UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Investigación y desarrollo de aplicaciones típicas de aprendizaje automático en microcontroladores mediante el flujo de trabajo de *Tiny Machine Learning*

Trabajo de graduación presentado por Luis Fernando De León Garcia
para optar al grado académico de Licenciado en Ingeniería Mecatrónica

Guatemala,

2022

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Investigación y desarrollo de aplicaciones típicas de aprendizaje automático en microcontroladores mediante el flujo de trabajo de *Tiny Machine Learning*

Trabajo de graduación presentado por Luis Fernando De León Garcia
para optar al grado académico de Licenciado en Ingeniería Mecatrónica

Guatemala,

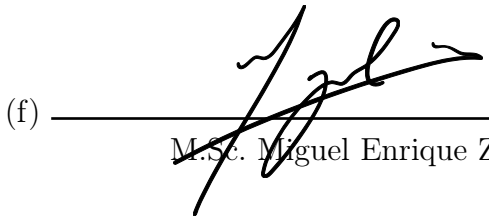
2022

Vo.Bo.:

(f) 

M.Sc. Miguel Enrique Zea Arenales

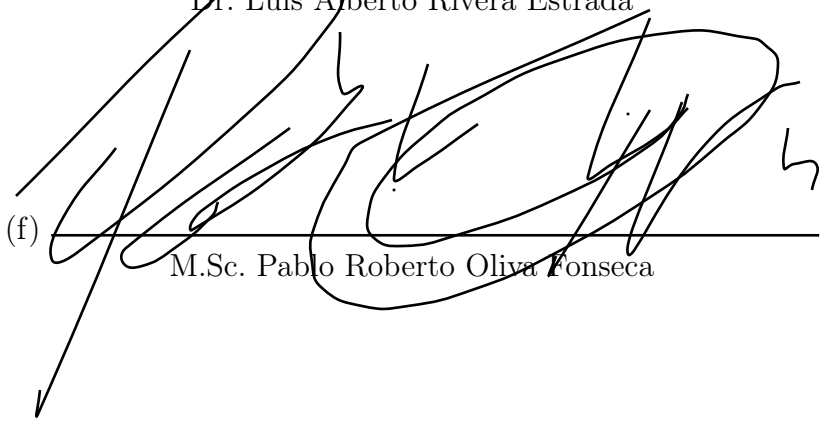
Tribunal Examinador:

(f) 

M.Sc. Miguel Enrique Zea Arenales

(f) 

Dr. Luis Alberto Rivera Estrada

(f) 

M.Sc. Pablo Roberto Oliva Fonseca

Fecha de aprobación: Guatemala, 5 de enero de 2022.

Prefacio

Muchas personas han contribuido al proceso y culminación de mi carrera universitaria. Este solo es el último paso de la trayectoria, que demuestra mi crecimiento en conocimientos, mi trabajo de excelencia y mi forma de aplicar lo aprendido en el planeta en que vivimos.

Agradezco a las personas que me acompañaron en todo el trayecto, principalmente a Dios y a mi familia por permitirme tener ejemplos de superación, lucha y dedicación en cada uno de sus integrantes. Además, agradezco a mis compañeros por brindarme buenos momentos durante mi carrera universitaria, nunca serán olvidados. También agradezco a mi asesor, Miguel Zea, por siempre brindar ideas para mejorar mi trabajo y exigir lo mejor de mí para lograr buenos resultados.

Siempre estaré agradecido a la Universidad del Valle de Guatemala por creer en mí y brindarme la oportunidad de asistir a esta casa de estudios a través de una beca.

Es a ustedes a quienes dedico este trabajo.

Índice

Lista de figuras

Lista de cuadros

Resumen

Este trabajo se desarrolló como fase inicial de un proyecto que busca la implementación de aprendizaje automático en microcontroladores por medio de redes neuronales. Este proyecto se dividió en tres aplicaciones principales que buscan presentar una introducción práctica a la nueva tendencia conocida como *Tiny ML*. El objetivo principal del trabajo es indagar y desarrollar aplicaciones típicas de aprendizaje automático a través del flujo de trabajo de *Tiny Machine Learning* en microcontroladores. Las aplicaciones presentadas en este trabajo muestran diferentes escenarios de aplicación de aprendizaje automático en microcontroladores con complejidad en aumento conforme se avanza en la lectura. Cada aplicación contó con experimentos que ayudan a demostrar su funcionamiento, aunque se abordó con más detalle el desarrollo e implementación de los modelos en los dispositivos.

La primera aplicación consistió en desarrollar un modelo de regresión simple e implementarlo en los dispositivos Arduino Nano 33 BLE Sense y Espressif ESP32 DevKitC. Se realizó el desarrollo y conversión del modelo para ejecución en los microcontroladores, así como gráficos de los resultados obtenidos en los dispositivos mediante comunicación serial y de forma visual mediante un LED. La aplicación mostró resultados satisfactorios al brindar buenas estimaciones para los datos de entrada.

La segunda aplicación consistió en el desarrollo de un modelo de reconocimiento de patrones utilizando redes neuronales e implementarlo en un Arduino Nano 33 BLE Sense. El desarrollo de esta aplicación fue enfocado en la recolección de datos e interacción del sensor con el modelo. La aplicación mostró buena ejecución en el dispositivo al obtener inferencias adecuadas acerca de los datos de entrada obtenidos con una Unidad de Medición Inercial (IMU).

La tercera aplicación consistió en utilizar el algoritmo YOLO para realizar detección y clasificación de objetos en la toma de un módulo de visión JeVois A33. Los datos de salida del algoritmo fueron utilizados en un Arduino Micro para ejecutar un controlador que actúa servomotores para centrar el objeto detectado en el plano de imagen con un brazo robótico. El modelo YOLO fue entrenado para reconocer un objeto y la aplicación mostró respuestas satisfactorias al ejecutar el controlador con base en las inferencias del modelo, aunque no pudo ser implementado de forma adecuada en el módulo de visión.

Abstract

This work was developed as the initial phase of a project that seeks the implementation of machine learning in microcontrollers through neural networks. This project is divided into three main applications that seek to present a practical introduction to the new trend known as Tiny ML. The main objective of the work is to investigate and develop typical machine learning applications through the Tiny Machine Learning workflow in microcontrollers. The applications presented in this work show different scenarios of application of machine learning in microcontrollers with increasing complexity as the reading progresses. Each application had experiments that help to demonstrate its operation, although the development and implementation of the models in the devices was addressed in more detail.

The first application consisted of developing a simple regression model and implementing it on the Arduino Nano 33 BLE Sense and Espressif ESP32 DevKitC devices. The development and conversion of the model was carried out for execution in the microcontrollers, as well as graphics of the results obtained in the devices through serial communication and in a visual way through an LED. The application showed satisfactory results by providing good predictions for the input data.

The second application consisted of developing a pattern recognition model using neural networks and implementing it in an Arduino Nano 33 BLE Sense. The development of this application was focused on data collection and interaction of the sensor with the model. The application showed good performance on the device by obtaining adequate inferences from the input data obtained with an Inertial Measurement Unit (IMU).

The third application consists of using the YOLO algorithm to perform object detection and classification in the shot of a JeVois A33 vision module. The algorithm's output data was used in an Arduino Micro to run a controller that actuated servo motors to center the detected object in the image plane with a robotic arm. The YOLO model was trained to recognize an object and the application showed satisfactory responses when executing the controller based on the model's inferences, although it could not be properly implemented in the vision module.

CAPÍTULO 1

Introducción

Este proyecto busca presentar una introducción práctica a la implementación de aprendizaje automático en microcontroladores utilizando la rama del aprendizaje profundo por medio de redes neuronales. Esta es la primera fase de esta rama de investigación, ya que esta tendencia es nueva y se busca implementarla lo antes posible para no quedar rezagados en su exploración y utilización. La investigación se divide en tres etapas, siendo la primera el desarrollo de una aplicación de regresión a través de redes neuronales en dos dispositivos distintos, donde se utilizan datos auto-generados para su entrenamiento con el fin de familiarizar al desarrollador con el flujo de trabajo.

La segunda parte consiste en el desarrollo de una aplicación de reconocimiento de patrones utilizando redes neuronales para inferir sobre datos del sensor integrado en un microcontrolador. Esta parte también aborda la recolección de datos, así como la implementación del sistema en el dispositivo. Con esta aplicación, se busca realizar reconocimiento de gestos o movimientos realizados con el dispositivo a través de la lectura de datos de la Unidad de Medición Inercial (IMU, por sus siglas en inglés) que contiene el dispositivo utilizado. Esta aplicación se aborda como segundo punto dado el nivel de complejidad mayor comparado con lo desarrollado inicialmente.

La tercera y última parte consiste en el desarrollo de un sistema de visión por computadora. Se utiliza el algoritmo YOLO para realizar detección, clasificación y encuadre de objetos en un módulo de visión dedicado a obtener imágenes y correr el algoritmo. Los resultados son interpretados por un Arduino para ejecutar un sistema de control que actúa servomotores para centrar el objeto detectado en el plano de imagen con un brazo robótico.

En el desarrollo de las aplicaciones mencionadas se utilizan herramientas como Google Colaboratory y la librería TensorFlow para facilitar la ejecución de código y desarrollo de modelos de aprendizaje automático. Se utilizan los dispositivos Arduino Nano 33 BLE Sense, Espressif ESP32 DevKitC y el módulo de visión JeVois A33 como principales herramientas.

CAPÍTULO 2

Antecedentes

2.1. Historia de aprendizaje automático y aprendizaje profundo

Las herramientas llamadas aprendizaje automático y aprendizaje profundo han sido conocidas y desarrolladas por los científicos ya por un largo tiempo. Sus inicios se remontan a décadas atrás, alrededor de los años sesenta. Si bien el ámbito teórico comenzó un crecimiento significativo, fue difícil encontrar implementaciones de la teoría en sus inicios ya que son escasas. Esto se debió a que las necesidades de capacidad computacional y de procesamiento no eran cumplidas por los equipos de cómputo de aquella época. Con el paso del tiempo se ha dado la evolución de los equipos de cómputo a versiones más capaces y con ello ha venido el auge de la utilización de estas herramientas. A partir del año 2010, la utilización de modelos de aprendizaje automático ha sido cada vez más común, incluso llegando a utilizar el nombre de la tecnología para mercadeo de productos, haciéndolos más llamativos. En la actualidad, la rama de investigación de aprendizaje automático se dedica a la optimización de modelos y a la búsqueda de nuevos campos en los que se pueda utilizar esta herramienta [1].

2.2. Internet de las cosas (IoT)

El internet de las cosas es una tendencia tecnológica que muestra el potencial de la tecnología al estar interconectada. Esta tendencia es parte del proceso de transformación digital en el que se está inmerso y se basa en la interconexión de los dispositivos a internet. Los objetos conectados a la red normalmente cuentan con capacidad de cómputo, almacenamiento y comunicación, de forma que se facilita el procesamiento, recopilación y distribución de la información a través de la red. La información que se transmite en la red puede ser utilizada para tomar decisiones u optimizar procesos, además de comunicación entre dispositivos [2]. Esta tendencia puede ser potenciada al ser combinada con aprendizaje automático, ya que

los dispositivos puede aprender de su entorno a través de los datos y experiencia, mientras el internet de las cosas trata con la interacción de los objetos con el internet, formando un sistema más completo. Entre los beneficios de combinar el internet de las cosas y aprendizaje automático se puede encontrar: aumento de la eficiencia operacional, mejor manejo de riesgos y desencadenamiento de productos y servicios nuevos y mejorados [3].

2.3. Desarrollo de hardware dedicado para aprendizaje automático

La ejecución de modelos de aprendizaje automático, en especial los modelos de aprendizaje profundo, involucran una gran cantidad de operaciones del procesador y alta transferencia de memoria. A partir de esta característica han sido creados aceleradores de aprendizaje automático, que es hardware diseñado para facilitar el procesamiento de información y ejecución de modelos de aprendizaje automático. Este tipo de hardware presenta diseños que permiten un procesamiento más eficiente y rápido, así como capacidades especiales que facilitan la ejecución de instrucciones específicas necesarias en el campo del aprendizaje profundo [4]. En el mercado se pueden encontrar los siguientes ejemplos.

2.3.1. Chip M1 de Apple

M1 es un chip formado por 16 mil millones de transistores e integra la unidad central de procesamiento (CPU), unidad de procesamiento gráfico (GPU) y el motor neural, entre otros. Este chip cuenta con 16 núcleos en su motor neural que habilita al chip a ejecutar 11 millones de millones de operaciones por segundo y permite que a través de software se puedan mejorar detalles en una cantidad mínima de tiempo. El chip M1 brinda a Apple una capacidad de ser 15 veces más veloz en sus capacidades de aprendizaje automático [5].

Figura 1: Chip M1 de Apple [5].

2.3.2. Unidad de procesamiento gráfico Tesla V100 de NVIDIA

NVIDIA Tesla V100 es una unidad de procesamiento gráfico con núcleos Tensor. Fue diseñada para acelerar tareas de inteligencia artificial que involucran aprendizaje automá-

tico, computación de alto rendimiento, ciencia de datos y gráficos. Su arquitectura brinda 640 núcleos Tensor, de forma que se vuelve más práctica el entrenamiento de modelos de aprendizaje automático cada vez más complejos. Además, al momento de utilizar los modelos para hacer inferencia, esta arquitectura permite el máximo rendimiento al hacerla hasta 24 veces más rápido que un servidor de CPU [6].

Figura 2: GPU Tesla V100 de NVIDIA [6].

2.3.3. Unidad de procesamiento tensorial (TPU) de Google

Las unidades de procesamiento tensorial son la perspectiva de Google en los aceleradores de inteligencia artificial. Existen versiones del mismo llamadas Edge TPU, que son circuitos integrados de aplicación específica, desarrollados por la empresa para brindar inferencia de alto rendimiento utilizando aprendizaje automático en dispositivos de bajo consumo de potencia. Un dispositivo de este tipo puede ejecutar 4 millones de millones de operaciones por segundo usando solamente 2 vatios de potencia. Es capaz de ejecutar redes neuronales profundas y redes neuronales convolucionales, haciéndolos ideales para aplicaciones de visión por computadora. Este dispositivo soporta modelos creados usando la plataforma TensorFlow y convertidos mediante TensorFlow Lite [7].

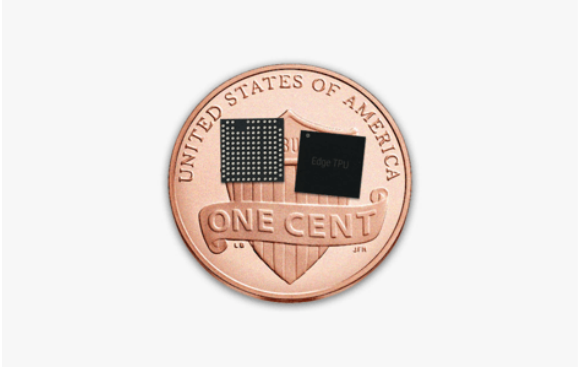


Figura 3: Unidad de procesamiento tensorial de Google [7].

2.4. Aprendizaje automático diminuto (*Tiny ML*)

Desde hace algún tiempo, el desarrollo tecnológico gira entorno a brindar cierta forma de inteligencia artificial a nuevos dispositivos a través de software. Se ha vuelto común tanto en la práctica como en el ámbito de la investigación el aprendizaje automático como una herramienta emergente que brinda solución a problemas complejos sin la necesidad de comprender el problema en su totalidad o complejidad. Si bien esta técnica es potente y aplicable en muchos campos, su implementación requiere de gran capacidad computacional en términos de memoria y procesamiento [8].

En consecuencia, el campo emergente del *Tiny Machine Learning*, que hace referencia a un aprendizaje automático minúsculo o diminuto, es investigado y abordado con la finalidad de brindar un universo nuevo de posibilidades de aplicación del aprendizaje automático al reducir de manera considerable la necesidad de una gran capacidad computacional. Dicha modificación permite su aplicación en dispositivos de recursos limitados como lo son los microcontroladores que a su vez son una parte importante del concepto de sistemas embebidos [8].

2.5. Plataforma TensorFlow

TensorFlow es una plataforma de código abierto para aprendizaje automático. Brinda un ecosistema exhaustivo de herramientas para desarrolladores, empresas e investigadores que desean impulsar el aprendizaje automático y construir aplicaciones escalables basadas en aprendizaje automático. Está diseñada de forma que se pueda aprender y construir modelos de forma fácil y rápida. Cuenta con un conjunto de APIs (interfaces de programación de aplicaciones) que hacen fácil la implementación de aprendizaje automático, aprendizaje profundo y otros. Esta plataforma incluye herramientas de pre-procesamiento de datos, ingesta de datos, evaluación de modelos y visualización [9]. Además, facilita el entrenamiento de modelos y su implementación. Su diseño portátil permite correr sus herramientas en una variedad amplia de dispositivos y plataformas, así como ser fácilmente escalable, de forma que se puede sacar ventaja de la capacidad de cada equipo [8].

