tiene tres entradas,  $x_1, x_2, x_3$ , pero podría bien tener muchas más. Rosenblatt propuso una regla simple para calcular el resultado. Introdujo pesos,  $w_1, w_2, \dots$ , números reales que expresan la importancia de las respectivas entradas para la salida. La salida de la neurona, 0 o 1, está determinada por si la suma ponderada  $\sum_j w_j x_j$  es menor o mayor que algún valor umbral. Al igual que los pesos, el umbral es un número real que es un parámetro de la neurona.

En términos algebraicos:

$$\text{salida} = \begin{cases} 0 & \text{si } \sum_j w_j x_j \leq \text{umbral} \\ 1 & \text{si } \sum_j w_j x_j > \text{umbral} \end{cases}$$

Ese es el modelo matemático básico. Una forma de pensar sobre el perceptrón es que es un dispositivo que toma decisiones sopesando la evidencia. El perceptrón no es un modelo completo de la toma de decisiones humana, pero ilustra cómo un perceptrón puede sopesar diferentes tipos de evidencia para tomar decisiones. Y debería parecer plausible que una red compleja de perceptrones pudiera tomar decisiones bastante aceptables:

### Red Compleja de Perceptrones

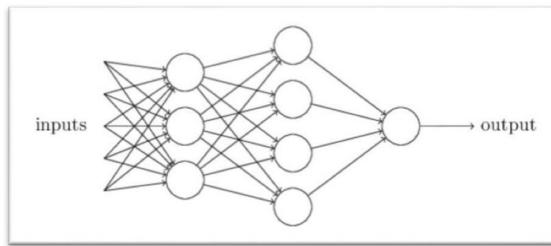


Figura 8. Red Neuronal

En esta red, ver figura #6, la primera columna de perceptrones (la primera capa de perceptrones) toma tres decisiones muy simples, sopesando la evidencia de entrada. En la segunda etapa (los 4 perceptores) cada uno de esos perceptrones toma una decisión sopesando los resultados de la primera capa de toma de decisiones. De esta manera, un perceptrón de la segunda capa puede tomar una decisión a un nivel más complejo y abstracto que los perceptrones de la primera capa. Y el perceptrón de la tercera capa puede tomar decisiones aún más complejas. De esta manera, una red de perceptrones de muchas capas puede participar en una toma de decisiones más precisa y sofisticada.

A pesar de que los perceptrones parecen tener múltiples salidas, siguen siendo de salida única. Las múltiples flechas de salida son simplemente una forma útil de indicar que la salida de un perceptrón se está utilizando como entrada para varios otros perceptrones.

Simplificando la función aritmética del perceptrón, podemos hacer dos cambios de notación para simplificarla. En lugar de escribir  $\sum_j w_j x_j$  como un producto escalar:

$$w \cdot x \equiv \sum_j w_j x_j$$

donde  $w$  y  $x$  son vectores cuyos componentes son los pesos y las entradas. El segundo cambio es mover el umbral al otro lado de la desigualdad y reemplazarlo por lo que se conoce como sesgo del perceptrón:

$$b \equiv -\text{umbral}$$

Usando el sesgo en lugar del umbral, la regla del perceptrón se puede reescribir:

$$\text{salida} = \begin{cases} 0 & \text{si } w \cdot x + b \leq \text{umbral} \\ 1 & \text{si } w \cdot x + b \leq \text{umbral} \end{cases}$$

El sesgo es una medida de lo fácil que es conseguir que el perceptrón se active. Para un perceptrón con un sesgo realmente grande, es extremadamente fácil para el perceptrón generar un 1. Pero si el sesgo es muy negativo, entonces es difícil para el perceptrón generar un 1.

Otra forma en que se pueden utilizar los perceptrones es para calcular las funciones lógicas elementales que normalmente consideramos computación subyacente, funciones como AND, OR y NAND. Por ejemplo, supongamos que tenemos un perceptrón con dos entradas, cada una con un peso: -2 y un sesgo general de: 3. Aquí está nuestro perceptrón:

### Perceptrón lógico

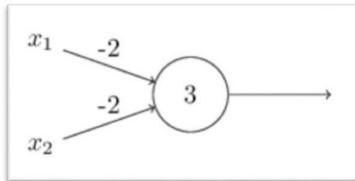


Figura 9. Perceptrones para realizar funciones lógicas: NAND

Si evaluamos la red de la figura #9, con una entrada binaria de: 00, produce la salida 1; ya que:  $(-2) * 0 + (-2) * 0 + 3 = 3$  y 3 es positiva. Cálculos similares muestran que las entradas 01 y 10 producen la salida 1. Pero la entrada 11 produce la salida 0, ya que  $(-2) * 1 + (-2) * 1 + 3 = -1$  y -1 es negativo. Este perceptrón implementa una puerta NAND.

Podemos utilizar redes de perceptrones para calcular cualquier función lógica. La razón es que la puerta NAND es universal para el cálculo, es decir, podemos construir cualquier cálculo a partir de puertas NAND.

### Equivalencia lógica con perceptrones

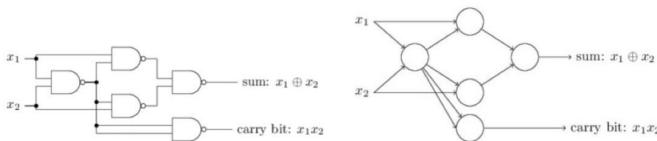


Figura 10. Equivalencia con perceptrones de un arreglo NAND de compuertas lógicas.

La universalidad computacional de los perceptrones puede ser algo decepcionante, porque hace parecer que los perceptrones son simplemente un nuevo tipo de puerta NAND. Sin embargo, la situación es mejor de lo que sugiere este punto de vista. Podemos idear algoritmos de aprendizaje que puedan ajustar automáticamente los pesos y sesgos de una red de neuronas artificiales. Esta sintonización ocurre en respuesta a estímulos externos, sin intervención directa de un programador.

Estos algoritmos de aprendizaje nos permiten utilizar neuronas artificiales de una manera radicalmente diferente a las puertas lógicas convencionales. En lugar de diseñar explícitamente un circuito de NAND y otras puertas, nuestras redes neuronales pueden simplemente aprender a resolver problemas, a veces problemas en los que sería extremadamente difícil diseñar directamente un circuito convencional.

#### 1.1.4 Neuronas Sigmoid

Pensemos en una red de perceptrones donde las entradas a la red podrían ser los datos de píxeles sin procesar de una imagen escaneada y escrita a mano de un dígito y queremos que la red aprenda pesos y sesgos para que la salida de la red clasifique correctamente el dígito. Supongamos que hacemos un pequeño cambio en algún peso (o sesgo) en la red, nos gustaría es que este pequeño cambio en el peso cause solo un pequeño cambio en la salida de la red. Esta propiedad hará posible el aprendizaje.

#### Delta de cambio en los pesos

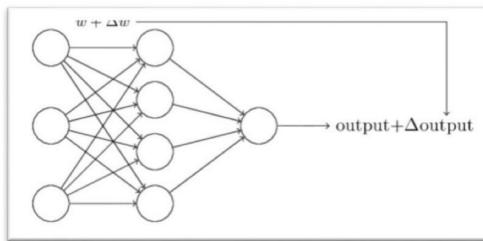


Figura 11. Diagrama de cambios

Si fuera cierto que un pequeño cambio en una ponderación (o sesgo) causa sólo un pequeño cambio en la producción, entonces podríamos usar este hecho para modificar las ponderaciones y los sesgos para lograr que nuestra red se comporte más de la manera que queremos. Por ejemplo, supongamos que la red clasifica por error una imagen como "8" cuando debería ser un "9". Podríamos descubrir cómo hacer un pequeño cambio en los pesos y sesgos para que la red se acerque un poco más a clasificar la imagen como un "9". Y luego repetiríamos esto, cambiando las ponderaciones y los sesgos una y otra vez para producir resultados cada vez mejores. La red estaría aprendiendo.

El problema es que esto no sucede cuando la red contiene perceptrones. De hecho, un pequeño cambio en los pesos o el sesgo de cualquier perceptrón en la red a veces puede causar que la salida de ese perceptrón cambie completamente, digamos de 0 a 1. Ese cambio puede causar que el comportamiento del resto de la red cambie. Entonces, si bien un "9" ahora puede estar clasificado correctamente, es probable que el comportamiento de la red en todas las demás imágenes haya cambiado por completo de alguna manera difícil de controlar. Eso dificulta ver cómo modificar

gradualmente los pesos y sesgos para que la red se acerque al comportamiento deseado.

Para superar este problema utilizaremos un nuevo tipo de neurona artificial llamada: **neurona sigmoid**. Las neuronas sigmoid son similares a los perceptrones, pero modificadas para que pequeños cambios en sus pesos y sesgos causen sólo un pequeño cambio en su producción. Ése es el hecho crucial que permitirá que una red de neuronas sigmoid aprenda a diferencia de las redes de perceptores.

Podemos representar las neuronas sigmoideas de la misma manera que representamos los perceptrones:

### Neurona Sigmoid

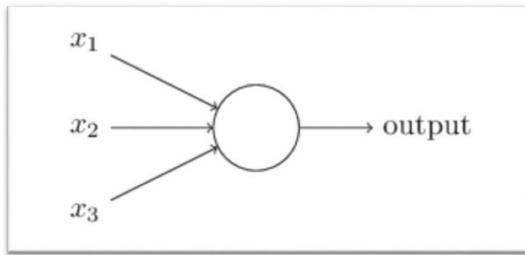


Figura 12. Reescritura de un diagrama de red neuronal

Al igual que un perceptrón, la neurona sigmoid tiene entradas:  $x_1, x_2, x_3, \dots$  pero en lugar de ser solo 0 o 1, estas entradas también pueden tomar cualquier valor entre 0 y 1. Entonces 0,638... es una entrada válida para una neurona sigmoid. Al igual que un perceptrón, la neurona sigmoid tiene pesos para cada entrada:  $w_1, w_2, \dots$ , y un sesgo general:  $b$ . Pero la salida no es 0 o 1. En cambio, es:  $\sigma(w \cdot x + b) \leq$  umbral, donde  $\sigma$  se llama función sigmoid y esta nueva clase de neuronas se llama neuronas logísticas. Es útil recordar esta terminología, ya que estos términos los utilizan muchas personas que trabajan con redes neuronales. Sin embargo, nos quedaremos con la terminología sigmoid, y se define por:

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}$$

Las neuronas sigmoid parecen muy diferentes a los perceptrones. De hecho, existen muchas similitudes entre los perceptrones y las neuronas sigmoid, y la forma algebraica de la función sigmoidea resulta ser más un detalle técnico que una verdadera barrera para la comprensión.