Figura 4: Logo de plataforma TensorFlow [9].

2.6. Plataforma TensorFlow Lite

TensorFlow Lite es una plataforma, derivada de TensorFlow, que ofrece un conjunto de herramientas que permiten la implementación de modelos de aprendizaje automático desa-

rrollados con TensorFlow en dispositivos móviles o embebidos. Cuenta con convertidores e intérpretes, entre otros, de forma que se puedan convertir los modelos a lenguajes que puedan ser interpretados por otros dispositivos, como microcontroladores [9]. Esta herramienta realiza conversiones y optimizaciones del modelo para implementarlos en dispositivos de memoria limitada, por lo que realiza modificaciones de forma que se utilice el menor espacio de memoria posible y se mejore el tiempo para correr el modelo, buscando no afectar su precisión o bajar su rendimiento [8].

Figura 5: Logo de plataforma TensorFlow Lite [9].

2.7. Aprendizaje automático en visión por computadora

Las técnicas de aprendizaje automático siempre han jugado un rol importante en el desarrollo de algoritmos de visión por computadora. La principal herramienta del aprendizaje automático utilizada en visión por computadora fue el ajuste de parámetros numéricos de los algoritmos. Estos algoritmos pueden ser utilizados para clasificación, segmentación, mejoramiento de imágenes, estimación de movimiento y recuperación de profundidad. Actualmente, las redes neuronales profundas son los modelos más popular y ampliamente utilizados en visión por computadora. Se ha evolucionado de técnicas clásicas de visión por computadora donde las características y algoritmos de un modelo son hechos manualmente, a delegar a técnicas de aprendizaje profundo el aprendizaje de características y optimización, ajuste y mejoramiento de los algoritmos de un modelo. En esta aplicación destacan las redes neuronales convolucionales por su buen rendimiento en la detección de objetos y otras ramas de visión por computadora [10].

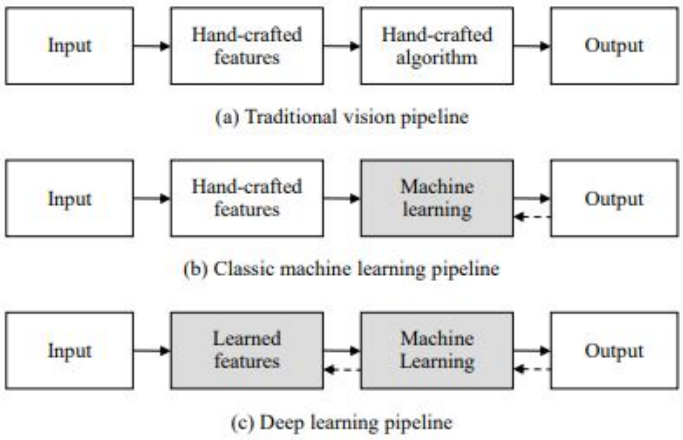


Figura 6: Diferentes enfoques de algoritmos de visión por computadora [10].

CAPÍTULO 3

Justificación

Actualmente, el aprendizaje automático es tendencia en el mundo de la tecnología y surgen nuevas ramas de ello, como lo es la aplicación del aprendizaje profundo en sistemas embebidos. Este escenario antes resultaba imposible por la complejidad de los programas y limitaciones de hardware, dificultades que se han solucionado con el paso del tiempo. El flujo de trabajo y esquema de *Tiny Machine Learning (Tiny ML)* resulta ser una herramienta esencial para mejorar las posibilidades y resultados de la aplicación del aprendizaje automático en sistemas embebidos, fundamentales en el campo de la Ingeniería Mecatrónica. Además, brinda una oportunidad de comenzar al día con los avances de este campo emergente y dar acceso a la academia guatemalteca, así como al Departamento de Ingeniería Electrónica, Mecatrónica y Biomédica, a esta tendencia y la posibilidad de abordar nuevas ramas de investigación relacionadas. Esta herramienta tiene aplicaciones dentro de la robótica y los sistemas de control, con lo que es vital para la investigación tecnológica que traerá nuevas herramientas para el futuro.

Tomando en cuenta lo mencionado anteriormente, este trabajo busca presentarse como una referencia base y guía para el entendimiento teórico, programación, entrenamiento e implementación de modelos de aprendizaje automático en sistemas embebidos. También se busca exponer a los lectores a la utilización de herramientas como Google Colaboratory y la librería TensorFlow, así como promover el aprovechamiento de sus recursos crecientes en este campo emergente con el fin de lograr tareas de clasificación y de visión de computadora a través de microcontroladores, que aporten nuevas soluciones a ecosistemas robóticos y otros campos.

CAPÍTULO 4

Objetivos

4.1. Objetivo general

Indagar y desarrollar aplicaciones típicas de aprendizaje automático a través del flujo de trabajo de *Tiny Machine Learning* en microcontroladores.

4.2. Objetivos específicos

- Investigar e implementar modelos de aprendizaje automático simples en un microcontrolador de recursos limitados usando la técnica emergente de *Tiny Machine Learning*.
- Utilizar aprendizaje profundo para desarrollar una aplicación compleja de visión por computadora a través de un módulo de visión de bajo costo y un microcontrolador de alto desempeño empleando redes neuronales convolucionales.
- Implementar un sistema de control basado en visión que emplee como retroalimentación el resultado del sistema de visión por computadora.

CAPÍTULO 5

Alcance

Este trabajo se centra exclusivamente en el diseño e implementación de aprendizaje automático a través de redes neuronales en microcontroladores y la utilización de sus resultados para crear aplicaciones más complejas. Se aborda la transformación de modelos de aprendizaje profundo desarrollados con la librería TensorFlow en Python, a modelos interpretables por microcontroladores. Esta problemática es emprendida desde el punto de vista del flujo de trabajo y las herramientas necesarias para lograrlo, entre las cuales se busca introducir la utilización de TensorFlow Lite y herramientas en línea como Google Colaboratory para mejor aprovechamiento de los recursos computacionales.

En el ámbito de creación de aplicaciones complejas para el aprovechamiento de la tendencia del *Tiny ML*, se aborda la temática de visión por computadora utilizando el algoritmo YOLO y la utilización de sus resultados para controlar un sistema robótico.

Debido a la situación socio-económica provocada por la pandemia COVID-19 a partir del año 2020, la asistencia a laboratorios para la utilización de equipos más completos para la realización del trabajo fue intermitente y se dio escasez de dispositivos electrónicos necesarios para el desarrollo del proyecto.

CAPÍTULO 6

Marco teórico

6.1. Inteligencia artificial

La inteligencia artificial es la ciencia e ingeniería de dotar a una máquina con cierto tipo de comportamiento que refleje inteligencia. Es cualquier técnica que permita a computadoras imitar el comportamiento humano en términos de razonamiento, análisis, toma de decisiones y otros factores, aunque no se limita a imitar lo que hacen los humanos. En muchas ocasiones, se busca plasmar en código mecanismos intelectuales que se necesitan para llevar a cabo una tarea [11].

Algunos de los campos en los que la inteligencia artificial se ve mayormente representada son:

- Inteligencia artificial lógica.
- Búsqueda.
- Reconocimiento de patrones.
- Representación.
- Inferencia.
- Razonamiento y sentido común.
- Planeamiento [11].

6.2. Aprendizaje automático

El aprendizaje automático es una rama de la inteligencia artificial que busca brindar a una computadora la capacidad de predecir eventos con base en observaciones del pasado.

Cuando se crea un programa de aprendizaje automático, se busca dar a una máquina la habilidad de aprender sin programar explícitamente lo que se busca aprender. Esta es una ciencia que ha desarrollado algoritmos que se optimizan y descubren relaciones y patrones por sí mismos a manera de hacer predicciones. Cualquier diseño de un modelo que incluya inteligencia artificial cuenta con dos procesos principales: entrenamiento e inferencia. En el entrenamiento se brinda información pasada a un modelo para que la utilice para mejorarse a sí mismo buscando realizar las predicciones esperadas. En el proceso de inferencia se utiliza el modelo creado para hacer predicciones basadas en datos reales que se utilizan como entrada al modelo [12].

6.3. Aprendizaje profundo

El aprendizaje profundo es uno de los enfoques más populares para abordar el aprendizaje automático. Este enfoque consiste en crear modelos que son redes que simulan las conexiones de las neuronas en el cerebro humano. El objeto fundamental que simula una neurona es llamado perceptrón. Estos objetos, al formar redes, permiten modelar relaciones entre múltiples entradas y salidas. Este es un enfoque flexible pues se puede orientar a resolver diferentes tares, además de ser una herramienta poderosa para resolver problemas que son adecuados para microcontroladores. Su función principal es extraer patrones de conjuntos de datos mediante la utilización de redes neuronales [13].

Este enfoque utiliza el siguiente flujo de trabajo:

1. Decidir un objetivo,
2. Recolectar un conjunto de datos,
3. Diseñar una arquitectura para el modelo,
4. Entrenar el modelo,
5. Hacer inferencias con el modelo, y
6. Evaluar y solucionar problemáticas del modelo [8].

6.4. *Tiny Machine Learning*

Tiny Machine Learning es una nueva tendencia que busca implementar el aprendizaje automático desde el enfoque de aprendizaje profundo en dispositivos de capacidades limitadas, tal como los microcontroladores. En esencia, se busca utilizar el flujo de trabajo del aprendizaje profundo para desarrollar modelos que sean implementables en sistemas embebidos a través de microcontroladores. Es importante tomar en cuenta que muchos dispositivos de la gama de los microcontroladores tienen limitantes tales como memorias que brindan relativamente poco espacio de almacenaje, capacidad de procesamiento limitada e implementan bajo consumo de potencia, por lo que es necesario modificar los algoritmos comunes de aprendizaje automático para poder utilizarlos de esta forma [8].

Originalmente, dado que los dispositivos computacionales no contaban con capacidades de procesamiento tan amplias y los algoritmos de aprendizaje automático era muy complejos y demandaban muchos recursos, era difícil su implementación. Con el crecimiento tecnológico y el paso del tiempo, se ha llegado a una época de auge del aprendizaje automático. Esta nueva tendencia brinda la oportunidad de dar acceso a soluciones inteligentes a dispositivos con capacidades limitadas a través de la utilización del flujo de trabajo del aprendizaje automático como base, aunque implementando ciertas modificaciones de forma que se tomen en cuenta las limitantes de los equipos. Este objetivo se ve facilitado ya que el crecimiento de las tecnologías en términos de procesamiento de información y los avances en la optimización de utilización de recursos en los modelos de aprendizaje automático acercan los campos de la inteligencia artificial y los sistemas embebidos de forma conveniente [8].

6.5. Perceptrón

Un perceptrón es la unidad básica de una red neuronal y del aprendizaje profundo. Busca simular una neurona del cerebro humano a nivel computacional con el fin de hacer inferencias al formar redes a partir de varias unidades de ellos [13]. Se puede representar matemáticamente de la siguiente forma:

$$y = g\left(\sum_{i=1}^{m} x_i w_i\right). \tag{1}$$

En la ecuación presentada, se toman los argumentos x_i y w_i, y se obtiene la salida y. Los argumentos representan a las variables de entrada al perceptrón y los pesos que se dan a cada variable de entrada, respectivamente. Dichos valores son normalmente números reales y se obtiene como resultado de la sumatoria una combinación lineal de las variables de entrada. Dicho resultado se utiliza como argumento en una función de activación no lineal $g(z)$, para luego obtener el valor de salida que puede ser utilizado para hacer inferencias. La unión de estos elementos conforma redes neuronales que pueden brindar resultados más interesantes y complejos [13].

6.6. Funciones de activación no lineales

Las funciones de activación no lineales son utilizadas en los perceptrones como la forma de añadir no linealidades al sistema que buscan imitar o del que se busca inferir. De no contar con funciones no lineales en los perceptrones, la gama de sistemas sobre los que se podría inferir usando perceptrones se vería reducida, puesto que la mayoría de sistemas en la realidad no son lineales [13], [10]. Entre las funciones de activación no lineales más comunes se encuentran:

- Función Sigmoide:

$$g(z) = \frac{1}{1 + e^{-z}}. \tag{2}$$

- Función Tangente Hiperbólica:

$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}. \tag{3}$$

- Función Unidad Lineal Rectificada (ReLU):

$$g(z) = \text{máx}(0, z). \tag{4}$$

6.7. Redes neuronales

Las redes neuronales son el producto de combinar conjuntos o unidades de perceptrones o neuronas. Esta idea surge de la necesidad de crear modelos que representen sistemas más complejos. Un único perceptrón brinda un cuello de botella muy limitante en el desarrollo de este campo y la solución a muchos problemas. Al combinar perceptrones, la interacción entre ellos puede brindar predicciones más precisas y comportamientos más complejos. Cuando hay una o más neuronas que trabajan a un mismo nivel, obteniendo las mismas entradas y brindando salidas a los mismos componentes, se dice que forman una capa. Una red neuronal puede estar formada de muchas capas con diferente número de perceptrones en cada una de ellas. Comúnmente una única capa recibe la información de entrada a la red y una única capa brinda la información de salida a la red, aunque es posible notar variantes en modelos complejos. Cada capa brinda su salida o predicción como entrada a una capa siguiente, o bien como resultado final. Además, las redes neuronales pueden ser configuradas a manera de recibir diferentes números de entradas y brindar diferente número de salidas según las necesidades que se tengan en la problemática que se busca solucionar [13], [10].

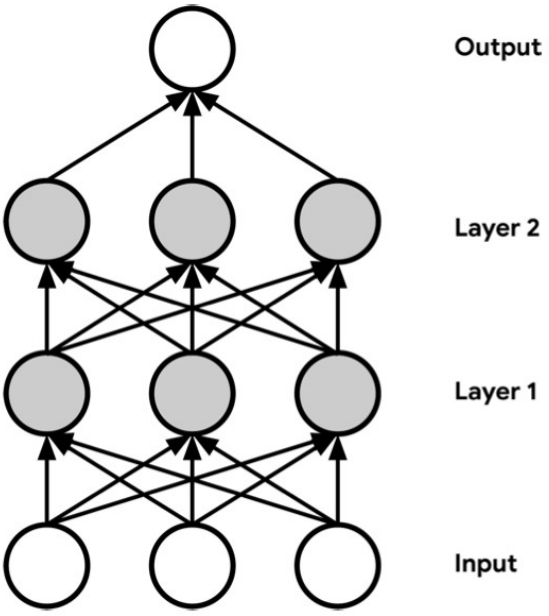


Figura 7: Ejemplo de red neuronal de aprendizaje profundo con dos capas [8].

6.8. Entrenamiento de un modelo (red neuronal)

El entrenamiento de un modelo es un proceso mediante el cual un modelo diseñado aprende a producir la inferencia correcta para un conjunto de datos de entrada dados. Este proceso involucra alimentar el modelo con datos de entrenamiento y modificar parámetros del modelo hasta obtener las inferencias más precisas posibles [8]. En este proceso es necesario contar con un conjunto de datos que cumpla lo siguiente:

- Contar con el conjunto de entradas al modelo, y
- Contar con la salida esperada del modelo para cada conjunto de entradas.

Utilizando dicho conjunto de datos, se realizan inferencias con el modelo. A partir de los resultados se realiza una comparativa de la salida del modelo con la salida esperada a partir de las entradas, de forma que se establece una medida del rendimiento del modelo. El resultado de la medida de rendimiento del sistema es utilizado como guía para la modificación de los parámetros del modelo. Los parámetros que se pueden modificar en un modelo son los pesos asignados a las variables de entrada en los perceptrones y los descentramientos [13], [10].

Se busca plantear el entrenamiento de los modelos de aprendizaje automático como un algoritmo iterativo que modifica los pesos automáticamente en función de la medida de rendimiento establecida para el modelo, de forma que con cada iteración se obtengan resultados más cercanos a los esperados del modelo en la salida. El momento de detener el entrenamiento normalmente se realiza cuando el modelo deja de mejorar su medida de rendimiento, puesto que luego de este punto pueden generarse problemas como el sobreajuste, tema que se tratará más adelante. Cuando luego de un entrenamiento adecuado de un modelo aún no se obtienen las inferencias esperadas, lo más adecuado es cambiar la arquitectura del modelo y comenzar el proceso de entrenamiento de nuevo. Cuando un modelo comienza a brindar predicciones precisas, se dice que el modelo convergió [13], [10].

6.9. Pérdidas en un modelo (red neuronal)

Las pérdidas en un modelo de aprendizaje automático son una medida de rendimiento del modelo. Se plantean como una medida de comparación entre las salidas obtenidas y las salidas esperadas al hacer inferencia con el modelo que se está desarrollando. La función de pérdida también puede ser interpretada como el costo incurrido por predicciones incorrectas, por lo que también es conocida como función costo. Se representa matemáticamente mediante la siguiente simbología [10].

$$\mathcal{L}\left(f(x_i; W), y_i\right) \tag{5}$$

De dicha notación se interpreta: $\mathcal{L}$ es una función que toma como argumentos $f(x_i, W)$ y y_i, donde $f(x_i, W)$ es la inferencia resultante de evaluar la iésima entrada x con el conjunto de pesos W en el modelo y y_i es la salida esperada al evaluar a iésima entrada en

el modelo. La función de pérdidas más comúnmente utilizada varía con la aplicación de la red neuronal en la que se utiliza. Entre las más conocidas para aplicaciones de regresión se puede hablar de la función conocida como error cuadrático medio, mientras que para aplicaciones de clasificación se utiliza entropía cruzada. Además, resulta natural computar las pérdidas en un modelo como la diferencia o resta entre la predicción y la salida esperada, aunque comúnmente se aplican modificaciones a este enfoque. Es importante mencionar que las funciones de perdidas en un modelos son consideradas funciones de los pesos W y los descentramientos, puesto que son los factores que influyen en la pérdida por parte de la red neuronal [13], [8], [10].

Algunas funciones de error comunes se expresan matemáticamente de la siguiente forma:

- Error cuadrático medio (MSE):

$$J(W) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f(x_i; W)\right)^2. \tag{6}$$

- Error absoluto medio (MAE):

$$J(W) = \frac{1}{n}\sum_{i=1}^{n}|y_i - f(x_i; W)|. \tag{7}$$

6.10. Optimización de pérdidas en un modelo (red neuronal)

La optimización de pérdidas en redes neuronales es un problema con el que se lidia al diseñar un modelo de aprendizaje profundo. Este consiste en encontrar los pesos W y los descentramientos correspondientes a cada perceptrón o neurona de la red neuronal para alcanzar la pérdida más pequeña posible. Minimizar la pérdida implicaría un comportamiento de predicción más cercano al esperado por parte de la red neuronal [13] [10].

Esta problemática es representada matemáticamente de la siguiente forma:

$$W^* = \arg\min_{W} J(W). \tag{8}$$

Donde se expresa que W es un vector que contiene los pesos de los perceptrones de la red neuronal y se busca encontrar sus valores tal que se minimice el valor de la función de pérdidas.

Al buscar solución a esta problemática, es común considerar la función de pérdidas como una superficie que muestra valles y picos en función de los valores de los pesos de los perceptrones de la red neuronal. Al realizar optimización de pérdidas, se utilizan algoritmos que utilizan el gradiente de la superficie mencionada para encontrar los cambios que hay que aplicar al vector de pesos de forma que las pérdidas sean mínimas [13], [10].

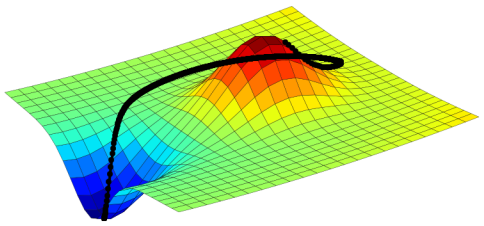


Figura 8: Superficie de pérdidas en función de los pesos de la red neuronal [8].

Con dicho algoritmo, se busca que el vector de pesos siga una trayectoria que lleve al valor de pérdidas a estar en un valle de la superficie. Los algoritmos de este tipo son llamados de descenso de gradiente y siguen los siguientes pasos [13], [10]:

1. Inicializar los pesos en la red neuronal de forma aleatoria.
2. Repetir hasta encontrar convergencia:
   - *a*) Computar el gradiente de la función de pérdidas.
   - *b*) Calcular nuevo vector de pesos para la red neuronal.
   - *c*) Actualizar el vector de pesos para la nueva iteración del algoritmo.
3. Retornar el vector de pesos óptimo para la red neuronal.

Al momento de calcular los nuevos pesos para la red neuronal se consideran los pesos de la iteración anterior y se resta una fracción del gradiente de la función de pérdida. Esto se hace con el fin de obtener siempre un efecto productivo de términos de minimizar las pérdidas, al siempre buscar estar más cerca de los valles de las superficies de pérdida. Se considera la fórmula a continuación [13], [10]:

$$W = W - \eta \frac{\partial J(W)}{\partial W}. \tag{9}$$

El valor η, llamado tasa de aprendizaje, debe ser fijado acorde al comportamiento que se desea en la optimización. Una tasa de aprendizaje mayor puede provocar un entrenamiento más rápido como efecto positivo y divergencia como efecto negativo. Una tasa de aprendizaje menor puede provocar un entrenamiento convergente como efecto positivo y un entrenamiento más lento como efecto negativo. En nuevos desarrollos se consideran tasas de aprendizaje adaptativas que varían en función de la velocidad de aprendizaje de la red neuronal, del valor del gradiente, del tamaño de algunos pesos de la red u otros, de forma que ya no es fija y puede ser controlada de mejor manera [13], [10].

Existen diferentes algoritmos de descenso de gradiente, entre los cuales se encuentran:

- Descenso de gradiente estocástico (SGD).
- Adam.
- Adadelta.
- Adagrad.
- RMSProp.

6.11. Retropropagación

La retropropagación es una forma de computar los gradientes de una función de pérdida. Esto es necesario pues se busca darle lógica y sencillez al cálculo de gradiente necesario en la optimización de las pérdida. Este método retorna un valor numérico a la pregunta: ¿Cómo afecta a la función de pérdidas $J(W)$ un cambio pequeño en un peso? La retropropagación rastrea el cambio de las pérdidas a partir del cambio de los valores de predicción que brindan los perceptrones en cada capa de la red neuronal, utilizando el concepto de la regla de la cadena del cálculo diferencial. Al utilizar este concepto, se puede encontrar el efecto del cambio de cada peso de la red en las pérdidas, de forma que se pueda variar para obtener mejores resultados. La expresión matemática que expone este modelo para una red neuronal de dos capas, una entrada y una salida es la siguiente [10]:

$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial y} \cdot \frac{\partial y}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}. \tag{10}$$

Donde y representa la predicción del modelo y w_1 representa un peso de la red neuronal, z_1 representa la predicción dada por el perceptrón de una primera capa. Esta expresión podría ser ampliada según las características del modelo que se busque analizar.

6.12. El problema del ajuste

El entrenamiento puede ser un proceso complicado si no se conoce el momento de pararlo. Este es un proceso iterativo y requiere de experiencia conocer el momento indicado para dejar de entrenar un modelo para lograr convergencia en el entrenamiento. Parar el entrenamiento de un modelo puede llevar a tres posibles resultados relacionados al ajuste [13]:

- Desajuste: el modelo no ha aprendido suficiente acerca de los patrones de los datos de entrenamiento para hacer buenas predicciones al hacer inferencia. Este caso comúnmente ocurre cuando se detiene el entrenamiento antes del momento adecuado.
- Ajuste ideal: el modelo ha aprendido de forma adecuado a extraer los patrones necesarios para hacer una buena estimación con nueva información. Este caso ocurre cuando se detiene el entrenamiento de una red neuronal en el momento adecuado.

- Sobre-ajuste: el modelo ha aprendido de forma muy exacta los patrones y resultados esperados por parte de los datos de entrenamiento, de forma que es muy bueno al realizar inferencias con datos conocidos, aunque brinda malos resultados al hacer inferencias con datos nuevos. Este caso ocurre comúnmente cuando se detiene el entrenamiento de la red neuronal demasiado tarde.

6.13. Regularización

La regularización es una técnica que busca solucionar los problemas de optimización y aprendizaje de un modelo de aprendizaje profundo al limitarlos mediante variaciones en el modelo. Estas técnica es necesaria ya que ayuda a mejorar las inferencias de nuestro modelo al utilizar datos nunca antes ingresados al modelo [10], [13]. A continuación se muestran dos formas de aplicar la técnica de regularización:

1. Técnica de marginado: consiste en definir aleatoriamente algunas activaciones (salidas de perceptrones) en la red como cero.
    - Comúnmente se colocan a cero el cincuenta por ciento de las activaciones en una capa.
    - Provoca que la red no dependa únicamente de algún nodo.
2. Parada temprana o anticipada: consiste en detener el entrenamiento de una red neuronal antes de tener una posibilidad de sobre-ajuste del modelo.
    - Se busca parar al momento que la pérdida llega a su mínimo.
    - Si no se detiene en el momento correcto las pérdidas seguirán aumentando aún con más entrenamiento.

En otros casos, la regularización hace referencia a añadir restricciones al problema de optimización que representa el entrenamiento de una red neuronal. Esto se logra añadiendo términos a la función de pérdida utilizada durante el entrenamiento.

6.14. Aprendizaje poco profundo vs. Aprendizaje profundo

Los enfoques del aprendizaje automático conocidos como aprendizaje poco profundo y aprendizaje profundo son en esencial muy similares. Su diferencia más grande radica en la extracción de características que ingresan a un modelo. En el caso del aprendizaje poco profundo la extracción de características de los datos en los que se basará un modelo es fundamentalmente manual. Esto significa que en el modelo de aprendizaje automático se realizará aprendizaje a partir de datos descritos por características predefinidas, razón por la cual se requiere expertos en la temática de los datos a trabajar al momento de usar este enfoque [14].

En un caso contrario, al utilizar el enfoque de aprendizaje profundo, las extracción de características de un conjunto de datos es computada de forma algorítmica sin intervención

humana. Esto significa que el sistema desarrollado seleccionará automáticamente las características importantes y les asignará un peso de forma que se puede hacer inferencia y tomar en cuenta las relevantes. Entre otras implicaciones de este enfoque se encuentra que se requieren más datos, pues se busca reconocer patrones no descritos durante el entrenamiento del modelo [13].

6.15. Visión por computadora usando aprendizaje automático

6.15.1. Redes neuronales convolucionales

Las redes neuronales convolucionales son redes complejas de perceptrones convolucionales combinadas con capas de filtrado específicamente dedicadas al procesamiento de imágenes en el campo del aprendizaje automático. Las imágenes son interpretadas por las computadoras como como matrices numéricas que representan colores. Debido a esto, es necesario encontrar la forma de extraer características de una imagen para utilizarlas como entrada de una red neuronal que puede ejecutar tareas como localización, detección o clasificación de objetos a partir de la imagen [10], [13].

¿Qué es la operación de convolución?

La convolución es una operación realizada entre las matrices representativas de imágenes y kernels, la cual brinda como resultado una nueva imagen. La imagen resultante cuenta con cambios dependiendo del kernel que se utilice para la operación, puesto que hay kernels para realizar diferentes tareas que pueden ser afilado, detección de bordes, suavizado u otros. Esta operación es realizada en visión por computadora como una etapa previa a la reducción de información, extracción de información e inferencia [13].

La operación se realiza mediante una multiplicación elemento por elemento entre el kernel y una región seleccionada de la imagen. Luego, una suma de los elementos resultantes es colocada en la posición dictada por el centro del kernel, aunque en una nueva imagen. El kernel cambia de posiciones de forma que abarca toda la imagen y se tiene un resultado de imagen con el efecto o filtro aplicado [13].

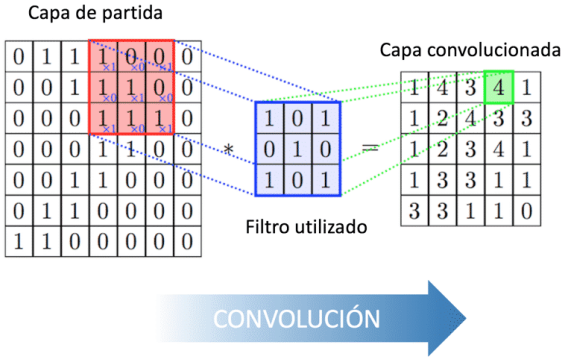


Figura 9: Ejemplo de operación de convolución, basado en [10].

Arquitectura y operación de una red neuronal convolucional

La arquitectura más comúnmente utilizada para una red neuronal convolucional en dos etapas que pueden contar con varias capas internas. Las capas que se utilizan y su funcionalidad se describen a continuación [10], [13]:

1. Etapa de aprendizaje de características
    a) Aplicación de la operación de convolución.
    b) Aplicación de no linealidades.
    c) Reducción de información mediante técnicas como agrupación.
2. Etapa de ejecución de tarea
    a) Ejecuta tareas como clasificación, detección de objetos, segmentación y control probabilístico mediante redes neuronales con la información reducida resultante de la etapa de aprendizaje de características.

En la etapa de aprendizaje de características, se busca hacer una extracción y énfasis efectivo en características relevantes de los objetos en las imágenes de forma que puedan ser utilizadas para su clasificación. A su vez, se busca reducir la información, puesto que las imágenes en su formato original pueden ser muy complejas para una red neuronal, de forma que solo se utiliza la información importante de las imágenes como entrada para hacer inferencias [10], [13].

En la etapa de ejecución de tareas se desarrolla una red convolucional que desarrolle la tarea deseada. En el campo de la visión por computadora es común que se ejecuten tareas de clasificación y detección de objetos. Es posible diseñar para esta etapa redes neuronales que ejecuten más de una tarea a la vez, de forma que se pueden obtener resultados más elaborados [10], [13].

6.15.2. YOLO - *You Only Look Once*

Este es un algoritmo de visión por computadora que implementa aprendizaje automático. Es un algoritmo unificado que cumple con la función de detección de objetos en tiempo real. Fue presentado en 2016 como un nuevo enfoque en el campo de detección de objetos y en su momento presentó capacidad de analizar hasta 45 cuadros por segundo. Se distingue de algoritmos clásicos de visión por computadora al utilizar una única red neuronal para detectar objetos, colocarles cuadros delimitadores y clasificarlos a través de probabilidades obtenidas de la inferencia mediante la red neuronal [15].

Existen otros sistemas que cumplen la misma función que el algoritmo YOLO, aunque estos reutilizan modelos de clasificación para realizar detección de objetos por lo que no son los más efectivos. El algoritmo YOLO utiliza un modelo que predice múltiples cuadros delimitadores de forma simultánea y brinda probabilidades de clasificación para lo contenido en cada cuadro, según [15]. El proceso es realizado en tres pasos principales que consisten en:

1. Redimensionar la imagen de entrada,

2. Ejecutar una única red neuronal convolucional en la imagen, y

3. Clasificación mediante un umbral de las probabilidades de la detección resultante [15].

Otro aspecto interesante al respecto es que el algoritmo YOLO toma en cuenta imágenes completas durante el entrenamiento y durante la inferencia, por lo que aprende a codificar el contexto de los objetos para optimizar su clasificación. Por otro lado, también presenta limitantes, pues con el fin de hacer el algoritmo más rápido se ve limitado el número de cuadros delimitadores que se pueden definir en una imagen según una cuadrícula que se utiliza para dividir los espacios. Además, cada cuadro delimitador puede únicamente tomar una clasificación. Esto podría generar problemas al trabajar en la detección y clasificación de objetos pequeños agrupados [15].

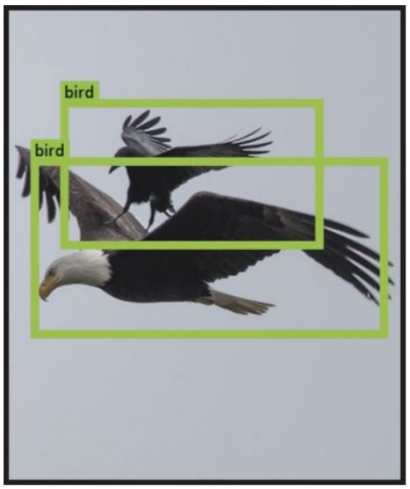


Figura 10: Ejemplo de resultado de detección y clasificación mediante algoritmo YOLO [15].

6.16. Control basado en visión

En temáticas menos complejas, se emplea cinemática de cuerpos rígidos no solo para describir el movimiento de manipuladores seriales y robots móviles (con ruedas) simples, sino también para describir la transformación proyectiva que realizan las cámaras al tomar imágenes del mundo tridimensional. En este punto, ambas tareas se ejecutan de forma independiente. El control basado en visión o *visual servoing* intenta unir ambas tareas al utilizar una cámara como sensor en el lazo de control cinemático del robot en cuestión. Se cuenta con un algoritmo general de cinemática inversa, que se expresa en la ecuación 11, la cual corresponde a un controlador que garantiza que el efector final alcance la pose fija deseada $\boldsymbol{\xi}_d$ a partir de cualquier configuración inicial $\boldsymbol{q}_0$.

$$\boldsymbol{q}_{k+1} = \boldsymbol{q}_k + \boldsymbol{J}^{\dagger}(\boldsymbol{q}_k)\boldsymbol{K}\boldsymbol{e}(\boldsymbol{q}_k) \tag{11}$$

Este algoritmo cuenta con supuestos que pueden llegar a ser problemáticos en un entorno de trabajo realista. Los supuestos son específicamente relacionados con el cálculo del error de pose $\boldsymbol{e}(\boldsymbol{q}_k) =^B \boldsymbol{\xi}_E(\boldsymbol{q}_k) - \boldsymbol{\xi}_d$ se listan a continuación.