### **1.1.5 TensorFlow sobre Phyton**

TensorFlow es una biblioteca de código abierto desarrollada por Google para realizar cálculos numéricos y construir modelos de aprendizaje automático, especialmente en redes neuronales. Está diseñada para ser flexible, escalable y fácil de usar, lo que la convierte en una de las bibliotecas más populares para el desarrollo de aplicaciones de aprendizaje automático y aprendizaje profundo. Para el objeto del proyecto será ejecutada sobre Python.

Algunas de las características clave de TensorFlow son:

- 6 **Grafos computacionales:** TensorFlow representa los cálculos como grafos computacionales, donde los nodos representan operaciones matemáticas y las aristas representan los datos multidimensionales (tensores) que fluyen entre ellas. Esto permite una mayor optimización y paralelización de los cálculos.
- 7 **Flexibilidad:** TensorFlow ofrece una amplia gama de herramientas y APIs que permiten construir y entrenar una variedad de modelos de aprendizaje automático, desde modelos simples hasta redes neuronales profundas complejas. Esta flexibilidad hace que se puedan ejecutar modelos sobre el Arduino.
- 8 **Escalabilidad:** TensorFlow puede ser ejecutado en una variedad de dispositivos, incluyendo CPUs, GPUs y TPUs (Unidades de Procesamiento Tensorial), lo que permite escalar el entrenamiento y la inferencia para manejar grandes conjuntos de datos y modelos complejos.
- 9 **Despliegue en producción:** TensorFlow proporciona herramientas para exportar modelos entrenados y desplegarlos en entornos de producción, permitiendo su integración en aplicaciones en tiempo real.
- 10 **Ecosistema amplio:** TensorFlow cuenta con un ecosistema activo de herramientas, bibliotecas y recursos de la comunidad que facilitan el desarrollo y la experimentación en aprendizaje automático.

Además, TensorFlow es compatible con Python, lo que significa que puedes utilizar todas sus funcionalidades desde un entorno de programación en Python, lo que lo hace aún más accesible y fácil de usar para la comunidad de desarrolladores.

### 1.1.6 Tensores.

Los tensores son **objetos matemáticos que almacenan valores numéricos** y que pueden tener **distintas dimensiones**. Así, por ejemplo, un tensor de 1D es un vector, de 2D una matriz, de 3D un cubo.

Tensores

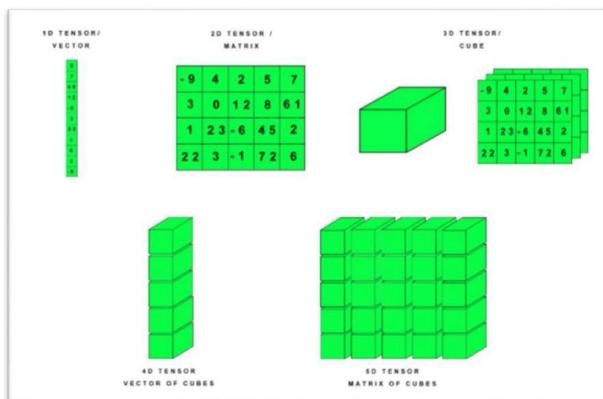


Figura 13. Representación grafica de Tensores

En Python, estos tensores normalmente se almacenan en lo que se conoce como NumPy arrays. NumPy es una de las librerías científicas de Python más se usan en cualquier entorno de trabajo con IA y sirve para manipular números.

A manera de ejemplo, pensemos en una lista de números, se puede “guardar” como un vector de 1D o quizás una lista de correos, con 10.000 usuarios, y 6 campos por cada uno (nombre, apellidos, dirección, ciudad, país, código postal) puede almacenarse en un tensor 2D con dimensiones (10.000, 6); pero si tenemos 10 listas de correo, tendríamos un juego de tensores 3D, que se representarían como (10, 10.000, 6), es decir (“número de listas”, “número de personas por lista”, “número de características por persona”).

Un caso típico de uso de tensores 3D es el análisis de tuits (anteriormente Twiter). Los tuits tienen 280 caracteres, y usan el estándar UTF-8. Aunque este estándar permite crear millones de caracteres, sólo nos interesan los primeros 128 que coinciden con los básicos del sistema ASCII. Por lo tanto, un solo tuit se podría encapsular en un vector 2D con la forma (280, 128). Pero en un *análisis de sentimiento* típico, no trabajamos con un único tuit. Si lo que queremos es analizar, por ejemplo, un cuerpo o paquete de 1 millón de tuits, los podemos “encapsular” en un tensor 3D de la siguiente forma (1.000.000, 280, 128). Por su parte, los tensores 4D, se suelen usar para el análisis de imágenes, como es el caso de nuestro proyecto de colorimetría.

Una imagen en color *RGB (Red, Green, Blue)*, con un tamaño de  $750 \times 750$  píxeles se puede almacenar en un tensor (750, 750, 3). Pero si lo que queremos es trabajar con un corpus de un millón de imágenes de gatos, con este tamaño y en este formato de color, usaremos un tensor 4D :(1.000.000, 750, 750, 3).

## 1.2 TML en Microporcesadores.

El aprendizaje automático (ML – machine learning) puede hacer que los microcontroladores sean accesibles para los desarrolladores que no tienen experiencia en desarrollo integrado, como es nuestro caso. Dentro del aprendizaje automático, existen técnicas que se pueden utilizar para adaptar modelos de redes neuronales a dispositivos con memoria limitada, como los microcontroladores de las placas Arduino. Uno de los pasos clave para poder realizar esto es: **la cuantificación de los pesos**, desde punto flotante hasta enteros de 8 bits. Esto también tiene el efecto de hacer que la inferencia sea más rápida de calcular y más aplicable a dispositivos con frecuencias de reloj más bajas.

Cuando se desarrollan redes neuronales modernas, el mayor desafío es lograr que funcionaran. Eso significaba que la precisión y la velocidad durante el entrenamiento eran las principales prioridades. Usar **aritmética de punto flotante**<sup>2</sup> era la forma más fácil de preservar la precisión y las GPU estaban bien equipadas para acelerar esos cálculos, por lo que es natural que no se prestara mucha atención a otros formatos numéricos.

Hoy en día, tenemos muchos modelos que se están implementando en aplicaciones comerciales. Las demandas computacionales de la capacitación crecen con el número de

---

<sup>2</sup> La representación de coma flotante es una forma de notación científica usada en las computadoras con la cual se pueden representar números reales extremadamente grandes y pequeños de una manera muy eficiente y compacta y con la que se pueden realizar operaciones aritméticas

investigadores, pero los ciclos necesarios para la inferencia<sup>3</sup> se expanden en proporción a los usuarios. Eso significa que la eficiencia de la inferencia pura se ha convertido en un tema candente para muchos equipos. Ahí es donde entra en juego la cuantización. Es un término general, la cuantificación cubre muchas técnicas diferentes para almacenar números y realizar cálculos con ellos en formatos más compactos que el punto flotante de 32 bits. Pares este proyecto el foco estará en un punto fijo de ocho bits.

El entrenamiento de redes neuronales se realiza aplicando muchos pequeños empujones a los pesos, y estos pequeños incrementos generalmente necesitan precisión de punto flotante para funcionar. **Tomar un modelo previamente entrenado y ejecutar la inferencia es muy diferente.**

Una de las cualidades mágicas de las redes profundas (Deep networks) es que tienden a afrontar muy bien, los niveles elevados de ruido en sus entradas. Por ejemplo, imagina que piensas como nosotros y deseas reconocer un objeto; pero en este ejemplo a través de una foto que acabas de tomar. Para esto la red tiene que ignorar todo el ruido del CCD<sup>4</sup>, los cambios de iluminación (es decir todo lo que este dentro de la foto y no forme parte del objeto que quieras reconocer) y otras diferencias no esenciales entre tu foto y los ejemplos de entrenamiento que ha visto antes. Esto para poder centrarse en lo importante: **las similitudes**. Esta capacidad es la razón por la cual en este proyecto no trabajamos a nivel de CCN para reconocimiento visual sino que usamos sensores de la placa Arduino para semejar una imagen controlada sin ruido de 1Megapixel. Esta misma propiedad de trabajar controlando el ruido de las entradas, significa que se deben tratar los cálculos de baja precisión como una fuente más de ruido y aún producen resultados precisos incluso con formatos numéricos que contienen menos información.

Otra característica que suma al uso de la cuantificación es que los modelos de redes neuronales pueden ocupar mucho espacio en el disco (más de 200 MB en formato flotante). Casi todo ese tamaño se ocupa con los pesos de las conexiones neuronales, ya que a menudo hay muchos millones de ellas en un solo modelo. Debido a que todos son números de punto flotante ligeramente diferentes, los formatos de compresión simples como zip no los comprimen bien. Sin embargo, están

---

<sup>3</sup> La inferencia de tipos en programación es un concepto clave en el mundo de la informática y el código. Se trata de un proceso que se lleva a cabo en muchos lenguajes para deducir automáticamente el tipo de datos de una variable o expresión sin que el programador tenga que especificarlo explícitamente

<sup>4</sup> El ruido del arco y el ruido de lectura son dos fuentes principales de ruido en CCD (Charge-coupled device (CCD)) y otras cámaras digitales así como para la microscopía. Aunque en los últimos años se han realizado grandes mejoras en la reducción del ruido oscuro del CCD a temperatura ambiente, enfriar el chip reduce aún más el ruido diez veces por cada disminución de 20 °C.

dispuestos en capas grandes y dentro de cada capa los pesos tienden a distribuirse normalmente dentro de un cierto rango, por ejemplo, de -3,0 a 6,0.

La motivación más simple para la cuantización es reducir el tamaño de los archivos **almacenando el mínimo y el máximo para cada capa** y luego comprimiendo cada valor flotante a un entero de ocho bits que represente el número real más cercano en un conjunto lineal de 256 dentro del rango.

Hagamos un ejemplo para aclarar el proceso. Por ejemplo, con un rango de -3,0 a 6,0, un byte 0 representaría -3,0, un 255 representaría 6,0 y 128 representaría aproximadamente 1,5. Esto significa que puedes obtener una reducción de disco en un 75% y luego volver a convertirlo a flotante después de cargarlo para que tu código de punto flotante existente pueda funcionar sin ningún cambio.

#### Proceso de Cuantificación

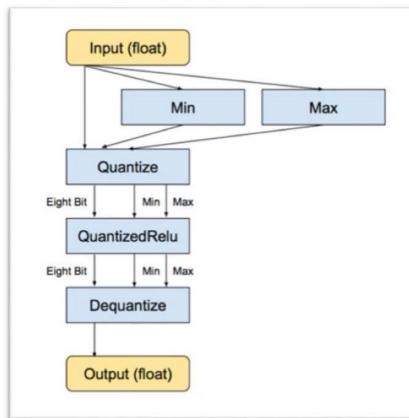


Figura 14. Modelo de bloques de un proceso de Cuantificación de 8 bits.

Otra razón para cuantificar es la reducir los recursos computacionales necesarios para realizar los cálculos de inferencia, y poder ejecutarlo completamente con entradas y salidas de ocho bits. Esto es mucho más difícil ya que requiere cambios en todos los lugares donde se realizan los cálculos, pero ofrece muchas ventajas potenciales. Como por ejemplo, obtener valores de ocho bits solo requiere el 25% del ancho de banda de la memoria de los flotantes, por lo que hará un uso

mucho mejor de las cachés y evitará cuellos de botella en el acceso a la RAM. Normalmente, también puede utilizar operaciones SIMD<sup>5</sup> que realizan muchas más operaciones por ciclo de reloj. En algunos casos, y dependiendo del hardware utilizado se podría tener acceso a un chip DSP<sup>6</sup> que también puede acelerar los cálculos de ocho bits, lo que puede ofrecer muchas ventajas.

Mover los cálculos a ocho bits no solo nos ayudará a ejecutar los modelos más rápido y utilizar menos energía (lo cual es especialmente importante en dispositivos móviles o placas Arduino) sino que también abre la puerta a muchos sistemas integrados que no pueden ejecutar código de punto flotante de manera eficiente, por lo que puede que se podrían habilitar muchas aplicaciones en el mundo de IoT a futuro sobre el proyecto actual.

Ahora bien cómo podríamos aplicar la cuantificación, a nuestro sistema de reconocimiento de imágenes? Lo bueno de los rangos mínimo y máximo es que a menudo se pueden calcular previamente. Los parámetros de peso son constantes conocidas en el momento de la carga, por lo que sus rangos también se pueden almacenar como constantes. A menudo conocemos los rangos de las entradas (por ejemplo, las imágenes suelen tener valores RGB en el rango de 0,0 a 255,0) y muchas funciones de activación también tienen rangos conocidos. Esto puede evitar tener que analizar las salidas de una operación para determinar el rango, lo que debemos hacer para operaciones matemáticas como la convolución o la multiplicación de matrices que producen resultados acumulados de 32 bits a partir de entradas de 8 bits.

Si está haciendo algún tipo de aritmética con entradas de 8 bits, naturalmente comenzará a acumular resultados con más de 8 bits de precisión. Si suma dos valores de 8 bits, el resultado necesita 9 bits. Si multiplicas dos números de 8 bits, obtendrás 16 bits en el resultado. Si suma una serie de multiplicaciones de 8 bits, como lo hacemos con la multiplicación de matrices, los resultados crecen más allá de los 16 bits, y el acumulador generalmente necesita al menos de 20 a 25 bits, dependiendo de la longitud de los productos escalares involucrados.

Esto puede ser un problema para nuestro enfoque de cuantización, ya que necesitamos tomar una salida que sea mucho más ancha que 8 bits y reducirla para alimentarla a la siguiente operación. Una forma de hacerlo para multiplicaciones de matrices sería calcular los valores de salida más

---

<sup>5</sup> SIMD (del inglés Single Instruction, Multiple Data - "una instrucción, múltiples datos") es una técnica empleada para conseguir paralelismo a nivel de datos. Los repertorios SIMD consisten en instrucciones que aplican una misma operación sobre un conjunto más o menos grande de datos.

<sup>6</sup> Un procesador de señales digitales o DSP (en inglés de digital signal processor) es un sistema basado en un procesador o microprocesador que posee un conjunto de instrucciones, un hardware y un software optimizados para aplicaciones que requieran operaciones numéricas a muy alta velocidad.

grandes y más pequeños posibles, suponiendo que todos los valores de entrada estuvieran en los extremos. Esto es seguro, ya que sabemos matemáticamente que ningún resultado puede quedar fuera de este rango, pero en la práctica la mayoría de los pesos y valores de activación están distribuidos de manera mucho más uniforme. Esto significa que el rango real de valores que vemos es mucho más pequeño que el teórico, por lo que si usáramos los límites mayores estaríamos desperdiando muchos de nuestros 8 bits en números que nunca aparecieron. En su lugar, utilizamos el operador: **QuantizeDownAndShrinkRange**<sup>7</sup> para tomar un tensor acumulado de 32 bits, analizarlo para comprender los rangos reales utilizados y reescalarlo para que el tensor de salida de 8 bits utilice ese rango de manera efectiva.

Hasta este momento hemos repasado la filosofía de TML la cual pudimos resumir en hacer aplicaciones con menos recursos, a saber: factores de forma más pequeños, menos energía y silicio de menor costo, de igual manera ampliaremos más sobre esto y la cuantificación de pesos más adelante en este documento.

También analizamos que ejecutar la inferencia en la misma placa que los sensores tiene beneficios en términos de privacidad y duración de la batería y la insolación de poder realizar y ejecutar aplicaciones independientemente de una conexión de red. El hecho de que tengamos el sensor de proximidad en la misma placa base significa que obtenemos una lectura instantánea de la profundidad de un objeto frente al tablero, en lugar de usar una cámara y tener que determinar si un objeto es de interés a través de la visión artificial.

Para nuestro proyecto, cuando el objeto está lo suficientemente cerca, **medimos el color**. Entonces utilizando el sensor RGB integrado (APDS9960), nuestra placa Arduino se puede ver como una cámara, a color, de 1pixel. Como hemos expuesto en este documento, este método tiene limitaciones pero también ventajas; pero para los fines del proyecto nos proporciona una forma rápida de clasificar objetos utilizando solo una pequeña cantidad de recursos.

Para realizar esto mostraremos una aplicación TML de uso simple pero completa (de extremo a extremo) y con referencias técnicas suficientes para reproducir el ejemplo a futuro. Esto sin considerar la existencia de conocimientos profundos en “Aprendizaje de Maquina” (ML) que puedan tener los lectores de este trabajo.

Este bloque de TML, se distribuye en los siguientes capítulos de este documento, como capítulos individuales por su contenido y complejida. Pero no se debe perder la perspectiva que

---

<sup>7</sup> Operador de TensorFlow, que convierte el tensor de 'entrada' cuantificado en una 'salida' de menor precisión, utilizando.

forman parte del sistema de bloques de TML, y abarca los procesos de:

- 1 **Transmisión y Visualización de datos así como la captura de datos de entrenamiento.**  
Que se utilizará la placa de Arduino y un código compilable en el software Arduino IDE, con énfasis en la utilización del sensor: APDS9960.
- 2 **Cuantificación y entrenamiento de nuestra red neuronal,** donde se iniciará la capacitación o aprendizaje para generar el ML y su modelo. Para esto utilizaremos Python y su librería de Tensor Flow.
- 3 **Clasificación de Datos.** Básicamente la salida de nuestro TML la generaremos como una TML entrenada con un modelo (patrón) que será cargado en el Arduino para la implementación del clasificador.

### 1.3 Transmisión y Visualización de los datos

#### 1.3.1 Configurando el IDE de Arduino

Lo primero necesario después de haber definido el hardware el método y las aplicaciones a utilizar es configurar la aplicación Arduino IDE que se utiliza para cargar modelos de inferencia a la placa y descargar los datos de entrenamiento. Para esto debemos descargar e instalar los controladores y software de nuestra placa así como las bibliotecas específicas en el IDE de Arduino.

Para lo cual podemos seguir los siguientes pasos:

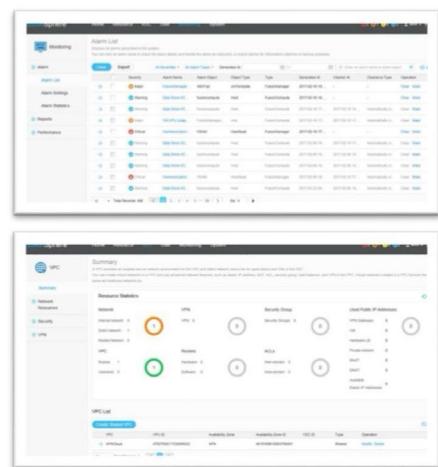
1. Descargar e instalar el IDE de Arduino desde: <https://arduino.cc/downloads>
2. Abrir la aplicación Arduino IDE, después de instalarla. Para nuestro proyecto hemos seleccionado la versión Windows. Que se ejecuta en un Máquina virtual (VM) dentro de un clúster personal de virtualización sobre Huawei FusionSphere.
3. En el menú Arduino IDE, se debe seleccionar: Herramientas > Placa > Administrador de placa y buscar “Nano BLE” y presione instalarlo.
4. Instalada la placa debemos agregar las bibliotecas principales para nuestro código de captura de colores. Hay que ir a Herramientas del administrador de bibliotecas > Administrar bibliotecas y buscar e instalar la biblioteca **Arduino\_TensorFlowLite**. Para nuestra placa en cuestión tuvimos inconvenientes y debimos instalar todas las disponibles.

- Luego debemos de dar de alta funcional nuestro sensor así que buscamos e instalamos la biblioteca: **Arduino\_APDS9960**

### Máquina Virtual del proyecto



The screenshot shows the Windows Device Manager interface. It includes sections for 'Acerca de' (About), 'Especificaciones del dispositivo' (Device specifications), and 'Especificaciones de Windows' (Windows specifications). The 'Especificaciones del dispositivo' section lists hardware details like the processor (Intel Core i5-8500 @ 3.00GHz) and RAM (8.00 GB). The 'Especificaciones de Windows' section shows the operating system as Windows 10 IoT Enterprise 22H2.



The screenshot shows the Huawei FusionSphere Resource Management interface. It displays two main windows: 'Resource Allocation' and 'Resource Statistics'. The 'Resource Allocation' window shows various resources assigned to VMs, including CPU, memory, and network. The 'Resource Statistics' window provides a summary of resource usage across different hosts.

Figura 15. Recursos Asignados a través del Huawei Fusionsphere.

- Hacemos la conexión por USB con la placa y la VM a través de un cable micro USB.
- Dentro de la interfaz se debe de elegir la placa, en: Herramientas > Placa > Arduino Nano 33 BLE. Así como elegir el puerto de comunicación en: Herramientas > Puerto > COM4 (Arduino Nano 33 BLE). En nuestro proyecto es el COM4.

## Ambiente de Trabajo



Figura 16. Selección de placas y puertos.

8. En nuestro caso y dado que debíamos establecer una ambiente colaborativo entre los integrantes del grupo, la VM fue publicada y accesible por medio de RDP.