1. Se cuenta con un modelo cinemático perfecto del robot, por lo que se puede calcular ${}^B\boldsymbol{\xi}_E(\boldsymbol{q}_k)$ conociendo únicamente la configuración actual del robot.

2. Se conoce a priori la pose deseada del efector final y que dicha pose no cambia durante la ejecución de la rutina de control.

Estos supuestos se tornan problemáticos, ya que asumen situaciones que difieren de la realidad de estos sistemas. En el caso del primer supuesto, si bien se cuenta con tecnologías sofisticadas que son modeladas de forma precisa, dichos modelos no contemplan desgastes de sistemas mecánicos u otros factores que podrían hacer que el sistema real difiera del modelo. En cuanto al segundo supuesto, dicho supuesto limita la aplicación del algoritmo a tareas repetitivas y no contempla cambios de orientación del objetivo, lo que podría causar una no convergencia cuando esto suceda.

Tomando en cuenta estos criterios, el control basado en visión busca corregir posibles errores que surgen de la utilización del modelo cinemático del robot únicamente y de su interacción con el mundo real. Existen dos metodologías que difieren en el tipo de información extraída por la cámara. Una es llamada *visual servoing* basado en posiciones y utiliza la cámara como un sensor para la estimación de pose de un objeto a manipular o rastrear, con el fin de colocar al efector final del robot en una pose deseada respecto del objeto. Por otro lado, existe una metodología llamada *visual servoing* basado en imágenes que utiliza la cámara como sensor de visión en un modelo cinemático que combina el manipulador y la cámara empleando características extraídas por la cámara en el plano de imagen. En el segundo caso no es necesaria ninguna estimación de pose para cuerpos rígidos, sino que se consideran cambios de coordenadas de diferentes puntos de interés en el plano de imagen de la cámara.

Ambas metodologías contemplan un manipulador serial en una configuración ojo en mano o *eye-in-hand*, donde la cámara se encuentra instalada en el efector final del robot.

6.16.1. *Visual Servoing* basado en posiciones

Esta metodología busca hacer que la pose del efector final respecto de la base ${}^B\boldsymbol{T}_E(\boldsymbol{q})$ llegue a una pose deseada $\boldsymbol{T}_d = {}^B\boldsymbol{T}_D$ mediante la utilización de poses relativas del objeto respecto de la cámara ${}^C\boldsymbol{T}_O$ y del objeto respecto del marco de referencia {D} de la pose deseada ${}^D\boldsymbol{T}_O$. La primera pose mencionada puede ser encontrada al resolver el problema de estimación de pose, mientras que la segunda pose mencionada brinda robustez a la tarea ya que cambia la pose deseada al cambiar la pose del objeto al estar especificadas de forma relativa. Uno de los supuestos más relevantes en esta metodología es que la pose de la cámara y del efector final son idénticas, lo que también debería ser forzado en el diseño del robot con la configuración de ojo en mano [16],[17].

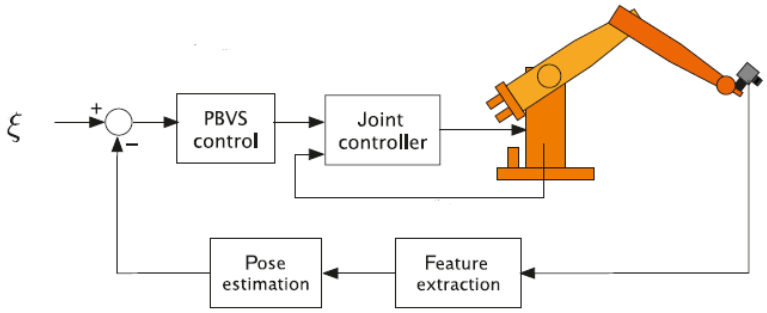


Figura 11: Lazo de control mediante *Visual servoing* basado en posiciones [17].

Al momento de hacer control, se define el error de la forma siguiente, con valores obtenidos de la matriz de transformación homogénea.

$$\boldsymbol{e}({}^{D}\boldsymbol{\xi}_{C}) = -\begin{bmatrix} {}^{D}\boldsymbol{o}_{C} \\ {}^{D}\boldsymbol{\epsilon}_{C} \end{bmatrix} \tag{12}$$

Luego, mediante distintas operaciones se hace efectiva la restricción de que las poses del efector final y la cámara son iguales, con lo que se logra que la pose de la cámara respecto de la base se iguale a la cinemática directa del manipulador, también igualando su velocidad a la de la cinemática diferencial. Producto de esto, se obtiene la dinámica del problema a solucionar, similar al algoritmo de cinemática inversa [16],[17].

$$\dot{\boldsymbol{e}} = -\boldsymbol{A}({}^{D}\boldsymbol{\xi}_{C}){}^{D}\boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}} \tag{13}$$

Se utiliza $\dot{\boldsymbol{q}}$ como variable de control ya que todas las juntas del manipulador se encuentran controladas mediante comandos de velocidad. La matriz $\boldsymbol{A}$ tiende a ser la matriz identidad cuando el sistema se estabiliza por lo que se omite en el controlador al utilizar ese caso, obteniendo la siguiente dinámica [16],[17].

$$\dot{\boldsymbol{e}} = -\boldsymbol{K}\boldsymbol{e} \tag{14}$$

En la ecuación anterior, se garantiza un sistema globalmente asintóticamente estable (G.A.S) siempre que $\boldsymbol{K} \succ 0$, que luego puede ser presentado como un algoritmo iterativo [16],[17].

$$\boldsymbol{q}_{k+1} = \boldsymbol{q}_{k} + {}^{D}\boldsymbol{J}^{\dagger}(\boldsymbol{q}_{k}, {}^{D}\boldsymbol{\xi}_{C}[k])\boldsymbol{K}\boldsymbol{e}_{k} \tag{15}$$

Este algoritmo de control requiere una estimación de pose del objeto a manipular con respecto al marco de la cámara, derivado de la pérdida de información que genera la utilización de una cámara como sensor. Requiere conocimiento de parámetros intrínsecos de la cámara y nociones de la geometría del objeto al considerar puntos conocidos. Esta falta de información representa a un problema muy similar al que se resuelve para calibrar cámaras.

La solución requiere de muchas operaciones y es computacionalmente cara por lo que normalmente se busca objetos con características sencillas y explotar otras características que ayuden a encontrar el parámetro necesario [16],[17].

6.16.2. *Visual Servoing* basado en imágenes

Esta metodología utiliza características de la imagen que fueron medidas por la cámara en su proceso de percepción, ignorando la naturaleza tridimensional del objeto a rastrear. En este problema surge la necesidad de encontrar un sistema que integre la dinámica del sistema robótico con las características de imagen ($\boldsymbol{s}$). En este caso se consideran las características $\bar{u}$ y $\bar{v}$ que componen $\boldsymbol{s}$ y son las coordenadas de un punto en el plano de imagen expresado en píxeles [16],[17].

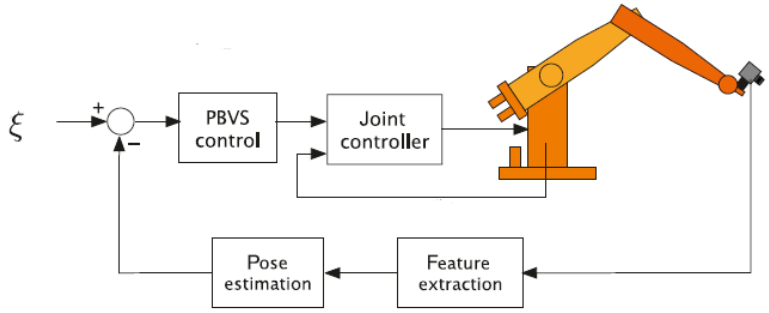


Figura 12: Lazo de control mediante *Visual servoing* basado en imágenes [17].

A partir de este concepto, es necesario encontrar una relación entre la velocidad de las características de imagen con las velocidades de la cámara o efector final que coincide. El objeto que relaciona ambas cantidades es conocido como Matriz de Interacción $\boldsymbol{L}_p$ y utiliza parámetros intrínsecos de la cámara como su longitud focal f, la cantidad de píxeles en el sensor ρ y la profundidad del punto a manipular c_z [16],[17].

$$\boldsymbol{L}_p(\boldsymbol{s}, c_z) = \begin{bmatrix} -\frac{f'}{c_z} & 0 & \frac{\bar{u}}{c_z} & \frac{\bar{u}\bar{v}}{f'} & -\frac{f'^2+\bar{u}^2}{f'} & \bar{v} \\ 0 & -\frac{f'}{c_z} & \frac{\bar{v}}{c_z} & \frac{f'^2+\bar{v}^2}{f'} & -\frac{\bar{u}\bar{v}}{f'} & -\bar{u} \end{bmatrix} \tag{16}$$

$$\dot{\boldsymbol{s}} = \boldsymbol{L}_p(\boldsymbol{s}, c_z) \begin{bmatrix} {}^E\boldsymbol{v}_E \\ {}^E\boldsymbol{\omega}_{BE} \end{bmatrix} \tag{17}$$

Al hacer coincidir el marco de referencia de la cámara con el marco de referencia del efector final se puede considerar la dinámica del manipulador al relacionar las características del plano de imagen con la configuración del robot [16],[17].

$$\dot{\boldsymbol{s}} = \boldsymbol{J}_L(\boldsymbol{q}, \boldsymbol{s}, c_z)\dot{\boldsymbol{q}} \tag{18}$$

La matriz $\boldsymbol{J}_L$ es el Jacobiano de imagen y se cumple aunque haya un offset entre la cámara y el efector final. Luego, al momento de fijar una referencia y hacer control para las características de imagen presenten el comportamiento deseado el problema es planteado de forma más sencilla [16],[17].

$$\boldsymbol{e_s} = \boldsymbol{s}_d - \boldsymbol{s} \tag{19}$$

$$\dot{\boldsymbol{e}}_{\boldsymbol{s}} = -\dot{\boldsymbol{s}} = -\boldsymbol{J}_L(\boldsymbol{q}, \boldsymbol{s}, c_z)\dot{\boldsymbol{q}} \tag{20}$$

$$\dot{\boldsymbol{q}} = \boldsymbol{J}_L^{\dagger}(\boldsymbol{q}, \boldsymbol{s}, c_z)\boldsymbol{K}\boldsymbol{e_s} \tag{21}$$

Este controlador, al considerar qe se puede controlar la velocidad $\dot{\boldsymbol{q}}$ de los actuadores, hace que el error presente una dinámica lineal invariante en el tiempo (L.T.I.) [16],[17].

$$\dot{\boldsymbol{e}}_{\boldsymbol{s}} = -\boldsymbol{K}\boldsymbol{e_s} \tag{22}$$

Además, esta dinámica es garantizada globalmente asintóticamente estable (G.A.S) siempre que $\boldsymbol{K} \succ 0$, que luego puede ser presentado como un algoritmo iterativo. Además, es importante que al utilizar una aproximación de inversa por el método de pseudo inversa y se asume que la matriz de constantes $\boldsymbol{K}$ absorbe el período de muestreo [16],[17].

$$\boldsymbol{q}_{k+1} = \boldsymbol{q}_k + \boldsymbol{J}_L^{\dagger}(\boldsymbol{q}_k, \boldsymbol{s}[k], c_z[k])\boldsymbol{K}\boldsymbol{e_s}[k] \tag{23}$$

Este algoritmo de control requiere una estimación para el parámetros de profundidad de los puntos de interés dado que la cámara provoca dicha perdida de información. Existen dos opciones para realizar dicha estimación [16],[17].

1. Asumir valor constante: se puede estimar un valor constante para la profundidad del punto de interés ya que el sistema es bastante robusto y esto permite que el sistema disminuya su sensibilidad en lazo cerrado a cambios de dicho parámetro.

2. Estimar mediante mínimos cuadrados: se puede calcular mediante este método aunque podría ser complicado para ejecución en microcontroladores ya que requiere de tareas con mayor cantidad de operaciones.

CAPÍTULO 7

Materiales, métodos y equipo

El trabajo consistió en el desarrollo de tres aplicaciones:

1. Aplicaciones para modelos de aprendizaje automáticos simples en microcontroladores: Regresión.
2. Aplicaciones para modelos de aprendizaje automáticos simples en microcontroladores: Reconocimiento de patrones.
3. Visión por computadora utilizando aprendizaje automático a través de módulo de visión y microcontrolador: Arduino, JeVois A33 y YOLO.

La realización de dichas aplicaciones y la experimentación con ellas tiene ciertas necesidades básicas para su desarrollo e implementación que a continuación se listan y explican.

7.1. Necesidades de material

- Cable para protoboard: se utilizó para realizar cableado entre componentes que interactúan en las aplicaciones que se desarrollaron.
- Jumpers: se utilizaron para realizar conexiones a motores y Arduino, que no necesariamente se integran al protoboard.
- Plancha de MDF de 3.2 mm de espesor: se utilizó para la elaboración de la estructura del brazo para giro e inclinación del módulo de visión.
- Cola blanca: se utilizó para unión y aseguramiento de las piezas de la estructura del brazo para giro e inclinación del módulo de visión.

- Tornillos: se utilizaron para el ensamblaje de los motores y el módulo de visión a la estructura del brazo para giro e inclinación del módulo de visión.
- Placa perforada de cobre: utilizada para realizar soldadura de cableado de motores y microcontrolador.
- Estaño: se utilizó para realizar soldaduras de cables de comunicación serial, energía de motores y otros.

7.2. Necesidades de equipo

- Computadora: necesaria para el uso de software para programación del módulo de visión, programación de Arduino, desarrollo de modelos de aprendizaje automático, modelado y otros. Se utilizó una computadora Dell Inspiron 15 5000, con un CPU Intel Core i7, sistema operativo Windows 10, Memoria RAM de 12 GB.
- Fuente de voltaje DC: necesaria para alimentar los dispositivos a utilizar en las aplicaciones que se desarrollaron.
- Cortadora láser: utilizada para realizar el corte de la plancha de MDF para obtener las piezas de la estructura del brazo para giro e inclinación del módulo de visión.
- Estación de soldadura: utilizada para soldar extensiones de cable a los cables de comunicación serial en el módulo de visión.

7.3. Necesidades de software

- Google Colaboratory: utilizado para la ejecución de programas de Python en hardware remoto de Google. Fue instalado desde el entorno de Google Drive.
- Python 3: fue utilizado para desarrollar de los modelos de aprendizaje automático en el proyecto. Dentro de él, se utilizaron diferentes librerías:
    - Glob: librería utilizada para encontrar todos los archivos en una ruta que coinciden con un formato específico.
    - Google.colab: empleada para obtener versiones específicas de librerías de python para ejecutar en Google Colaboratory.
    - Math: utilizada para obtener valores de constantes conocidas y realizar operaciones matemáticas.
    - Matplotlib: utilizada para realizar gráficas de resultados que ayudaron a entender los modelos que se desarrollaron.
    - NumPy: utilizada para realizar operaciones matemáticas en Python.
    - OpenCV: librería de código abierto que incluye algoritmos de visión por computadora empleada para realizar inferencia utilizando redes neuronales convolucionales en imágenes.
    - Os: empleada para creación y verificación de existencia de rutas y directorios.

    - Pandas: utilizada para realizar la manipulación de datos de forma más sencilla.
    - Random: utilizada para la generación de números pseudo - aleatorios.
    - Re: librería utilizada para verificar la existencia de secuencias o patrones en textos.
    - Sklearn.metrics: empleada para calcular matrices de confusión y graficarlas de forma adecuada.
    - TensorFlow: utilizada para desarrollar modelos y entrenarlos de forma más fácil y rápida. Además, también se utiliza TensorFlow Lite para convertir el modelo a un formato para microcontroladores. En TensorFlow se incluye Keras, utilizado por su representación simple de elementos necesarios en el desarrollo de redes neuronales.
    - xxd: empleado para desplegar contenido de un archivo en formato binario o hexadecimal.
- Repositorio de Darknet: repositorio clonado para la implementación del algoritmo YOLO utilizando el framework de Darknet. El repositorio de GitHub de Darknet puede ser encontrado en este enlace.
- Arduino IDE: entorno de desarrollo integrado utilizado para desarrollar código para Arduino, compilar y cargarlo al dispositivo.
- RealTerm Serial: utilizado para visualizar resultados mediante un monitor serial y guardarlos para el análisis de los datos.
- Autodesk Inventor: software de modelado 3D utilizado para diseñar la estructura del brazo para giro e inclinación del módulo de visión.
- Open Broadcaster Software (OBS): aplicación de video utilizada para la captura de video a través de puerto serial.
- JeVois Inventor: entorno de desarrollo integrado utilizado para desarrollar código para el dispositivo JeVois A33 y controlar sus configuraciones.