## Método de Acceso

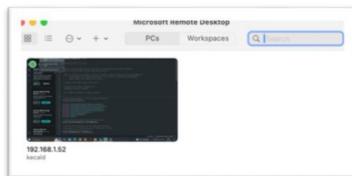


Figura 17. MS\_RD.

Como ha sido trazado el plan de trabajo debemos capturar algunos datos de entrenamiento. Y bueno que son estos datos pues algunos objetos en los cuales hagamos una medición de sus colores. Para capturar estos registros de datos del sensor de la placa Arduino, se va utilizar el mismo cable USB que utiliza para programar la placa.

Las placas Arduino ejecutan pequeñas aplicaciones o bocetos llamados: **sketches**. Estos se compilan a partir del código fuente Arduino en formato .ino y se programan en la placa utilizando “Arduino IDE” o “Arduino Create” su versión CloudWEB. Para la captura tomamos como ejemplo editable un sketch que forma parte del material de ayuda y tutoriales de uso de los sensores. Para nuestro proyecto el nombre del archivo es: **object\_color\_capture\_V1\_KECALD.ino** y tiene la

siguiente estructura:

```
#include <Arduino_APDS9960.h>

void setup() {
    Serial.begin(9600);
    while (!Serial) {};

    if (!APDS.begin()) {
        Serial.println("Error initializing APDS9960 sensor.");
    }

    // print the header
    Serial.println("Red,Green,Blue");
}

void loop() {
    int r, g, b, c, p;
    float sum;

    // wait for proximity and color sensor data
    while (!APDS.colorAvailable() || !APDS.proximityAvailable()) {}

    // read the color and proximity data
    APDS.readColor(r, g, b, c);
    sum = r + g + b;
    p = APDS.readProximity();

    // if object is close and well enough illuminated
    if (p == 0 && c > 10 && sum > 0) {

        float redRatio = r / sum;
        float greenRatio = g / sum;
        float blueRatio = b / sum;

        // print the data in CSV format
        Serial.print(redRatio, 3);
        Serial.print(',');
        Serial.print(greenRatio, 3);
        Serial.print(',');
        Serial.print(blueRatio, 3);
        Serial.println();
    }
}
```

La finalidad primordial de este código es utilizar un sensor de color y proximidad APDS9960 para detectar objetos cercanos y medir los niveles de color RGB (rojo, verde y azul) de los objetos detectados.

```
#include <Arduino_APDS9960.h>
```

Esta línea incluye la biblioteca para el sensor APDS9960. Esta biblioteca proporciona funciones para interactuar con el sensor.

```
void setup() {
    Serial.begin(9600);
    while (!Serial) {};

    if (!APDS.begin()) {
        Serial.println("Error initializing APDS9960 sensor.");
    }

    // print the header
    Serial.println("Red,Green,Blue");
}
```

La función **setup()** se ejecuta una vez al inicio del programa. En esta función es similar a un módulo global normalmente las variables globales y funciones elementales se ejecutan aquí. Recordemos que la base del Arduino IDE es C/C++. Es por esto que aquí:

1. Se inicia la comunicación serial a una velocidad de 9600 baudios.
2. Se espera hasta que la comunicación serial esté establecida.
3. Se inicializa el sensor APDS9960. Si no se puede inicializar el sensor, se imprime un mensaje de error.
4. Se imprime un encabezado indicando los datos que se imprimirán en serie, con un encabezado “Red, Green, Blue” dado que es un código RGB.

```

void loop() {
    int r, g, b, c, p;
    float sum;

    // wait for proximity and color sensor data
    while (!APDS.colorAvailable() || !APDS.proximityAvailable()) {}

    // read the color and proximity data
    APDS.readColor(r, g, b, c);
    sum = r + g + b;
    p = APDS.readProximity();

    // if object is close and well enough illuminated
    if (p == 0 && c > 10 && sum > 0) {

        float redRatio = r / sum;
        float greenRatio = g / sum;
        float blueRatio = b / sum;

        // print the data in CSV format
        Serial.print(redRatio, 3);
        Serial.print(',');
        Serial.print(greenRatio, 3);
        Serial.print(',');
        Serial.print(blueRatio, 3);
        Serial.println();
    }
}

```



La función **loop()** se ejecuta continuamente después de que la función **setup()** haya terminado de ejecutarse, genera un ciclo repetitivo para las diferentes lecturas. En esta función:

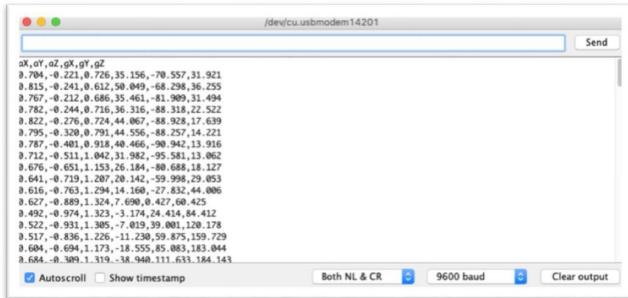
1. Se declaran variables para almacenar los valores de rojo (r), verde (g), azul (b), claridad (c) y proximidad (p) como enteros. También se declara una variable sum de tipo flotante para calcular la suma de los valores de rojo, verde y azul.
2. Se espera hasta que estén disponibles tanto los datos de color como de proximidad del sensor.
3. Se leen los datos de color y proximidad del sensor y se almacenan en las variables correspondientes.
4. Se calcula la suma de los valores de rojo, verde y azul.
5. Si el objeto está lo suficientemente cerca ( $p == 0$ ), iluminado adecuadamente ( $c > 10$ ) y se detecta algún color ( $sum > 0$ ), se calculan los ratios de rojo, verde y azul en relación

con la suma total de color.

6. Los ratios de color se imprimen en formato CSV (valores separados por comas) a través de la comunicación serial.

De esta forma con este código, que utiliza el sensor APDS9960 para medir los niveles de color de objetos cercanos y bien iluminados, la idea principal es imprimir los niveles de color, en la consola serie en formato CSV, si estos cumplen los criterios de proximidad (esto para evitar el ruido CCD) y luminosidad para una buena captura.

#### Salida del Monitor Serial del Arduino IDE



The screenshot shows the Arduino IDE's Serial Monitor window. The title bar reads "/dev/cu.usbmodem14201". The main area displays a series of floating-point numbers representing color ratios for various objects. At the bottom, there are several configuration buttons: 'Autoscroll' (checked), 'Show timestamp' (unchecked), 'Both NL & CR' (checked), '9600 baud' (selected), and 'Clear output'.

```
gX,gY,gZ,gR,gB,gA  
0.791,-0.221,0.206,35.156,-70.557,31.921  
0.815,-0.241,0.612,50.049,-68.298,36.255  
0.767,-0.212,0.686,35.461,-81.909,31.494  
0.782,-0.242,0.716,36.316,-88.318,22.522  
0.822,-0.276,0.724,44.067,-88.928,17.639  
0.742,-0.401,0.618,49.466,-88.257,14.221  
0.787,-0.401,0.518,49.466,-88.257,14.221  
0.712,-0.511,1.042,31.982,-95.581,13.062  
0.676,-0.651,1.153,26.184,-80.688,18.127  
0.641,-0.719,1.287,28.142,-59.998,29.053  
0.616,-0.763,1.299,14.168,-27.832,44.000  
0.627,-0.794,1.324,7.096,-11.111,.08.429  
0.644,-0.874,1.344,174,-24.414,12.412  
0.522,-0.931,1.305,-7.019,39.001,128.178  
0.517,-0.836,1.225,-11.239,59.875,159.729  
0.604,-0.694,1.173,-18.555,85.083,183.044  
0.684,-0.309,1.319,-38.940,111.633,184.143
```

Figura 18. Salida generalizada de un monitor serial utilizando el sensor APDS9960

Al final este proceso hemos inicializado nuestra infraestructura para ingresar mediciones de diferentes objetos de diferentes colores. Como el objeto de nuestro proyecto, hemos seleccionado 03 diferentes frutas: una manzana Gala, un kiwi y una remolacha.

## 1.4 Transmisión y Visualización de los datos

### 1.4.1 Transmisión de datos desde el monitor serial

Para cada medición inicializamos el dispositivo Arduino para que la salida del monitor serial, pudiera exportarse a un archivo CSV, que sería la base cromática de entrenamiento del TML. Dado el ambiente colaborativo a la VM que soporta la plataforma de Arduino, por lo que después de las mediciones realizamos un **copy-paste** a un archivo CSV, pero también ejecutamos una salida directa a través de una maquina Linux, de la siguiente forma:

```
$ cat /dev/cu.usbmodem[nnnnn] > sensorlog.csv
```

Al finalizar esta etapa hemos generado 03 archivos CSV, que cuentan con la medición RGB de nuestro 03 objetos de prueba, los cuales cuentan con el mismo formato:

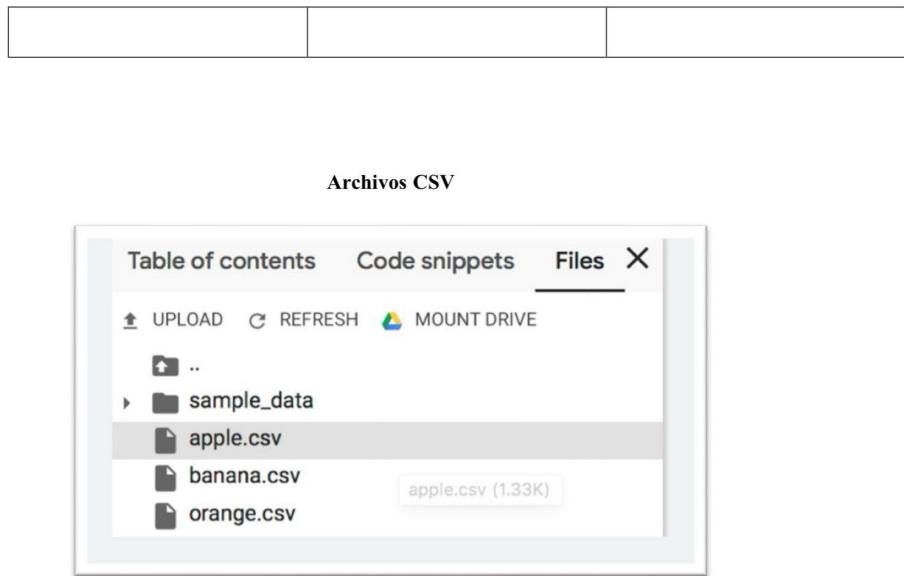


Figura 19. Salida generalizada de los archivos CSV

#### 1.4.2 Visualización grafica de la salida

Otro modulo importante del Arduino IDE para la visualización de la información de los sensores es la gráfica de los datos en tiempo real.

Grafica en tiempo real de los sensores

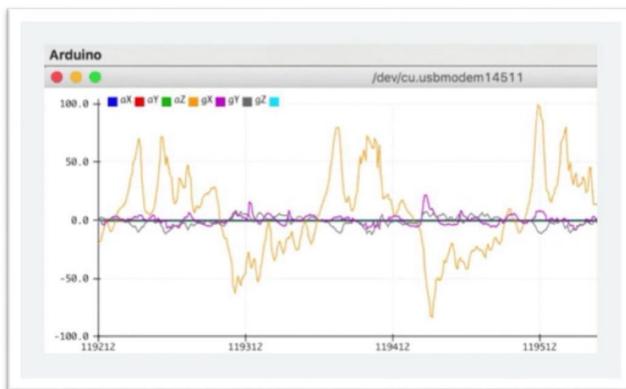


Figura 20. Salida generalizada de las lecturas del sensor para la Manzana

### 1.5 Captura de Datos de Entrenamiento y Entrenamiento TML

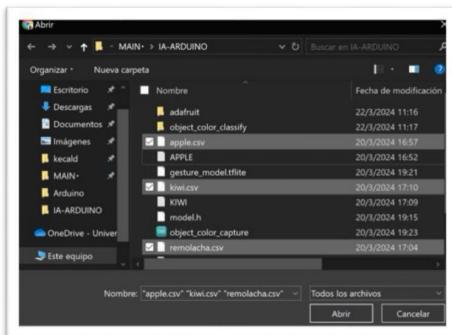
#### 1.5.1 Entrenamiento y modelo en Python con Tensorflow.

Para garantizar la colaboración de nuestro proyecto, utilizamos la herramienta: **Colab** de Google, en: <https://colab.research.google.com>. Aquí es donde en nuestro ambiente de trabajo de Python vamos a ejecutar el modelo de TML o de IA. Para lo cual ejecutamos los siguientes pasos:

1. Configurar el ambiente de trabajo en Python, mediante las siguientes instrucciones,

```
# Setup environment
!apt-get -qq install xxd
!pip install pandas numpy matplotlib
%tensorflow_version 2.x
!pip install tensorflow
```

2. Subimos los archivos CSV de entrenamiento, para cargarlos al ambiente de trabajo,



3. El código siguiente analizara, los archivos CSV y los transformara a un formato que se utilizará para entrenar la red neuronal completamente conectada. Este código está diseñado para preparar un conjunto de datos para entrenar un clasificador utilizando TensorFlow.

**Importación de bibliotecas:** Se importan las bibliotecas necesarias para el análisis de datos, visualización, manipulación de matrices, aprendizaje automático con TensorFlow y manejo de archivos y directorios.

```
python
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import tensorflow as tf
import os
import fileinput
```

**Configuración del entorno y versión de TensorFlow:** Se imprime la versión de TensorFlow que se está utilizando.

```
python
print(f"TensorFlow version = {tf.__version__}\n")
```

**Configuración de la semilla aleatoria:** Se establece una semilla aleatoria fija para garantizar la reproducibilidad de los resultados. La configuración de la semilla aleatoria es una práctica común en el aprendizaje automático y otras áreas donde se utilizan números aleatorios. Esta práctica se utiliza para garantizar que los resultados de los experimentos sean reproducibles, es decir, que al ejecutar el mismo código varias veces con la misma semilla aleatoria, se obtengan los mismos resultados. Al establecer una semilla aleatoria fija de esta manera, se asegura que los resultados del entrenamiento del modelo, la división de datos o cualquier otra operación que implique aleatoriedad se mantengan consistentes entre diferentes ejecuciones del código. Esto es especialmente útil cuando se está desarrollando y depurando un modelo, ya que permite obtener los mismos resultados cada vez que se ejecuta el código, lo que facilita la comparación y la depuración.

```
python
SEED = 1337
np.random.seed(SEED)
tf.random.set_seed(SEED)
```

**Lista de clases y codificación one-hot:** Se crea una lista vacía para almacenar las clases encontradas en los archivos CSV y se crea una matriz codificada one-hot para representar las clases. La lista de clases (**CLASSES**) es una estructura de datos que contiene los nombres de las clases que se deben clasificar. En el contexto de este código, se están procesando archivos CSV que contienen datos de diferentes clases. Por lo tanto, la lista de clases contendrá los nombres de estas clases encontradas en los archivos CSV.

Por ejemplo, si se tienen archivos CSV que contienen datos de "gatos", "perros" y "pájaros", entonces la lista de clases contendrá estos nombres en algún orden, como `["gatos", "perros", "pájaros"]`.

La codificación one-hot es una técnica utilizada para representar etiquetas categóricas (como las clases en un problema de clasificación) de una manera que sea más adecuada para el entrenamiento de modelos de aprendizaje automático.

En la codificación one-hot, cada clase se representa como un vector binario donde un solo elemento está marcado como 1 y todos los demás están marcados como 0. El índice del elemento

marcado como 1 corresponde al índice de la clase en la lista de clases.

Por ejemplo, supongamos que tenemos tres clases: "gatos", "perros" y "pájaros". La codificación one-hot de estas clases sería la siguiente:

1. "gatos" se representa como [1, 0, 0]
2. "perros" se representa como [0, 1, 0]
3. "pájaros" se representa como [0, 0, 1]

En el código, la codificación one-hot se realiza utilizando la función `np.eye(NUM_CLASSES)`, que crea una matriz identidad de tamaño `NUM_CLASSES` x `NUM_CLASSES`. Cada fila de esta matriz representa la codificación one-hot de una clase, donde el índice de la fila corresponde al índice de la clase en la lista de clases.

Por lo tanto, después de realizar esta codificación one-hot, la variable `ONE_HOT_ENCODED_CLASSES` contendrá una matriz donde cada fila representa la codificación one-hot de una clase, y esta matriz se utilizará más adelante para representar las salidas esperadas del modelo de clasificación.

```
python  
  
CLASSES = []  
ONE_HOT_ENCODED_CLASSES = np.eye(NUM_CLASSES)
```

**Exploración de archivos CSV en el directorio:** Se recorren los archivos en el directorio `"/content/"` (donde previamente los cargamos) y se agregan las clases encontradas a la lista `CLASSES`, después se ordenan alfabéticamente.

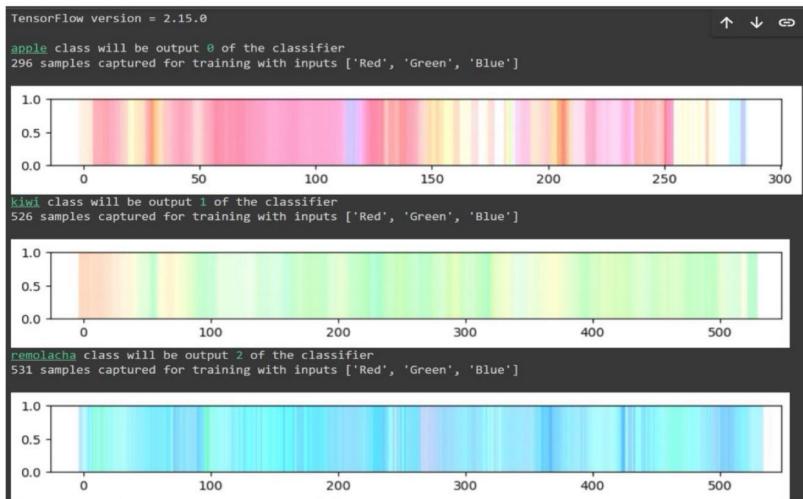
```
python  
  
for file in os.listdir("/content/"):   
    if file.endswith(".csv"):   
        CLASSES.append(os.path.splitext(file)[0])  
CLASSES.sort()
```

**Procesamiento de cada archivo CSV:** Para cada clase encontrada, se lee el archivo CSV correspondiente, se eliminan los valores NaN, y se calcula el número de grabaciones y se visualizan los datos de color en forma de gráfico.

```
python

for class_index in range(NUM_CLASSES):
    objectClass = CLASSES[class_index]
    df = pd.read_csv("/content/" + objectClass + ".csv", delimiter=';')
    # Procesamiento y visualización de datos
    ...
```

Obteniendo el siguiente resultado de salida:



**Preparación de datos para TensorFlow:** Se convierten las listas de entradas y salidas en matrices NumPy.

```
python

inputs = np.array(inputs)
outputs = np.array(outputs)
```

**Aleatorización y división de datos:** Se aleatorizan los datos y se dividen en conjuntos de entrenamiento, prueba y validación. La aleatorización y división de datos es una práctica común en el aprendizaje automático para preparar conjuntos de datos para el entrenamiento, la validación y la evaluación de modelos.

```
python
num_inputs = len(inputs)
randomize = np.arange(num_inputs)
np.random.shuffle(randomize)
inputs = inputs[randomize]
outputs = outputs[randomize]
TRAIN_SPLIT = int(0.6 * num_inputs)
TEST_SPLIT = int(0.2 * num_inputs + TRAIN_SPLIT)
inputs_train, inputs_test, inputs_validate = np.split(inputs, [TRAIN_SPLIT, TEST_SPLIT])
outputs_train, outputs_test, outputs_validate = np.split(outputs, [TRAIN_SPLIT, TEST_SPLIT])
```

Se calcula el número total de muestras en el conjunto de datos de entrada. Luego se crea un array de índices que representan las muestras en el conjunto de datos. Luego, se mezclan estos índices aleatoriamente utilizando la función **shuffle** de NumPy.

Una vez finalizado, los datos de entrada y salida se reordenan según el orden aleatorio generado en el paso anterior. Esto garantiza que las entradas y salidas correspondan correctamente después de la aleatorización.

Se calculan los tamaños para los conjuntos de entrenamiento, prueba y validación y luego se utilizan estos tamaños para dividir los datos de entrada y salida en tres conjuntos distintos utilizando la función **np.split**.

1. El 60% de los datos se utilizan para el entrenamiento (**inputs\_train, outputs\_train**).
2. El 20% de los datos se utilizan para la prueba (**inputs\_test, outputs\_test**).
3. El 20% restante se utiliza para la validación (**inputs\_validate, outputs\_validate**).

Esta división asegura que los conjuntos de datos de entrenamiento, prueba y validación estén equilibrados y que los datos se seleccionen aleatoriamente para cada conjunto, lo que es esencial para el entrenamiento y la evaluación efectivos de modelos de aprendizaje automático.

**Información sobre la preparación de datos:** Se imprime un mensaje indicando que la

preparación de datos ha finalizado.

```
python
```

```
print("Data set parsing and preparation complete.")
print("Data set randomization and splitting complete.")
```

### 1.5.2 Entrenamiento y modelo en Python con Tensorflow.

Este es el momento más crítico en el desarrollo de AI, en nuestro proyecto. Vamos a construir y entrenar un modelo de TensorFlow utilizando la API de alto nivel Keras.