7.4. Necesidades de hardware

Dado que la ejecución de modelos de aprendizaje automáticos fue dependiente de la librería TensorFlow en su apartado para microcontroladores, se seleccionaron dispositivos a partir de la lista de microcontroladores para los que se encuentra disponible la plataforma con base en su disponibilidad, popularidad, documentación y comunidad. La lista limitada de plataformas que se pueden utilizar con estos fines son:

1. Arduino Nano 33 BLE Sense.
2. SparkFun Edge.
3. STM32F746 Discovery Kit.
4. Adafruit EdgeBadge.

5. Adafruit TensorFlow Lite for Microcontrollers Kit.
6. Adafruit Circuit Playground Bluefruit.
7. Espressif ESP32-DevKitC.
8. Espressif ESP-EYE.
9. Wio Terminal: ATSAMD51.
10. Himax WE-I Plus EVB Endpoint AI Developement Board.
11. Synopsys DesignWare ARC EM Software Developement Platform.
12. Sony Spresense.

Tomando en cuenta lo expresado anteriormente, el hardware utilizado para el desarrollo del trabajo de graduación fue el siguiente.

- Arduino Nano 33 BLE Sense: plataforma utilizada para ejecutar los modelos de aprendizaje automático, seleccionada por su comunidad, documentación, utilización en ejemplos de libros guía y popularidad. El dispositivo tuvo que ser comprado en Estados Unidos e importado ya que no existían unidades disponibles en Guatemala.

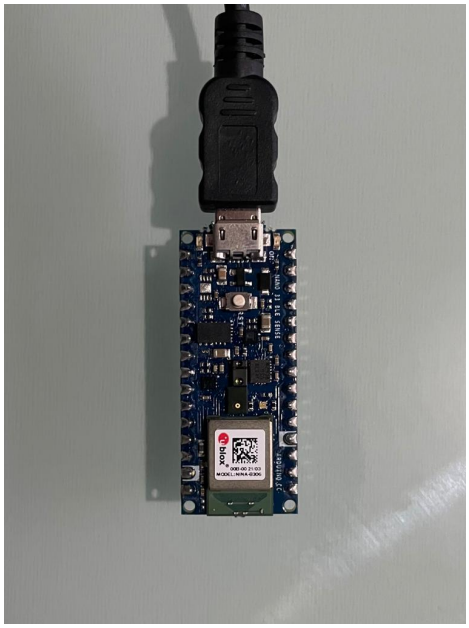


Figura 13: Arduino Nano 33 BLE Sense utilizado en el trabajo.

- Espressif ESP32-DevKitC: plataforma seleccionada para ejecución de modelos de aprendizaje automátivo. Utilizada por su disponibilidad en Guatemala y documentación disponible.

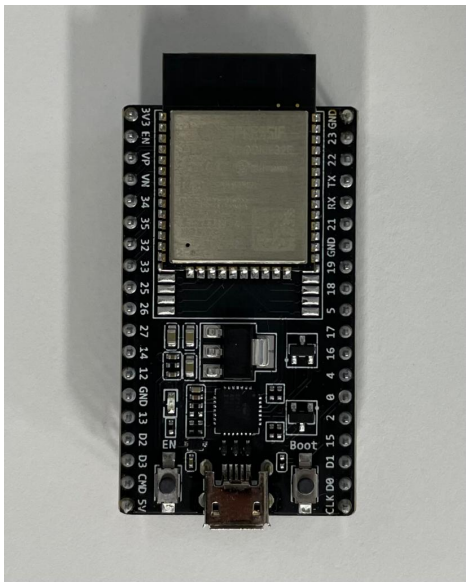


Figura 14: Espressif ESP32-DevKitC utilizado en el trabajo.

- Módulo de visión JeVois A33: módulo de visión utilizado para ejecutar algoritmo YOLO y enviar resultados mediante comunicación serial.
- Servomotores Hitec HS-8360TH: servomotores utilizados para realizar las acciones de inclinación y giro en el brazo para el módulo de visión. Se utilizaron dos unidades y fueron seleccionadas por su alta velocidad de reacción.
- Potenciómetro de 10K: se utilizó para poder hacer un ADC para leer una entrada analógica para el Arduino Nano 33 BLE Sense.

CAPÍTULO 8

Aplicaciones para modelos de aprendizaje automático simples en microcontroladores: Regresión

8.1. Diseño experimental

Este experimento consistió en realizar una aplicación de regresión en Arduino a través de la ejecución de modelos de aprendizaje automático simple. La señal de entrada para los modelos es obtenida mediante la lectura de un ADC del microcontrolador y el valor de salida es representado mediante la intensidad del LED integrado en el microcontrolador y numéricamente en el monitor serial del Arduino IDE. La regresión se realizó para la función $y = \sin x$ en el rango de 0 a 2π, donde la entrada del modelo representó a la variable x y la salida a la variable y. Se realizó el experimento con el fin de ilustrar el flujo de trabajo para ejecutar un modelo de aprendizaje automático en un microcontrolador, así como la interacción con la ejecución del modelo en el código. Esta aplicación se realizó tomando como base el ejemplo "Hello World"de [8]. Los modelos resultantes se implementaron en los microcontroladores Arduino Nano 33 BLE Sense y Espressif ESP32 DevKitC.

El flujo de trabajo empleado se describe a continuación:

1. Desarrollo del modelo de aprendizaje automático.
    - *a*) Generación de datos (auto generados).
    - *b*) División de los datos.
    - *c*) Diseño del modelo.
    - *d*) Entrenamiento del modelo.
        1) Parámetros de entrenamiento.
        2) Resultados del entrenamiento.

e) Inferencia en datos de prueba para evaluación del modelo.

2. Generación de un modelo de TensorFlow Lite.

   a) Conversión del modelo.

   b) Comparación de los modelos.

   c) Modelos resultantes como archivo fuente en C.

3. Implementación del modelo mediante Arduino IDE.

   a) Creación de un intérprete.

   b) Obtención y manejo de entradas y salidas al modelo.

4. Pruebas de la aplicación.

8.2. Características del sistema utilizado

Las etapas de desarrollo de los modelos de aprendizaje automático y generación de modelos de TensorFlow Lite fueron realizados utilizando el lenguaje Python en cuadernos ejecutados mediante Google Colaboratory. Todo el código se encuentra en un solo cuaderno, por lo que las características del sistema fueron las mismas en todos los casos. Las características del sistema asignado por la plataforma para la ejecución del programa se listan a continuación.

- CPU: Intel Xeon @ 2.20GHz.
- GPU: No fue utilizada.
- RAM: 12.68 GB.
- Disco: 107.72 GB.

8.3. Desarrollo del modelo de aprendizaje automático

En esta sección se detalla el proceso de selección de la arquitectura y entrenamiento del modelo que se empleó para la regresión. El proceso fue iterativo y concluyó al obtener una aproximación que satisfizo las expectativas del desarrollador. En esta ocasión, se realizaron tres iteraciones hasta lograr los resultados esperados.

8.3.1. Generación de datos

Dado que lo buscado fue hacer una regresión de una función matemática conocida, los datos fueron auto generados mediante código. La generación de los datos para la variable independiente x consistió en obtener un conjunto de números distribuidos uniformemente en el rango 0 a 2π, con lo que se cubre una oscilación completa de la onda seno. Luego,

los valores obtenidos fueron revueltos aleatoriamente para garantizar que no estuviesen en orden. Esto como parte de una protección para evitar que el modelo tome los datos como una secuencia con la que puede inferir cierto comportamiento que no es deseado durante el entrenamiento. Luego, se obtuvo el valor correspondiente de seno para cada valor de x, representativo de la variable dependiente y esperada. Los datos obtenidos fueron graficados para observar el resultado y son mostrados en la Figura 15.

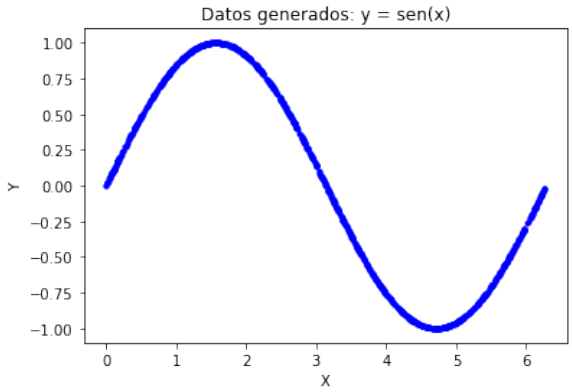


Figura 15: Datos auto generados para entrenamiento de creación de red neuronal.

Luego, fue agregado ruido a la salida y, ya que se buscaba demostrar la capacidad de las redes neuronales para la extracción de patrones y comportamientos debajo de datos con ruido y otros defectos. La gráfica de los datos con ruido se muestra en la Figura 16.

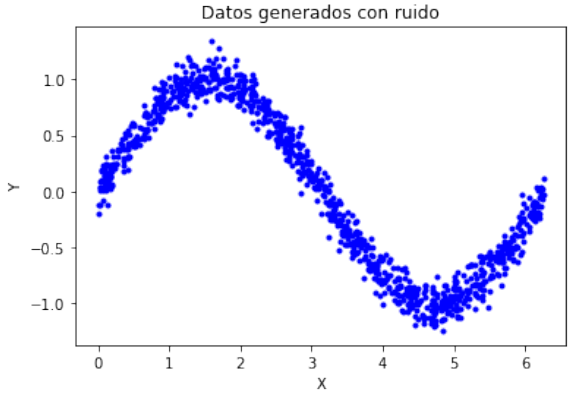


Figura 16: Datos auto generados con ruido incluido.

8.3.2. División de los datos

Dado que se buscaba entrenar una red neuronal, se necesitó dividir los datos obtenidos en tres conjuntos diferentes. Es importante usar datos para entrenamiento, validación y prueba, donde las últimas dos hacen referencia a comparaciones de evaluación del rendimiento del modelo durante y después de su entrenamiento. Los datos se dividieron de la siguiente forma, según recomendación de [8]:

- Entrenamiento: 60 %.
- Validación: 20 %.
- Prueba: 20 %.

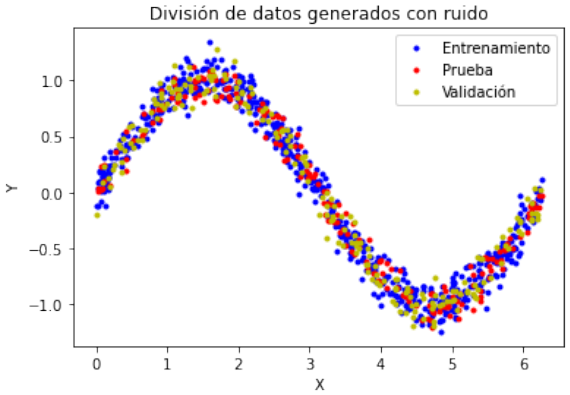


Figura 17: División de datos generados.

8.3.3. Diseño del modelo inicial

La red neuronal diseñada inicialmente surgió puramente de intuición, experiencia y lógica. Además, el proceso fue iterativo por lo que se planteó un modelo básico con el que se podría solucionar el problema y luego se mejoró. El modelo inicial seleccionado se describe en el Cuadro 1.

No. de capa	Tipo de capa	Nodos	Función de activación	Parámetros entrenables
1	Densa	8	ReLU	16
2	Densa	1	Lineal	9

Cuadro 1: Hiper-parámetros de la arquitectura del modelo inicial.

Se utilizó una primera capa densa de 8 nodos con función de activación ReLU para integrar no linealidades en el modelo con la intención de que pudiese representar la función seno de forma adecuada. La siguiente capa es únicamente de salida, por lo que no cuenta con función de activación no lineal y solo ejecuta una suma de los valores resultantes de la primera capa, lo que se hace para obtener un escalar como salida y es conocido como una función de activación lineal o *passthrough*. También se puede observar que la cantidad de parámetros entrenables de cada capa fue pequeño, sumando 25 en total para el modelo, lo que nos indica poca complejidad.

8.3.4. Entrenamiento de modelo inicial

Parámetros de entrenamiento

En este punto, con el modelo definido, fue necesario definir hiper-parámetros de entrenamiento. Este proceso también puede ser iterativo, aunque colocando un número amplio al inicio se puede dar una idea mejor de lo que se necesita para el modelo en la siguiente iteración. En este caso, los parámetros seleccionados para el modelo inicial se muestran en el Cuadro 2.

Modelo inicial	
Optimizador	Adam
Función de pérdida	Error cuadrático medio
Métrica	Error absoluto medio
Épocas	500
Tamaño de lote	64

Cuadro 2: Hiper-parámetros de entrenamiento del modelo inicial.

El entrenamiento del modelo con los parámetros mostrados duró 33.907 segundos en Google Colaboratory sin utilización de aceleradores de hardware.

Resultados del entrenamiento

El entrenamiento de la red neuronal diseñada brindó los resultados mostrados en las Figuras 18, 19 y 20. Los resultados no fueron los esperados. La Figura 18, que muestra un comportamiento positivo de descenso de la pérdida en las épocas iniciales, evidenció que luego de alrededor de 100 épocas, el entrenamiento se estanca, aunque no es posible observar claramente el comportamiento de los datos luego de esa etapa.

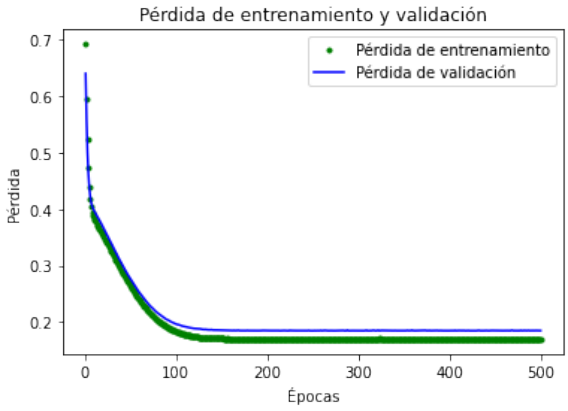


Figura 18: Pérdidas durante el entrenamiento.

Se decidió graficar únicamente los datos de la época 20 en adelante, para una visualización

más adecuada de la etapa de interés. La Figura 19, muestra que las pérdidas de entrenamiento y de validación se mantienen constantes aún con el avance de las épocas de entrenamiento.

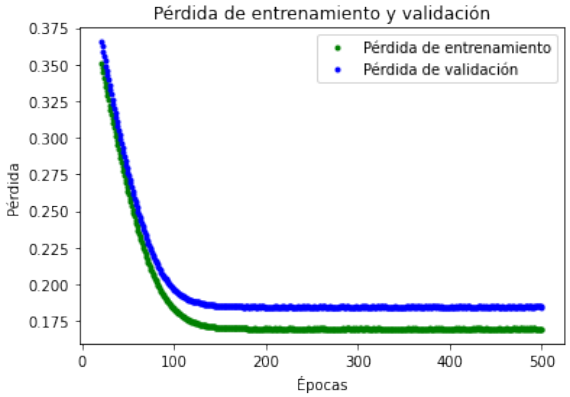


Figura 19: Pérdidas durante el entrenamiento (saltando épocas).

En la Figura 20, se puede observar cómo las métricas mostraron un comportamiento igual al de las pérdidas. Se estabilizaron en un punto y no cambiaron aún con el paso de las épocas. Además, el error absoluto medio fue demasiado grande, dados los valores que puede tomar la función seno, con lo que se evidencia que los valores que se obtendrán a la salida no serán adecuados. Esto se puede deber a que el modelo no es suficientemente complejo como para representar fielmente la función seno. Otras causas podrían ser que el optimizador ya no puede encontrar una dirección de descenso de la superficie de pérdidas, que se haya llegado a un valor óptimo o que se haya llegado a un mínimo local no deseado. La evolución de los valores de la función de pérdida y métrica durante el entrenamiento no son concluyentes.

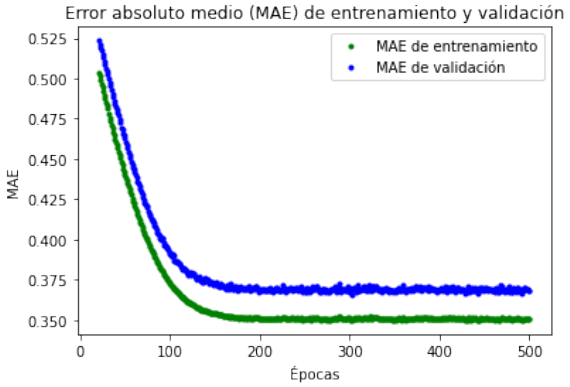


Figura 20: Métricas de durante el entrenamiento.

Luego, para verificar el comportamiento del modelo y llegar a una conclusión, se procedió a obtener las predicciones del modelo para los datos de prueba y compararlos con los valores de salida reales. La Figura 21 muestra cómo el modelo no es capaz de reproducir las no linealidades de la función seno de forma adecuada, lo que llevó a tomar la decisión de elaborar un modelo más complejo.

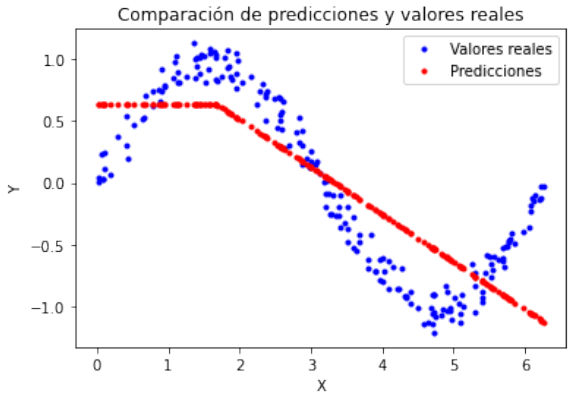


Figura 21: Valores reales comparados con predicciones del modelo inicial desarrollado.

8.3.5. Diseño de un modelo mejorado

Luego de la visualización de los resultados anteriores, se decidió diseñar un modelo más complejo al aumentar la cantidad de capas densas y la cantidad de nodos para cada una, siempre contando con una capa de un único nodo para la salida. En el Cuadro 3, se puede observar que se duplicó el número de capas densas con función de activación ReLU, así como el número de nodos para cada capa. De este modelo se esperó una respuesta más adecuada a la función seno, dado que se aumentaron sus capacidades de imitar su comportamiento.

No. de capa	Tipo de capa	Nodos	Función de activación	Parámetros entrenables
1	Densa	16	ReLU	32
2	Densa	16	ReLU	272
3	Densa	1	Lineal	17

Cuadro 3: Hiper-parámetros de la arquitectura del modelo mejorado.

En el cuadro anterior se puede observar que la complejidad del modelo aumentó ya que se cuenta con mayor cantidad de parámetros entrenables. Con ello, aumentó la versatilidad del modelo para imitar comportamientos no lineales, aunque también la complejidad de su entrenamiento.

8.3.6. Entrenamiento de modelo mejorado

Parámetros de entrenamiento

Los parámetros de entrenamiento para el nuevo modelo se mantuvieron intactos ya que fueron un buen inicio. Además, se promovió que podrían ser modificados luego si se determinaba que requerían cambios.

Modelo mejorado	
Optimizador	Adam
Función de pérdida	Error cuadrático medio
Métrica	Error absoluto medio
Épocas	500
Tamaño de lote	64

Cuadro 4: Hiper-parámetros de entrenamiento del modelo mejorado.

El entrenamiento realizado a la nueva arquitectura de la red neuronal duró 34.632 segundos en Google Colaboratory sin utilización de aceleradores de hardware.

Resultados del entrenamiento

Los resultados se muestran en la Figura 22, donde a la izquierda se puede observar la evolución de la pérdida y a la derecha la evolución de la métrica seleccionada. Se omitieron las primeras 20 épocas para una mejor visualización a detalle del final del entrenamiento ya que inicialmente se muestra un descenso brusco de ambos indicadores que es esperado aunque su visualización es innecesaria para evaluar el entrenamiento.

La figura muestra pérdidas y métricas descendientes aunque a un ritmo lento. Los comportamientos de ambos indicadores muestran que los valores tienden a acercarse para los conjuntos de datos de entrenamiento y validación. Además, los valores mostrados son a escalas pequeñas, lo que es importante en el caso de la función seno dado su rango. El entrenamiento fue detenido antes de un estancamiento en la evolución de la pérdida y la métrica, indicando que el modelo aún es bueno para hacer inferencia sobre datos con los que no fue entrenado, lo que implica que no existe sobre-ajuste.

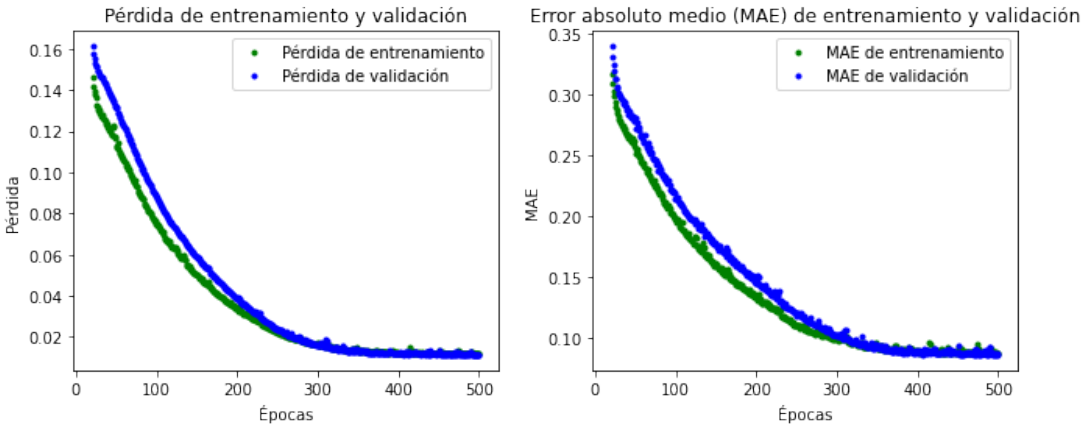


Figura 22: Métricas durante el entrenamiento del modelo mejorado.

Luego de observar los resultados de entrenamiento, fue necesario comprobar el comportamiento del modelo utilizando los datos de prueba para obtener inferencias. En la figura siguiente se cuenta con una comparación de los datos reales (del conjunto de datos de prueba)

contra las predicciones. En ella se puede observar que el nuevo modelo muestra un comportamiento bastante similar a los esperado, por lo que fue seleccionado como el adecuado para aplicación que se busca realizar.

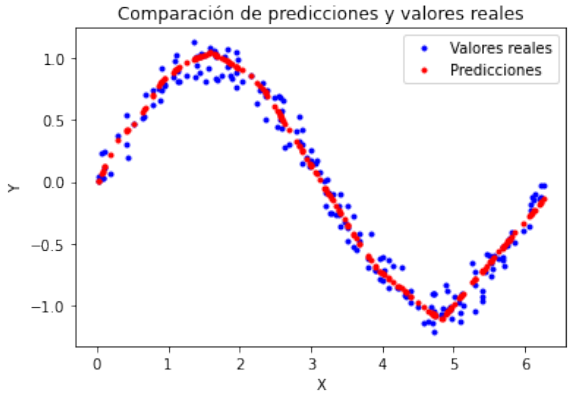


Figura 23: Valores reales comparados con predicciones del modelo mejorado desarrollado.

También fue posible observar cómo los resultados obtenidos parecieron despreciar el efecto del ruido en el conjunto de datos de entrenamiento. Esto evidenció la capacidad de las redes neuronales de extraer información valiosa de los datos con ruido, como lo hacen los modelos de inferencia con buena generalización. Además, producto de las funciones de activación utilizadas, el modelo presenta dificultad para modelar las no linealidades de la función a la cual se aplica la regresión, generando picos y líneas rectas en lugar de curvas. Aún así, el modelo es suficientemente bueno para una demostración.

8.4. Generación de un modelo de TensorFlow Lite para el modelo mejorado

Entre las principales problemáticas de la utilización de microcontroladores se encuentra la limitante de espacio en memoria, puesto que es muy pequeña. Por ello, se realizó una conversión del modelo a un formato más eficiente para utilización en dispositivos de memoria limitada. En este paso se utilizó el convertidor de TensorFlow Lite en Python y se ejecutó utilizando Google Colaboratory.

8.4.1. Conversión del modelo

La conversión del modelo consistió en dos vertientes diferentes. Una variante consistió en convertir el modelo original al formato TensorFlow Lite, sin pasos adicionales. Una segunda variante consistió en convertir el modelo utilizando optimización de espacio. Se utilizó cuantificación de números enteros que convirtió los parámetros del modelo de tipo float a tipo int8. Esto llevó a contar con tres modelos en este punto.

1. Modelo de TensorFlow.

2. Modelo de TensorFlow Lite.

3. Modelo de TensorFlow Lite con cuantificación.

8.4.2. Comparación de los modelos

La comparación de los modelos se realizó desde dos puntos de vista: la exactitud y el espacio que ocupaban en memoria. El punto de vista de la exactitud requirió de hacer inferencia para los datos de prueba para los modelos y compararlos entre sí y con los valores reales. Ejecutar los modelos de TensorFlow Lite requirió de intérpretes de dicho formato y cuantificación o des-cuantificación, según fuera necesario. Dichos segmentos de código fueron desarrollados en Python ejecutado a través de Google Colaboratory.

En la Figura 24, se puede observar el comportamiento, del cual se obtuvo buenos resultados para todos los modelos. Los modelos convertidos (cuantificado y no cuantificado) muestran comportamientos sin cambios significativos respecto del comportamiento del modelo original, con lo cual también se acoplan de forma adecuada a los datos de prueba, aunque siguen teniendo dificultades mínimas para modelar algunas no linealidades como los máximos y mínimos, que terminan siendo picos.

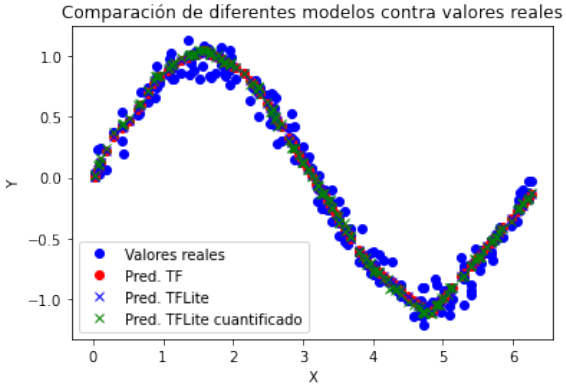


Figura 24: Comparación de inferencias de modelos contra valores reales.

Si bien la optimización anterior puede provocar una disminución del rendimiento del modelo, en este caso no fue significativa. Por otro lado, dichas transformaciones y optimizaciones sí tienen un efecto importante en el ámbito del espacio que ocupan en memoria y favorecen la velocidad de ejecución, aunque ese aspecto no se evaluó en este trabajo. El cuadro siguiente demuestra que al utilizar la conversión a formato TensorFlow Lite y optimización por cuantificación se redujo el espacio a utilizar en memoria por alrededor de un 40 %. Además, el rendimiento del modelo no sufrió cambios significativos al comparar la pérdida (MSE) y la métrica (MAE) en inferencias a partir de los datos de prueba.

Modelo	Pérdida (MSE)	Métrica (MAE)	Tamaño (bytes)
TensorFlow	0.011479	0.086847	4096
TensorFlow Lite	0.011479	0.086847	2788
TensorFlow Lite Cuantificado	0.013047	0.093625	2488

Cuadro 5: Comparación de los modelos obtenidos a partir de la pérdida, la métrica y el tamaño.

La diferencia en este caso no fue enorme al realizar cuantificación, con solo 300 bytes respecto del modelo de TensorFlow Lite sin cuantificación. Esto se debe a que el modelo seleccionado no es especialmente complejo a comparación de lo que se puede generar en otras aplicaciones. Esto significa que la diferencia de tamaño en modelos más complejos podría ser más significativa, por lo que es una ventaja la utilización de este procedimiento. Aún así, se puede observar que el rendimiento de los modelos es muy similar en el conjunto de datos de prueba y sí se reduce el tamaño del modelo respecto del modelo original, lo que es un efecto positivo al contar con limitaciones de memoria.

8.4.3. Modelos resultantes como archivos fuente en C

Los modelos de TensorFlow Lite deben ser convertidos a un formato comprensible para los microcontroladores. Por ello, se transformó el modelo mejorado del formato TensorFlow Lite sin cuantificación a un archivo fuente en C, donde se guarda la información del modelo y que puede interpretarse en microcontroladores como los seleccionados. La salida de la conversión se obtuvo con la siguiente forma:

```
unsigned char g_model[] = {
  0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x14, 0x00, 0x20, 0x00,
  0x04, 0x00, 0x08, 0x00, 0x0c, 0x00, 0x10, 0x00, 0x14, 0x00, 0x00, 0x00,
  ...                                                             ...
  0x74, 0x3a, 0x30, 0x00, 0x02, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff,
  0x01, 0x00, 0x00, 0x00, 0xfc, 0xff, 0xff, 0xff, 0x04, 0x00, 0x04, 0x00,
  0x04, 0x00, 0x00, 0x00
};
unsigned int g_model_len = 2788;
```

Luego, se procedió a convertir también el modelo mejorado del formato TensorFlow Lite con cuantificación a un archivo fuente en C. El resultado fue el siguiente:

```
unsigned char g_model[] = {
  0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x14, 0x00, 0x20, 0x00,
  0x1c, 0x00, 0x18, 0x00, 0x14, 0x00, 0x10, 0x00, 0x0c, 0x00, 0x00, 0x00,
  ...                                                             ...
  0x0c, 0x00, 0x10, 0x00, 0x0f, 0x00, 0x00, 0x00, 0x08, 0x00, 0x04, 0x00,
  0x0c, 0x00, 0x00, 0x00, 0x09, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x09
};
```

```
unsigned int g_model_len = 2488;
```

Los resultados obtenidos como texto fueron copiados y utilizados en Arduino IDE para el desarrollo de la aplicación.

8.5. Desarrollo y conversión de un modelo más compacto

Luego de desarrollar el modelo mejorado, se decidió desarrollar un nuevo modelo. El modelo mejorado mostró un funcionamiento adecuado aunque mostraba ciertas dificultades para replicar comportamientos de la función seno al realizar inferencias en Python. Además, en el modelo anterior se utilizó la función de activación ReLU, que parece tener un comportamiento contrario al que se desea generar, pues retorna curvas muy diferentes a las esperadas. Por ello, se desarrolló un modelo con una arquitectura más sencilla con una función de activación diferente, a manera de crear un modelo más compacto que tuviese la capacidad de hacer inferencias para el seno de los valores de entrada.

8.5.1. Arquitectura de un modelo más compacto

En el Cuadro 6 se puede observar la nueva arquitectura utilizada y sus parámetros.

No. de capa	**Tipo de capa**	**Nodos**	**Función de activación**	**Parámetros entrenables**
1	Densa	16	Tangente Hiperbólica	32
2	Densa	1	Lineal	17

Cuadro 6: Hiper-parámetros de la arquitectura del modelo compacto.

Se utilizó la función de activación tangente hiperbólica debido a que muestra comportamientos más similares a la función seno y devuelve valores negativos, lo que podría facilitar la regresión que se desea hacer al realizarla con un modelo más sencillo. Luego, se procedió al entrenamiento. Este entrenamiento fue más largo en épocas. Se dio inicio al entrenamiento del modelo con 500 épocas, aunque al observar los resultados se pudo confirmar que la función de pérdida tenía potencial para mayor evolución, al igual que la métrica, por lo que se decidió darle un entrenamiento más largo. El Cuadro 7 muestra los hiper-parámetros de entrenamiento finales utilizados.

Modelo compacto	
Optimizador	Adam
Función de pérdida	Error cuadrático medio
Métrica	Error absoluto medio
Épocas	1500
Tamaño de lote	64

Cuadro 7: Hiper-parámetros de entrenamiento del modelo compacto.

El entrenamiento realizado a la nueva arquitectura de la red neuronal duró 113.572 segundos en Google Colaboratory sin utilización de aceleradores de hardware.

8.5.2. Resultados del entrenamiento

El entrenamiento del nuevo modelo diseñado mostró evoluciones positivas durante el entrenamiento de 1500 épocas. La Figura 25 muestra la evolución de las métricas tomadas durante el entrenamiento del modelo para los datos de entrenamiento y de prueba. Se probó realizar el entrenamiento del modelo con más épocas, pero al aumentar este parámetro solo se obtenían modelos con probable sobre-ajuste, ya que las métricas de los datos de entrenamiento mejoraban, aunque las de datos de validación empeoraban. Utilizando 1500 épocas se logró que el modelo presentara aún una buena capacidad de inferencia para datos distintos a los de entrenamiento y con una pérdida pequeña. Con el fin de corroborar lo observado en la gráfica de entrenamiento, se realizó inferencia utilizando el modelo obtenido sobre los datos de prueba.