La construcción de un modelo implica definir la arquitectura del mismo, es decir, la estructura de las capas y cómo están conectadas entre sí. Con TensorFlow y la API de Keras, esto se logra mediante la creación de una instancia de la clase Sequential o utilizando la API funcional de Keras para construir modelos más complejos. En la construcción del modelo, se agregan capas, se especifican las funciones de activación, las dimensiones de entrada y salida, entre otros detalles.

Una vez que se ha definido la arquitectura del modelo, se procede a entrenarlo utilizando datos de entrenamiento. Durante el entrenamiento, el modelo ajusta sus parámetros (pesos y sesgos) utilizando un algoritmo de optimización para minimizar una función de pérdida. Se realizan múltiples iteraciones a través de los datos de entrenamiento (llamadas épocas) para mejorar gradualmente el rendimiento del modelo.

La API de alto nivel de Keras en TensorFlow facilita tanto la construcción como el entrenamiento de modelos de aprendizaje profundo. Proporciona una interfaz intuitiva y fácil de usar que abstrae gran parte de la complejidad subyacente del aprendizaje profundo, permitiendo a los desarrolladores concentrarse en la creación y experimentación con diferentes arquitecturas de modelos. Esto se hace con el objetivo de que el modelo aprenda a realizar una tarea específica, como clasificación de imágenes, reconocimiento de voz o predicción de series temporales, entre otras.

```
# build the model and train it
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(8, activation='relu'))
model.add(tf.keras.layers.Dense(5, activation='relu'))
model.add(tf.keras.layers.Dense(NUM_CLASSES, activation='softmax'))
model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
history = model.fit(inputs_train, outputs_train, epochs=400, batch_size=4, validation_data=(inputs_validate, outputs_validate))
```

Una vez que se ha definido la arquitectura del modelo, se procede a entrenarlo utilizando datos de entrenamiento. Durante el entrenamiento, el modelo ajusta sus parámetros (pesos y sesgos) utilizando un algoritmo de optimización para minimizar una función de pérdida. Se realizan múltiples iteraciones a través de los datos de entrenamiento (llamadas épocas) para mejorar gradualmente el rendimiento del modelo.

### Construcción del modelo:

```
python
```

```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(8, activation='relu'))
model.add(tf.keras.layers.Dense(5, activation='relu'))
model.add(tf.keras.layers.Dense(NUM_CLASSES, activation='softmax'))
```

Lo primero es definir un modelo secuencial (Sequential), que es una pila lineal de capas. Este tipo de modelo es adecuado para una secuencia lineal de capas, donde la salida de una capa alimenta a la siguiente. Se agregan capas densamente conectadas (Dense) al modelo. Estas son capas totalmente conectadas donde cada nodo de la capa anterior está conectado a cada nodo de la capa siguiente.

La primera capa densa tiene 8 unidades (neuronas) y utiliza la función de activación ReLU (Rectified Linear Unit), debido a su eficacia en la mejora del rendimiento y la velocidad de convergencia. La segunda capa densa tiene 5 unidades y también utiliza la función de activación ReLU.

La última capa densa tiene un número de unidades igual a NUM\_CLASSES, que es el número de clases en el problema de clasificación. Esta capa utiliza la función de activación softmax, que es comúnmente utilizada en problemas de clasificación multiclas, ya que produce una distribución de probabilidad sobre las clases.

### Compilación del modelo:

```
python
```

```
model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
```

Se compila el modelo utilizando el optimizador 'rmsprop', que es un optimizador de gradiente estocástico muy utilizado en problemas de aprendizaje profundo. Se especifica la función de pérdida como 'mse' (Mean Squared Error), que es una función de pérdida comúnmente utilizada en problemas de regresión. Y por último se especifica la métrica de evaluación como 'mae' (Mean Absolute Error), que es una métrica comúnmente utilizada para evaluar la precisión de modelos de regresión.

### Entrenamiento del modelo:

```
python                                         Copy code
history = model.fit(inputs_train, outputs_train, epochs=400, batch_size=4,
validation_data=(inputs_validate, outputs_validate))
```

history = model.fit(inputs\_train, outputs\_train, epochs=400, batch\_size=4, validation\_data=(inputs\_validate, outputs\_validate))

Compilado el modelo procedemos con el entrenamiento utilizando los datos de entrenamiento (inputs\_train, outputs\_train), que vienen del proceso de preparación de la Data. Es importante definir el número de épocas, en nuestro caso: 400, lo que significa que el modelo se entrenará durante 400 iteraciones completas a través del conjunto de datos de entrenamiento.

Se debe de especificar el tamaño del lote como 4, lo que significa que se actualizarán los pesos del modelo después de cada lote de 4 muestras. Y por ultimo se valida el modelo, especificando los datos de validación (inputs\_validate, outputs\_validate) para evaluar el rendimiento del modelo después de cada época durante el entrenamiento. Esto permite monitorear si el modelo está sobre ajustando o generalizando correctamente durante el entrenamiento.

Al final del entrenamiento, el historial de entrenamiento se guarda en la variable history, que contiene información sobre la pérdida y las métricas de evaluación durante el proceso de entrenamiento. Este historial se puede utilizar para visualizar la evolución del rendimiento del modelo a lo largo del tiempo. La salida de nuestro sistema es la iteración de las 400 épocas:

```
Epoch 15/400
282/282 [=====] - 1s 2ms/step - loss: 0.1135 - mae: 0.2513 - val_loss: 0.1135 - val_mae: 0.2513
Epoch 14/400
282/282 [=====] - 1s 2ms/step - loss: 0.1067 - mae: 0.2398 - val_loss: 0.1007 - val_mae: 0.2320
Epoch 15/400
282/282 [=====] - 1s 2ms/step - loss: 0.0977 - mae: 0.2248 - val_loss: 0.0908 - val_mae: 0.2183
Epoch 16/400
282/282 [=====] - 1s 2ms/step - loss: 0.0889 - mae: 0.2099 - val_loss: 0.0824 - val_mae: 0.2034
Epoch 17/400
282/282 [=====] - 1s 2ms/step - loss: 0.0814 - mae: 0.1950 - val_loss: 0.0741 - val_mae: 0.1854
```

### Salida del sistema para la evaluación de Épocas.

```
Epoch 388/400
282/282 [=====] - 1s 2ms/step - loss: 0.0410 - mae: 0.0792 - val_loss: 0.0392 - val_mae: 0.0770
Epoch 389/400
282/282 [=====] - 1s 2ms/step - loss: 0.0413 - mae: 0.0788 - val_loss: 0.0390 - val_mae: 0.0777
Epoch 390/400
282/282 [=====] - 1s 2ms/step - loss: 0.0410 - mae: 0.0793 - val_loss: 0.0395 - val_mae: 0.0770
Epoch 391/400
282/282 [=====] - 1s 2ms/step - loss: 0.0411 - mae: 0.0794 - val_loss: 0.0391 - val_mae: 0.0775
Epoch 392/400
282/282 [=====] - 1s 2ms/step - loss: 0.0411 - mae: 0.0792 - val_loss: 0.0391 - val_mae: 0.0786
Epoch 393/400
282/282 [=====] - 1s 2ms/step - loss: 0.0412 - mae: 0.0791 - val_loss: 0.0398 - val_mae: 0.0804
Epoch 394/400
282/282 [=====] - 1s 2ms/step - loss: 0.0413 - mae: 0.0793 - val_loss: 0.0392 - val_mae: 0.0786
Epoch 395/400
282/282 [=====] - 1s 2ms/step - loss: 0.0411 - mae: 0.0783 - val_loss: 0.0408 - val_mae: 0.0821
Epoch 396/400
282/282 [=====] - 1s 2ms/step - loss: 0.0414 - mae: 0.0795 - val_loss: 0.0392 - val_mae: 0.0787
Epoch 397/400
282/282 [=====] - 1s 2ms/step - loss: 0.0411 - mae: 0.0790 - val_loss: 0.0393 - val_mae: 0.0769
Epoch 398/400
282/282 [=====] - 1s 2ms/step - loss: 0.0412 - mae: 0.0790 - val_loss: 0.0389 - val_mae: 0.0770
Epoch 399/400
282/282 [=====] - 1s 2ms/step - loss: 0.0412 - mae: 0.0788 - val_loss: 0.0390 - val_mae: 0.0777
Epoch 400/400
282/282 [=====] - 1s 2ms/step - loss: 0.0410 - mae: 0.0790 - val_loss: 0.0399 - val_mae: 0.0803
```

Figura 21. Salida generada de las 400 épocas de entrenamiento

#### 1.5.3 Agregar Datos de prueba y ejecución.

Después de que el modelo ha sido entrenado, se utiliza un conjunto de datos de prueba (que no ha sido visto por el modelo durante el entrenamiento) para evaluar su rendimiento. Esto se hace utilizando el método `evaluate` proporcionado por la API de Keras. Este método toma como entrada los datos de prueba (`inputs_test`, `outputs_test`) y devuelve la pérdida y las métricas especificadas durante la compilación del modelo.

Aquí además de presentar las predicciones estimadas, vamos a graficarlas para ver la curva de desviación que tendrá nuestro modelo de reconocimiento de imágenes recientemente entrenado. Para generar estas predicciones utilizaremos un modelo previamente entrenado y luego visualizaremos estas predicciones en conjunto con los valores reales del conjunto de datos de prueba.

#### Generación de predicciones:

```
python

predictions = model.predict(inputs_test)
```

Del código, se utiliza el método **predict** del modelo para generar predicciones utilizando los datos de entrada de prueba (inputs\_test). El resultado se almacena en la variable predictions.

#### Impresión de predicciones y valores esperados:

```
python
print("predictions =\n", np.round(predictions, decimals=3))
print("actual =\n", outputs_test)
```

Se imprimen las predicciones generadas por el modelo (redondeadas a tres decimales) y los valores reales del conjunto de datos de prueba. Esto permite comparar las predicciones del modelo con los valores reales y evaluar el rendimiento del modelo.

#### Visualización de las predicciones y valores reales:

```
python
plt.clf()
plt.title('Training data predicted vs actual values')
plt.plot(inputs_test, outputs_test[:, :3], 'b.', label='Actual')
plt.plot(inputs_test, predictions[:, :3], 'r.', label='Predicted')
plt.show()
```

Finalmente utilizando la librería: **Matplotlib** se traza un gráfico que muestre las predicciones generadas por el modelo (predictions) junto con los valores reales del conjunto de datos de prueba (outputs\_test). El gráfico tiene el título "Training data predicted vs actual values". Se utilizan puntos azules para representar los valores reales y puntos rojos para representar las predicciones del modelo.