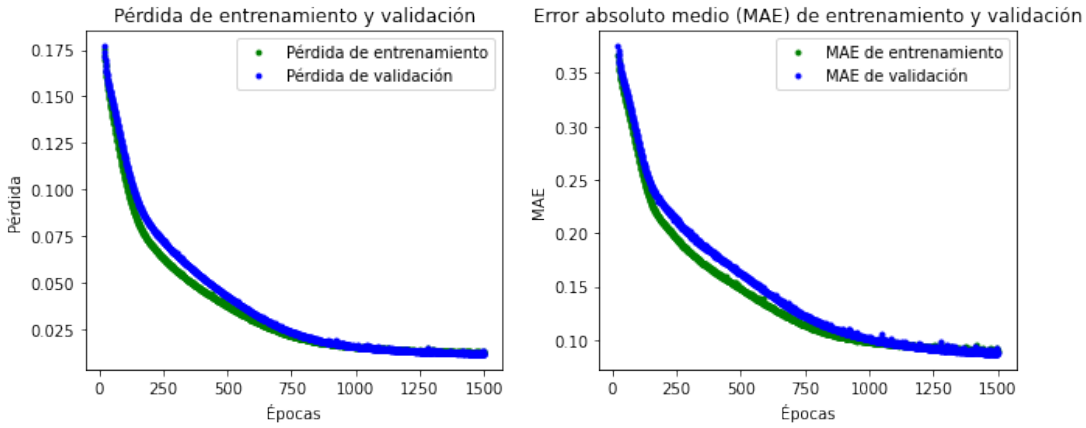


Figura 25: Métricas durante el entrenamiento del modelo compacto.

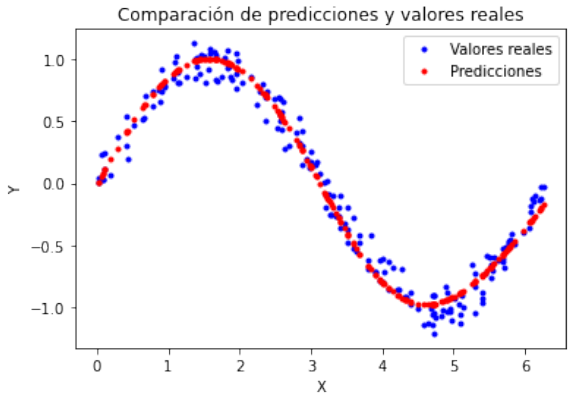


Figura 26: Valores reales comparados con predicciones del modelo compacto desarrollado.

La Figura 26 muestra el efecto del entrenamiento realizado sobre la capacidad de inferen-

cia del modelo. El modelo muestra una buena aproximación de la función seno y no evidencia dificultad para replicar comportamientos no lineales naturales de la función. También muestra un comportamiento más suave que el modelo desarrollado anteriormente. Observando los resultados obtenidos, se decidió continuar con la implementación del nuevo modelo desarrollado.

8.5.3. Generación de modelos de TensorFlow Lite para el modelo compacto

Se siguió el mismo flujo de trabajo que en el modelo anterior para el generación de los modelos de TensorFlow Lite. Se obtuvieron 3 incluyendo el diseñado. Dos modelos extras surgieron de realizar conversión a TensorFlow Lite del modelo original, uno sin cuantificación y otro con cuantificación. A continuación se muestra la comparación realizada para evaluar el cambio de exactitud en la inferencia al transformar el modelo.

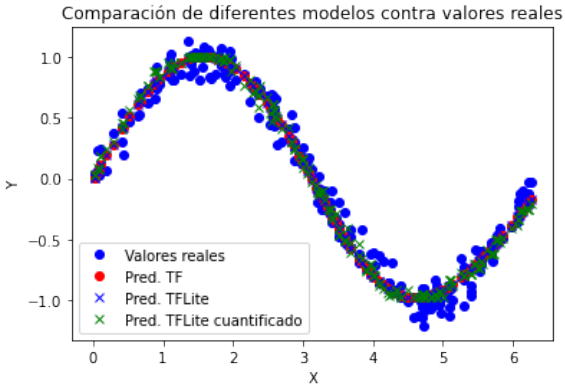


Figura 27: Comparación de inferencias de modelos compactos contra valores reales.

La Figura 27 muestra que los modelos convertidos a TensorFlow Lite no difieren en cuanto a exactitud significativamente respecto del modelo original, lo que nos lleva ahora a comparar el tamaño de los modelos y su exactitud mediante métricas de rendimiento.

Modelo	Pérdida (MSE)	Métrica (MAE)	Tamaño (bytes)
TensorFlow	0.011479	0.086847	4096
TensorFlow Lite	0.011479	0.086847	1524
TensorFlow Lite Cuantificado	0.013047	0.093625	1896

Cuadro 8: Comparación de los modelos compactos obtenidos a partir de la pérdida, la métrica y el tamaño.

El Cuadro 8 muestra que los modelos obtenidos no difieren significativamente en exactitud respecto del modelo original, como se puede observar en las columnas de pérdida y métrica. Si bien el modelo cuantificado es el menos exacto, la diferencia de precisión es despreciable. Luego, al observar la comparación de tamaño de los modelos se puede observar que los modelos convertidos a TensorFlow Lite muestran una disminución significativa de tamaño respecto del modelo original.

El modelo convertido a TensorFlow Lite sin cuantificación muestra menor tamaño que el modelo con cuantificación, aunque por una diferencia de únicamente 364 bytes. Aún siendo esta diferencia importante al compararla con el tamaño de los modelos, se promueve implementar el modelo cuantificado. Esto se debe a que la cuantificación facilita la ejecución del modelo para el microcontrolador, aumentando la velocidad de inferencia, que es fundamental en muchas aplicaciones por lo que se aprovecha la cuantificación para tomar ventaja en este aspecto.

Además, es importante mencionar que la conversión de un modelo de TensorFlow Lite con cuantificación resulta normalmente en un modelo de menor tamaño que sin cuantificación. En este caso que el modelo con cuantificación contase con un mayor tamaño es producto de la utilización de la función tangente hiperbólico. La aproximación de dicha función de activación en un modelo cuantificado requiere de almacenamiento de información dentro de los datos del modelo en TensorFlow Lite, por lo que se ocupa mayor espacio.

Los resultados mostrados llevan a considerar que aunque se puedan implementar modelos un tanto más complejos, siempre es valioso dedicar tiempo a buscar soluciones más sencillas. Esta solución más sencilla libera recursos del dispositivo en uso para usarlos en otras tareas que podrían ser necesarias, de forma que es más eficiente su implementación. Además, es probable que modelos más sencillos brinden mejores inferencias. Esto se debe a que hay muchos parámetros influyentes que se pueden variar en un modelo para obtener mejores inferencias sin hacer el modelo más complejo, aunque hay casos donde es inevitable.

Luego, los modelos fueron convertidos a código fuente en C para su ejecución en el microcontrolador. A continuación, se muestra un resumen del código fuente en C obtenido para el modelo compacto sin cuantificación.

```
unsigned char g_model[] = {
  0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x14, 0x00, 0x20, 0x00,
  0x04, 0x00, 0x08, 0x00, 0x0c, 0x00, 0x10, 0x00, 0x14, 0x00, 0x00, 0x00,
  ...                                                                 ...
  0x02, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0x01, 0x00, 0x00, 0x00,
  0xfc, 0xff, 0xff, 0xff, 0x04, 0x00, 0x04, 0x00, 0x04, 0x00, 0x00, 0x00
};
unsigned int g_model_len = 1524;
```

A continuación, se muestra el código fuente en C representativo del modelo compacto con cuantificación.

```
unsigned char g_model[] = {
  0x20, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x00, 0x00, 0x00, 0x00,
  0x14, 0x00, 0x20, 0x00, 0x1c, 0x00, 0x18, 0x00, 0x14, 0x00, 0x10, 0x00,
  ...                                                                 ...
  0x0f, 0x00, 0x00, 0x00, 0x08, 0x00, 0x04, 0x00, 0x0c, 0x00, 0x00, 0x00,
  0x09, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x09
};
unsigned int g_model_len = 1896;
```

8.6. Implementación de los modelos en Microcontroladores

La implementación de la ejecución de los modelos en microcontroladores, en este caso usando los dispositivos Arduino Nano 33 BLE Sense y Espressif ESP32 DevKitC, requirió de ciertas piezas de código que llevaron a cabo funciones especiales que ayudan a interpretar el formato especial de los modelos desarrollado. El desarrollo de código se realizó en Arduino IDE.

8.6.1. Ejecución en microcontroladores

La creación de una aplicación para la ejecución de los modelos requirió de varias acciones y segmentos de código que se listan y describen a continuación:

1. Instalación de bibliotecas:
   a) Arduino_TensorflowLite: biblioteca que permitió la realización de inferencias mediante la ejecución de modelos de aprendizaje automática de forma local en el dispositivo. Se instaló por medio del Gestor de Librerías de Arduino IDE.
2. Inclusión de librerías: en estas librerías se encontró información asociada a las operaciones utilizadas por el intérprete para ejecutar el modelo, información de depuración de salidas y código de control y ejecución de modelos.
   a) TensorFlowLite.h
   b) tensorflow/lite/micro/all_ops_resolver.h
   c) tensorflow/lite/micro/micro_error_reporter.h
   d) tensorflow/lite/micro/micro_interpreter.h
   e) tensorflow/lite/schema/schema_generated.h
   f) tensorflow/lite/version.h
3. Definición de variables globales: en esta sección se definieron punteros a varias entidades importantes de la ejecución del modelo. Es una buena práctica definirlas dentro de un namespace, como clases a las cuales se asignan instancias en las que se asignan los punteros.
   a) Definición de puntero de clase ErrorReporter.
   b) Definición de puntero de clase Model.
   c) Definición de puntero de clase MicroInterpreter.
   d) Definición de puntero TfLiteTensor para las entradas y salidas del modelo.
   e) Definición de valor de tamaño de la arena de tensores.
   f) Definición de arena de tensores.
4. Función de *setup*: en ella se realizaron operaciones para construir al interprete y las herramientas que necesita para ejecutar el modelo.
   a) Registro de configuración.

b) Obtener operaciones de implementación necesarias mediante AllOpsResolver.

c) Construir un intérprete para correr el modelo.

d) Asignar memoria de la arena de tensores para los tensores del modelo.

e) Obtener punteros a los tensores de entrada y salida del modelo.

5. Función de *loop*: en esta sección se realizó la ejecución del modelo mediante un intérprete. Esta sección se ejecuta en forma de bucle, por lo que el modelo de aprendizaje automático se ejecutó continuamente en cada ciclo.

   a) Obtención de variables de entrada al modelo.

   b) Cuantificación de variables de entrada (si es necesario).

   c) Colocar la entrada cuantificada en el tensor de entrada del modelo.

   d) Inferir utilizando el modelo.

   e) Reportar cualquier error.

   f) Obtener la salida cuantificada del modelo del tensor de salida.

   g) Des-cuantificación del valor de salida del modelo (si es necesario).

   h) Manejo del valor de salida para realizar otras acciones.

Es importante mencionar que las secciones del código mencionadas en el listado pueden tener modificaciones y añadidos dependiendo de las funciones que se desean realizar con la aplicación.

8.6.2. Obtención y manejo de entradas y salidas al modelo

Variable de entrada

La variable de entrada hacia los modelos fue obtenida mediante la lectura del ADC de 12 bits de un pin analógico de los dispositivos mediante la función analogRead(). Dicho valor se encontraba en el rango de 0 a 4095, por lo que fue necesario realizar una razón proporcional para obtener su equivalencia en el rango de 0 a 2π. Se obtuvo un valor de tipo float que ya fue manipulado para cuantificación cuando se requirió y utilizado como entrada al modelo para hacer inferencia sobre su valor. Los valores relacionados a la obtención de la variable de entrada también fueron enviados por comunicación serial para ser mostrados en el Monitor Serial de Arduino IDE.

Variable de salida

La variable de salida de los modelos, luego de la inferencia, pasa por un proceso inverso al de la variable de entrada. Luego de des-cuantificar el valor, se utiliza la salida para hacer un mapeo hacia la intensidad de un LED. Los valores de salida del modelo se esperan en el rango de -1 a 1, por lo que se aplica una interpolación para obtener el equivalente del valor de salida en el rango 0 a 255. Luego, el valor obtenido de intensidad de la luz es utilizado mediante la función analogWrite() para encender el LED integrado en el dispositivo a la

intensidad deseada. Los valores relacionados a la obtención de la variable de salida también fueron enviados por comunicación serial para ser mostrados en el Monitor Serial de Arduino IDE.

8.7. Pruebas de la aplicación

Al finalizar la programación de la aplicación en Arduino IDE, se procedió a realizar pruebas para evaluar los resultados. Las pruebas evaluaron las dos principales formas de salida de información del dispositivo: salida visual (LED) y salida serial (Arduino Serial Monitor y RealTerm Serial). En ella se buscó comprobar que los resultados obtenidos fueran sensatos y congruentes con el modelo desarrollado y lo esperado de la aplicación.

Los datos utilizados para la ejecución de la aplicación fueron extraídos del dispositivo mediante comunicación serial. Los datos presentados son separados por comas y son lectura de ADC, valor de x equivalente, predicción y e intensidad del LED, respectivamente. Se verificó que los datos se mostraran en el formato deseado en el monitor serial de Arduino IDE, como se muestra en la Figura 28.

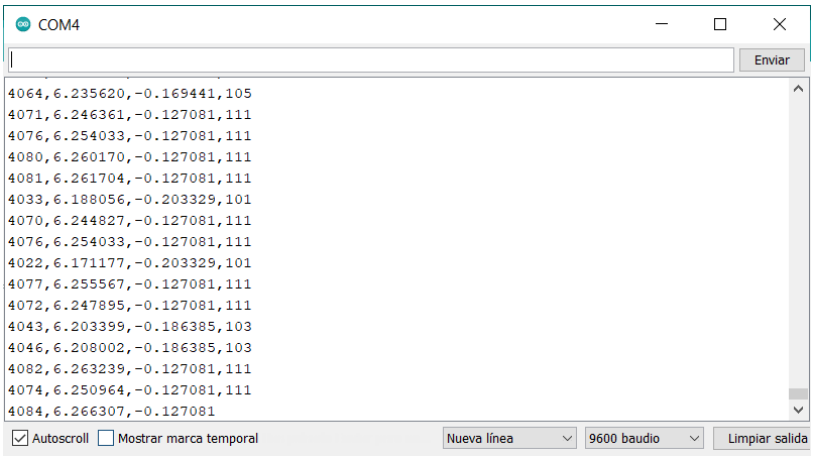


Figura 28: Visualización de resultados en monitor serial de Arduino IDE.

Luego, se utilizó la aplicación RealTerm Serial, donde también se puede observar la comunicación serial, aunque la principal función es facilitar la captura de datos. En esta aplicación se guardaron los datos obtenidos de la comunicación serial en un archivo de texto con valores separados por comas. Dicho archivo sería luego importado a MATLAB para obtener los datos y poder analizar los resultados mediante gráficas.

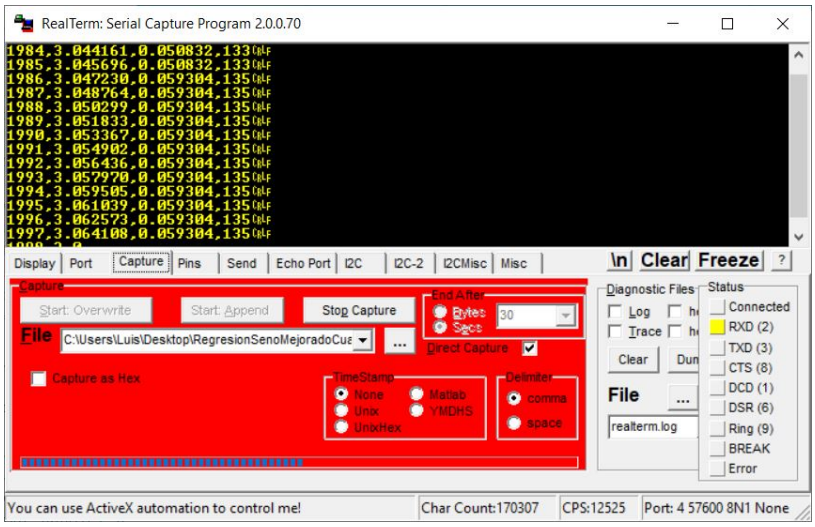


Figura 29: Captura de datos seriales mediante la utilización de RealTerm.