La visualización no sólo nos permite una comparación directa entre las predicciones del modelo y los valores reales, esto facilita la evaluación del rendimiento del modelo. Generalmente, las predicciones deberían estar lo más cerca posible de los valores reales, lo que indicaría que el modelo ha aprendido con precisión la relación entre las características de entrada y las etiquetas de salida.

### Evaluación del Modelo predictivo

```
12/12 [=====] - 0s 3ms/step
predictions =
[[0.    0.005 0.995 0.    ]
[0.555 0.004 0.    0.441]
[0.872 0.021 0.    0.107]
...
[0.788 0.012 0.    0.2  ]
[0.369 0.002 0.    0.629]
[0.32  0.68  0.    0.   ]]
actual =
[[0. 0. 1. 0.]
[1. 0. 0. 0.]
[1. 0. 0. 0.]
...
[1. 0. 0. 0.]
[0. 0. 0. 1.]
[0. 1. 0. 0.]]
```

Figura 22. Comparativa de datos de prueba, sus predicciones y estado actual.

### Grafica de Rendimiento del Modelo predictivo

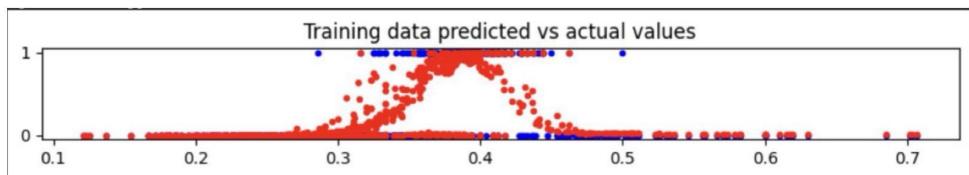


Figura 23. gráfico de las predicciones generadas por el modelo vs los valores reales del conjunto de datos de prueba (outputs\_test).

En este momento nuestro modelo esta listo para ser reproducido y ejecutado, pero nos queda un reto más. Y es que Arduino no entiende TensorFlow o Python directamente. Por lo tanto debemos pasar a convertir nuestro modelo en **TensorFlow Lite** que si es de manejo de la plataforma de Arduino. En síntesis estamos pasando de una IA de ML a una **TinyML (TML)**

#### **1.5.4 Convertir el modelo entrenado en su versión a TensorFlow Lite.**

La transformación del modelo incluye la conversión de formatos del archivo de salida a su versión: **TFLite**, así como la compresión del archivo.

**Convertir el modelo a TensorFlow Lite:**

```
python  
  
converter = tf.lite.TFLiteConverter.from_keras_model(model)  
tflite_model = converter.convert()
```

Para esto primero se debe crear una instancia del convertidor de TensorFlow Lite (TFLiteConverter) utilizando el método `from_keras_model`, que toma como entrada el modelo de TensorFlow Keras (`model`) que se desea convertir. Despues, se llama al método `convert()` del convertidor para convertir el modelo de TensorFlow a su versión correspondiente en TensorFlow Lite (**tflite\_model**) y finalmente se guarda en nuestro ambiente de directorio `/model/`

**Guardar el modelo en disco:**

```
python  
  
open("gesture_model.tflite", "wb").write(tflite_model)
```

Atraves de la función: `open()` abrimos un archivo llamado "gesture\_model.tflite" en modo de escritura binaria ("`wb`"). Que será el lugar donde trasladaremos el contenido del modelo de TensorFlow Lite (**tflite\_model**) en el archivo "gesture\_model.tflite" recién creado. Adicionalmente podemos verificar el tamaño del archivo final para ceriorarse de que puede ser agregado al Arduino.

```
python  
  
import os  
basic_model_size = os.path.getsize("gesture_model.tflite")  
print("Model is %d bytes" % basic_model_size)
```

Ahora el archivo "`gesture_model.tflite`" contiene el modelo convertido a TensorFlow Lite, lo que permite su implementación en nuestro Arduino. Además, logramos verificar el tamaño del

modelo para tener una idea del espacio que ocupará en la placa Arduino de destino y es de: **Model is 2496 bytes**. Finalmente vamos a codificar el modelo en un archivo de encabezado Arduino, con la finalidad de crear una matriz de bytes constantes que contienen el modelo TFLite. Y lo descargamos.

```
!echo "const unsigned char model[] = {" > /content/model.h
!cat gesture_model.tflite | xxd -i      >> /content/model.h
!echo "};;"                                >> /content/model.h

import os
model_h_size = os.path.getsize("model.h")
print(f"Header file, model.h, is {model_h_size:,} bytes.")
print("\nOpen the side panel (refresh if needed). Double click model.h to download the file.")
```

## 1.6 Programar el modelo TensorFlow Lite Micro en la placa Arduino

Finalmente, tomaremos el modelo que entrenamos anteriormente y lo compilaremos y lo cargaremos en nuestra placa Arduino usando Arduino IDE. Esto significa que nuestra placa deberá ejecutar otro código que esta vez en lugar de capturar datos para entrenamiento, generara instrucciones para analizar y predecir los objetos basados en su entrenamiento previo de colores.

Para esto vamos generar un nuevo archivo de extensión: .INO, para el cual utilizaremos la valiosa colaboración de: **Don Coleman, Sandeep Mistry**. Donde utilizamos su código de comienzo público para ejecutar este proceso. Para efectos del proyecto nuestro archivo es: **object\_color\_classify\_V2\_KECALD.INO**.

### Cargar de Sketch para modelo Predictivo

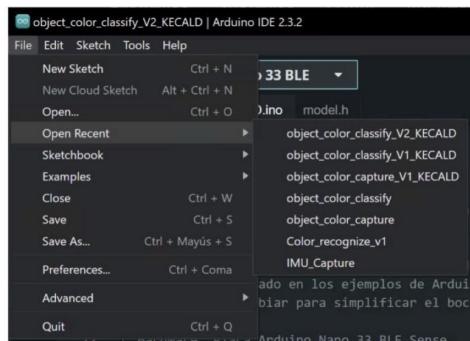


Figura 24. Selección en el Arduino IDE del nuevo Sketch de trabajo

Este nuevo archivo ejecuta las siguientes funciones para utilizar la placa Arduino y sus sensores como identificadores utilizando un modelo entrenado que se cargara junto con el archivo .INO.

Las primeras líneas son las inclusiones de las bibliotecas necesarias para TensorFlow Lite, así como las bibliotecas para el sensor APDS9960, como lo hicimos inicialmente pero ahora incluyendo el archivo de encabezado del modelo **model.h**.

```
cpp

#include <TensorFlowLite.h>
#include <tensorflow/lite/micro/all_ops_resolver.h>
#include <tensorflow/lite/micro/micro_error_reporter.h>
#include <tensorflow/lite/micro/micro_interpreter.h>
#include <tensorflow/lite/schema/schema_generated.h>
#include <tensorflow/lite/version.h>
#include <Arduino_APDS9960.h>
#include "model.h"
```

También y después de gestionar los errores en el proceso y que estos puedan emitir alguna alerta es importante definir punteros para almacenar el modelo, el intérprete, y los tensores de entrada y salida del modelo, a través del siguiente fragmento de código:

```
cpp

const tflite::Model* tflModel = nullptr;
tflite::MicroInterpreter* tflInterpreter = nullptr;
TfLiteTensor* tflInputTensor = nullptr;
TfLiteTensor* tflOutputTensor = nullptr;
```

Luego se debe gestionar el ambiente de trabajo y reservar un área de memoria para la arena de tensores, que se utilizará para almacenar los tensores de entrada y salida del modelo. Así como definir un arreglo de nombres de clases y se calcula el número de clases en el modelo

```
cpp

constexpr int tensorArenaSize = 8 * 1024;
byte tensorArena[tensorArenaSize];
```

```
cpp

const char* CLASSES[] = {
    "apple", // u8"\U0001F34E", // Apple
    "kiwi", // u8"\U0001F34C", // Banana
    "remolacha" // u8"\U0001F34A" // Orange
};

#define NUM_CLASSES (sizeof(CLASSES) / sizeof(CLASSES[0]))
```

Ahora empezamos a generar las entradas al modelo y su interacción con TensorFlow. La función **setup()** inicializa la comunicación serial, inicializa el sensor APDS9960 y configura el intérprete TensorFlow Lite.

```
cpp
Copy code

void setup() {
    Serial.begin(9600);
    while (!Serial) {};
    // Inicialización del sensor APDS9960
    if (!APDS.begin()) {
        Serial.println("Error initializing APDS9960 sensor.");
    }
    // Obtiene el modelo de TensorFlow Lite del archivo de encabezado
    tflModel = tflite::GetModel(model);
    // Crear un intérprete TensorFlow Lite y asignar memoria para los tensores
    tflInterpreter = new tflite::MicroInterpreter(tflModel, tflOpsResolver, tensorArena);
    tflInterpreter->AllocateTensors();
    // Obtener punteros para los tensores de entrada y salida del modelo
    tflInputTensor = tflInterpreter->input(0);
    tflOutputTensor = tflInterpreter->output(0);
}
```

Ahora debemos ingresar el proceso de clasificación para lo cual hemos pasar a la función **void loop()** la cual hemos segmentado funcionalmente en los siguientes procesos:

### Lectura de datos del sensor:

```
cpp

while (!APDS.colorAvailable() || !APDS.proximityAvailable()) {}

APDS.readColor(r, g, b, c);
p = APDS.readProximity();
sum = r + g + b;
```

- El bucle **while** espera hasta que estén disponibles los datos del sensor de color y proximidad.
- Después de que los datos estén disponibles, se leen los valores de color (rojo, verde, azul) y proximidad del sensor APDS9960.

### Verificación de condiciones:

```
cpp

if (p == 0 && c > 10 && sum > 0) {
```

- Se verifica si hay un objeto cerca y bien iluminado. Esto se determina verificando si la proximidad es cero (indicando que hay un objeto cercano), si la luz es suficiente (se asume que la luz es suficiente si el valor de **c** es mayor que 10, esto sería aproximadamente: 1700lumnes) y si la suma de los valores de color es mayor que cero (para evitar divisiones por cero).

### Normalización de valores de color y asignación a los tensores de entrada:

```
cpp

float redRatio = r / sum;
float greenRatio = g / sum;
float blueRatio = b / sum;
tflInputTensor->data.f[0] = redRatio;
tflInputTensor->data.f[1] = greenRatio;
tflInputTensor->data.f[2] = blueRatio;
```

- Los valores de color se normalizan dividiéndolos por la suma de los valores de color para obtener las proporciones de rojo, verde y azul en la entrada del modelo de TensorFlow Lite. Estos valores normalizados se asignan a los tensores de entrada del modelo.

#### Inferencia del modelo:

```
cpp

TfLiteStatus invokeStatus = tflInterpreter->Invoke();
if (invokeStatus != kTfLiteOk) {
    Serial.println("Invoke failed!");
    while (1);
    return;
}
```

- Se realiza la inferencia invocando el intérprete TensorFlow Lite.

#### Impresión de resultados de la inferencia:

```
cpp

for (int i = 0; i < NUM_CLASSES; i++) {
    Serial.print(CLASSES[i]);
    Serial.print(" ");
    Serial.print(int(tflOutputTensor->data.f[i] * 100));
    Serial.print("%\n");
}
Serial.println();
```

- Los resultados de la inferencia, que son las probabilidades de cada clase, se imprimen en la salida del puerto serial.

**Espera hasta que el objeto se aleje:** Se espera hasta que el objeto se aleje antes de repetir el proceso de inferencia.

```
while (!APDS.proximityAvailable() || (APDS.readProximity() == 0)) {}
```

Con el archivo finalizado dentro del Arduino IDE, debemos agregar el modelo generado; nuestro archivo: **model.h** (generado desde Colab). Esto con el fin de contar con los encabezados en un modelo cuantificado (según revisamos en este documento) y comprimido a una expresión de TML. Internamente un archivo de este tipo se ve en un editor de texto según las figura #26.

### Cargar de Model.h

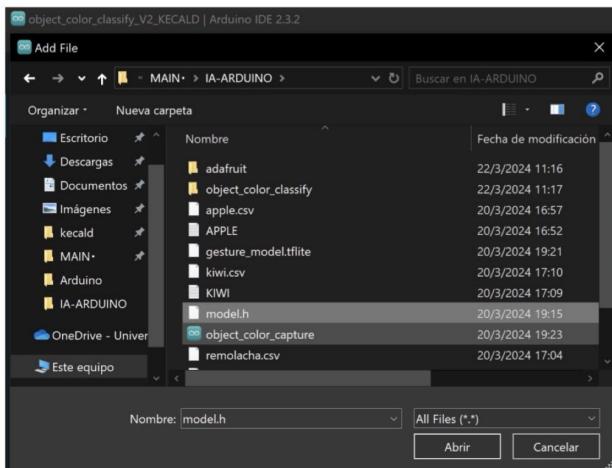


Figura 25. Arduino IDE, inclusión de archivos de ejecución de Sketch

### Estructura interna de un archivo .H

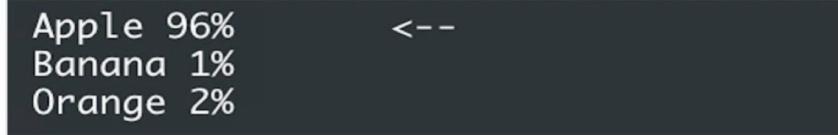
```
1 const unsigned char model[] = {  
2     0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x00, 0x00, 0x12, 0x00,  
3     0x1c, 0x00, 0x04, 0x00, 0x08, 0x00, 0x0c, 0x00, 0x10, 0x00, 0x14, 0x00,  
4     0x00, 0x00, 0x18, 0x00, 0x12, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00,  
5     0x70, 0x07, 0x00, 0x00, 0x10, 0x00, 0x00, 0x1c, 0x00, 0x00, 0x00,
```

Figura 26. Editor de texto de un modelo de encabezados.

## **Etapa Final**

Finalmente nos queda la parte final del proyecto que es compilar y ejecutar. Al compilar en Arduino debemos posteriormente enviar la aplicación a la placa para su ejecución. El dispositivo deberá aplicar la nueva programación y empezar a detectar los objetos entrenados ejecutando una pequeña red neuronal internamente para que en su Salida de monitor (dentro del Arduino IDE) se puedan visualizar las predicciones de nuestro proyecto en la identificación de las frutas.

A final del proceso nuestra salida debe visualizarse de la siguiente forma, como una probabilidad en la detección a través de un modelo de aprendizaje aplicado en tiempo real en un dispositivo de bajo consumo.



Apple 96%      <--  
Banana 1%  
Orange 2%

## **Capítulo 3. Conclusiones y Recomendaciones**

### **3. Conclusiones**

El proyecto ha demostrado con éxito la integración de tecnologías de machine learning y hardware de bajo costo en la forma de Arduino para la clasificación de frutos en la agricultura. El modelo de red neuronal implementado, entrenado con TensorFlow y optimizado para ejecutarse en plataformas de microcontroladores mediante TinyML, ha mostrado un notable desempeño en la identificación de colores de frutos. Esta capacidad de clasificación automatizada es crítica para optimizar el manejo agrícola y potencialmente reducir la carga laboral, haciendo el proceso más eficiente y menos susceptible a errores humanos.

La autonomía del sistema y su bajo consumo energético, gracias a la utilización de componentes electrónicos especializados y técnicas de cuantificación de modelos, permite su uso en entornos rurales donde las conexiones de energía y datos son limitadas. Esto no solo resuelve un problema técnico sino que también aumenta la accesibilidad de la tecnología para agricultores en regiones menos desarrolladas.

El proyecto ha revelado la escalabilidad del modelo de machine learning en aplicaciones agrícolas. La implementación exitosa en una plataforma como Arduino, que es relativamente limitada en términos de capacidad de procesamiento y memoria, demuestra que la tecnología puede adaptarse y escalar a otros dispositivos IoT más robustos o complejos. Esto abre puertas para futuras aplicaciones en una variedad más amplia de entornos agrícolas y industriales, lo que puede facilitar la adopción de IA en sectores donde tradicionalmente ha sido limitada.

La implementación de este sistema no solo mejora la eficiencia de los procesos agrícolas, sino que también tiene el potencial de impactar positivamente en las economías locales. Al reducir la carga de mano de obra y aumentar la producción a través de decisiones agrícolas más precisas y automatizadas, puede contribuir significativamente a la sostenibilidad y rentabilidad de las granjas, especialmente en comunidades rurales donde la agricultura constituye una parte principal de la economía.

No obstante, el sistema enfrenta desafíos inherentes a la variabilidad ambiental y la diversidad biológica de los cultivos que afectan la precisión del modelo. Aunque los resultados iniciales son prometedores, la adaptabilidad y escalabilidad del sistema bajo diferentes condiciones de cultivo todavía requieren investigación y desarrollo adicionales para asegurar su eficacia y robustez a largo plazo.

#### **4. Recomendaciones para su correcto uso y aplicación.**

Para avanzar hacia una solución más robusta y adaptable, se recomienda ampliar significativamente la base de datos de entrenamiento del modelo. Incluir una variedad más amplia de frutos y condiciones ambientales ayudará a mejorar la capacidad del modelo para generalizar y funcionar eficazmente en diferentes escenarios agrícolas. Este enriquecimiento de datos podría ser facilitado por colaboraciones con agricultores y centros de investigación agrícola que pueden proporcionar acceso a variedades más amplias de datos de cultivos.

Es crucial continuar con la optimización del modelo para asegurar que el sistema sea no solo preciso sino también eficiente en términos de recursos computacionales. Experimentar con diferentes arquitecturas de red neuronal y técnicas avanzadas de optimización puede contribuir a modelos más ligeros y rápidos que sean ideales para implementaciones en campo.

Asimismo, se sugiere realizar pruebas piloto en condiciones de campo reales para observar el comportamiento del sistema en entornos operativos naturales. Estas pruebas proporcionarán datos valiosos que pueden utilizarse para afinar el modelo y ajustar los parámetros del sistema para mejorar su rendimiento y fiabilidad.

Se recomienda el desarrollo de múltiples prototipos y la realización de iteraciones continuas del sistema. Cada versión debe ser probada bajo diferentes condiciones para asegurar que el sistema es robusto y confiable. Además, las iteraciones permitirán integrar retroalimentación directa de los usuarios finales, asegurando que el sistema final sea no solo tecnológicamente avanzado sino también práctico y fácil de usar en el día a día agrícola.

Formar alianzas con universidades y centros de investigación puede proporcionar acceso a recursos valiosos, como talento humano especializado y tecnologías avanzadas. Estas colaboraciones pueden facilitar estudios más profundos sobre la aplicación del machine learning en la agricultura y acelerar la innovación y mejora del sistema actual. Además, trabajar con instituciones educativas puede ayudar en la formación de los futuros agricultores y técnicos sobre cómo utilizar y mantener tecnologías avanzadas en la agricultura.

Finalmente, el desarrollo de interfaces de usuario intuitivas y accesibles facilitará la adopción de esta tecnología por parte de los agricultores. Estas interfaces deben permitir a los usuarios configurar fácilmente el sistema, entender los datos procesados y tomar decisiones informadas basadas en la información proporcionada por el dispositivo. La participación de los usuarios finales en el proceso de diseño y desarrollo también garantizará que el producto final sea más acorde con sus necesidades y expectativas.

## Bibliografía

**Oppenheim, A. V., & Schafer, R. W. (2010).** Discrete-time signal processing. Pearson Education.

**Bishop, C. M. (2006).** Pattern Recognition and Machine Learning. Springer.

**Hastie, T., Tibshirani, R., & Friedman, J. (2009).** The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer.

**LeCun, Y., Bengio, Y., & Hinton, G. (2015).** "Deep Learning". Nature, 521(7553), 436-444.

**Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014).** "Generative Adversarial Nets". Proceedings of the International Conference on Neural Information Processing Systems (NIPS).

**Goodfellow, I., Bengio, Y., & Courville, A. (2016).** Deep Learning. MIT Press. [Referencia a técnicas avanzadas de machine learning aplicables a proyectos similares.]

**TensorFlow Official Website:** <https://www.tensorflow.org/>

Recursos para implementar y optimizar modelos de machine learning.

**Arduino Official Website:** <https://www.arduino.cc/>

Documentación y tutoriales sobre la utilización de placas Arduino en proyectos de IoT.

**Scikit-Learn Official Documentation.** <https://scikit-learn.org>

Proporciona documentación y tutoriales sobre la implementación de técnicas de machine learning que pueden ser útiles para la construcción y optimización del modelo.

**International Conference on Learning Representations (ICLR)**

Los trabajos presentados en esta conferencia suelen estar a la vanguardia de las técnicas de aprendizaje profundo y podrían proporcionar ideas avanzadas para la mejora continua del proyecto.

**Neural Information Processing Systems (NeurIPS)** <https://neurips.cc/>

Una de las principales conferencias en el campo de la inteligencia artificial que aborda desde teoría fundamental hasta aplicaciones prácticas, ideal para seguir las últimas tendencias.