Los datos obtenidos permitieron graficar el mapeo de la señal de voltaje de entrada a la variable x necesaria en la ejecución del modelo. La Figura 30 muestra la relación entre la lectura que obtiene el ADC en el dispositivo y dicha variable. Es una relación lineal, aunque los valores que el ADC puede tomar son finitos, solo 4096, con lo que no se obtiene todo el rango sino solo una cantidad de valores suficiente para representar el intervalo.

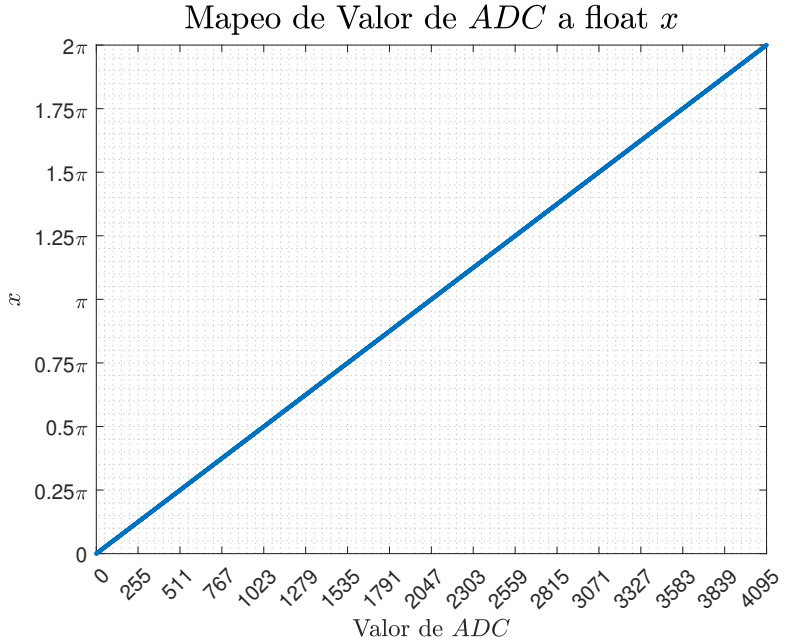


Figura 30: Mapeo de valores de ADC a valores de x a usar como entrada para inferencias del modelo mejorado.

La salida visual hace referencia a un mapeo de la salida predicha del modelo hacia un valor que define la intensidad de un LED en el dispositivo. Dado los valores esperados de una

función seno, se mapea el mínimo esperado a 0 y el máximo a 255. Es importante mencionar que las salidas visuales mediante LED mostraron comportamiento adecuados al reflejar el comportamiento de la salida del modelo mediante luz. Aún así, la luz no refleja fielmente la inferencia de los modelos cuando se obtienen salidas fuera del rango de valores que se esperan de la función seno. A continuación se muestra un ejemplo de lo sucedido.

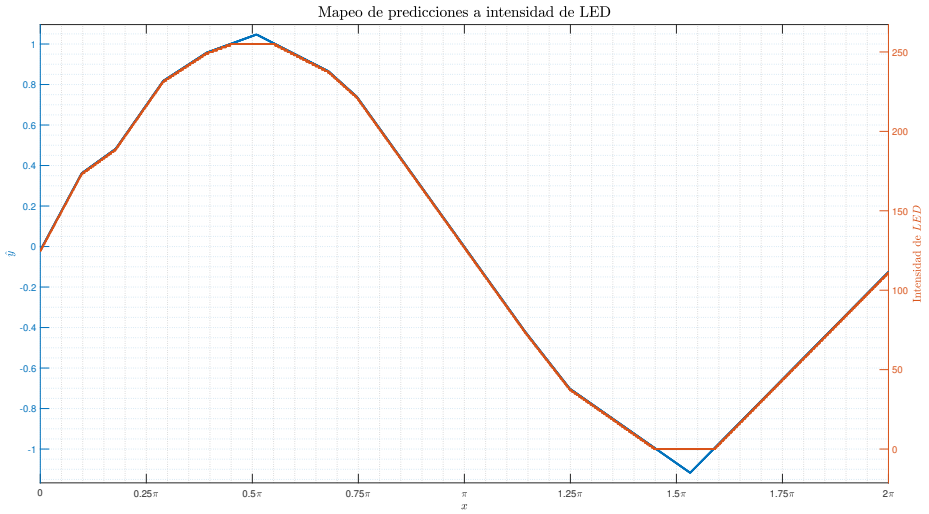


Figura 31: Comparación de intensidad de LED contra predicciones del modelo mejorado.

La Figura 31 fue obtenida a partir de inferencias del modelo mejorado sin cuantificación implementado en el microcontrolador Arduino Nano 33 BLE Sense. Esta figura muestra la problemática de predecir los rangos de los valores de salida de un modelo hasta el momento de su ejecución. Esta problemática se reduce desarrollando un mejor modelo que replique de mejor manera el comportamiento para la función que se desea imitar, aunque sigue siendo complicado estimar los máximos y mínimos sin antes inferir con el modelo. Al utilizar un LED como salida, esta aplicación no toma relevancia este problema, aunque si se utilizaran otras salidas como motores u otros dispositivos podría llegar a ser problemático.

8.7.1. Inferencia de modelos utilizando Arduino Nano 33 BLE Sense

La implementación de los modelos se dio mediante la utilización del código desarrollado en Arduino IDE explicado en secciones anteriores. El único cambio realizado entre modelos fue el reemplazo del código fuente en C del modelo anterior por el código fuente del nuevo modelo. Luego, se realizaron pruebas para capturar la salida por el monitor serial. Las pruebas a la salida del modelo fueron realizadas mediante captura de datos utilizando Realterm. Los datos fueron graficados en MATLAB y se evaluó la función de pérdida y la métrica sobre las inferencias de los modelos, siendo comparadas con los valores reales de la función seno.

Inferencia utilizando modelo mejorado

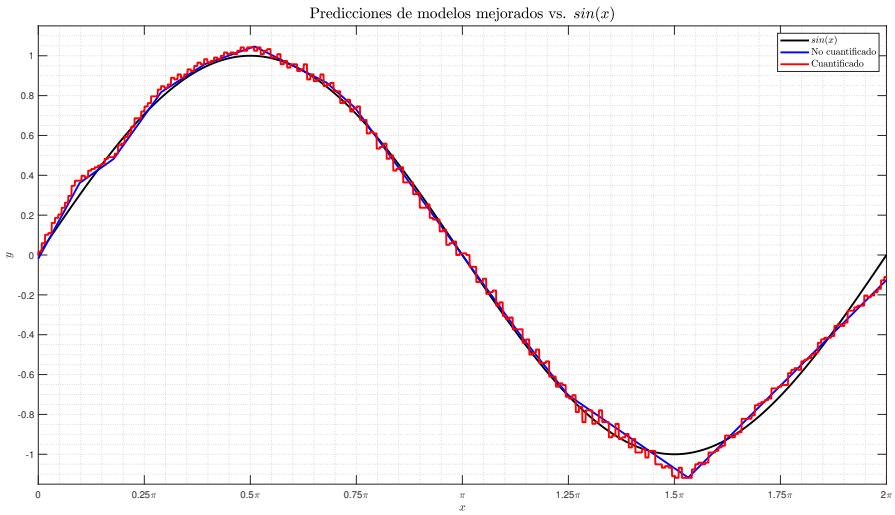


Figura 32: Comparación de inferencias de los modelos mejorados en Arduino Nano 33 BLE Sense.

Modelo	Pérdida (MSE)	Métrica (MAE)
No cuantificado	0.0012	0.0259
Cuantificado	0.0018	0.0333

Cuadro 9: Comparación de los modelos mejorados usando Arduino Nano 33 BLE Sense.

Inferencia utilizando modelo compacto

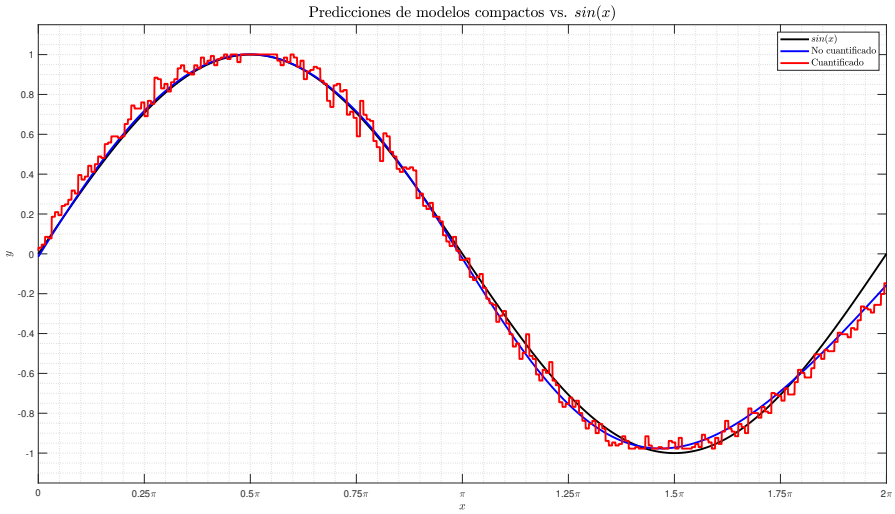


Figura 33: Comparación de inferencias de los modelos compactos en Arduino Nano 33 BLE Sense.

Modelo	Pérdida (MSE)	Métrica (MAE)
No cuantificado	0.0014	0.0242
Cuantificado	0.0030	0.0416

Cuadro 10: Comparación de los modelos compactos usando Arduino Nano 33 BLE Sense.

8.7.2. Inferencia de modelos utilizando Espressif ESP32 DevKitC

Inferencia utilizando modelo mejorado

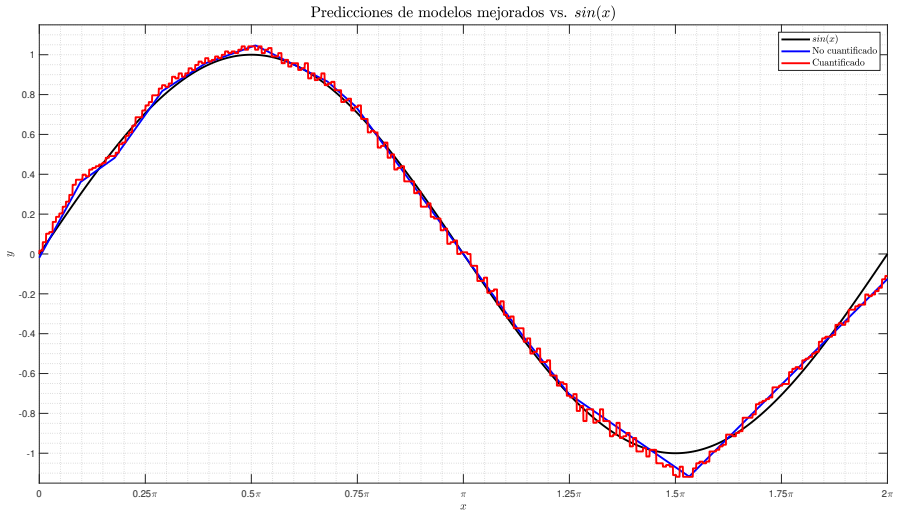


Figura 34: Comparación de inferencias de los modelos mejorados en Espressif ESP32 DevKitC.

Modelo	Pérdida (MSE)	Métrica (MAE)
No cuantificado	0.0012	0.0259
Cuantificado	0.0018	0.0333

Cuadro 11: Comparación de los modelos mejorados usando Espressif ESP32 DevKitC.

Inferencia utilizando modelo compacto

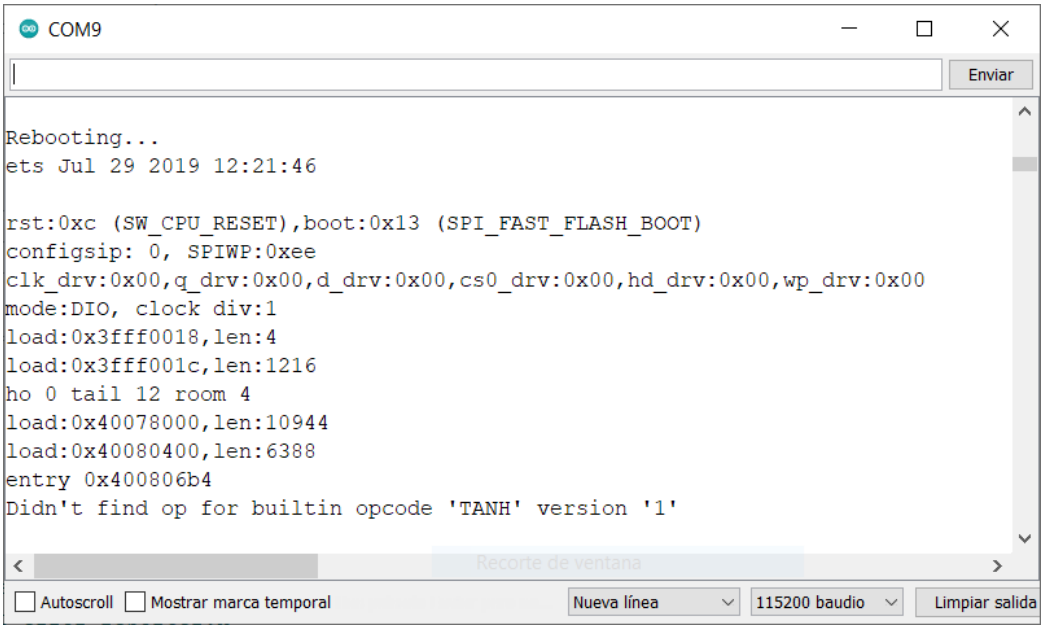


Figura 35: Error de ejecución de modelo compacto en Espressif ESP32 DevKitC.

Discusión

Las Figuras 32, 33 y 34 muestran que al evaluar los datos resultantes de la inferencia de los diferentes modelos obtenidos a través de la captura serial, fue notorio que al ser comparados con la función seno real, los modelos muestran inferencias suficientemente cercanas al comportamiento de la función seno. Los modelos con mayor similitud a la función seno en sus inferencias fueron los modelos compactos. También se evidencia que la ejecución de los modelos no mostró diferencias en sus resultados al implementarlos en dispositivos diferentes. Además, el código no requiere cambios significativos, por lo que se puede cambiar de dispositivo para la ejecución fácilmente.

Luego, al momento de implementar los modelos compactos en el microcontrolador Espressif ESP32 DevKitC se evidenciaron problemas. La librería utilizada por el dispositivo no cuenta con ciertas operaciones para funciones de activación con las que sí cuenta el dispositivo Arduino Nano 33 BLE Sense. Esto se evidencia en la Figura 35, donde el dispositivo muestra un mensaje de error al no encontrar operaciones de implementación de la función de activación tangente hiperbólica. Este problema persistió aún variando las versiones de las librerías para intentar la implementación sin errores. Este problema demuestra la poca y pobre documentación de las librerías y el desarrollo a distinto ritmo de las librerías para diferentes dispositivos. Esta situación lleva a limitaciones más grandes para implementación de modelos en algunos dispositivos, ya que se cuenta con función de activación más limitadas que con el Arduino Nano 33 BLE Sense. Los recursos de la plataforma TensorFlow Lite para microcontroladores son limitados y cuentan con problemáticas importantes en la diversidad de dispositivos, como indica [18], por lo que cobra importancia el resultado obtenido al demostrar de mejor forma dichas limitantes.

Los Cuadros 9, 10 y 11 demuestran que los modelos presentan mejor pérdida (MSE) y métrica (MAE) al compararlos con los valores de seno real que al compararlos con los datos de entrenamiento. Esto evidencia que los modelos son buenos generalizando sus inferencias a partir de datos con ruido en el entrenamiento aún al ejecutarlos en microcontroladores. Por otro lado, los resultados de los modelos cuantificados no muestran diferencia significativa en la pérdida y en la métrica respecto de los modelos no cuantificados, aunque gráficamente si tienen efectos notorios. Esto significa que si el modelo cuantificado presenta un menor tamaño o diferencia poco significativa respecto del modelo no cuantificado, se promueve la ejecución del modelo optimizado por cuantificación para obtener beneficios en la velocidad de ejecución del modelo. Estos resultados se vieron reflejados en todos los modelos ejecutados.

CAPÍTULO 9

Aplicaciones para modelos de aprendizaje automático simples en microcontroladores: Reconocimiento de patrones

9.1. Diseño experimental

Este experimento consistió en realizar una aplicación de reconocimiento de patrones de movimiento en un Arduino Nano 33 BLE Sense a través de la ejecución de un modelo de aprendizaje automático simple. Las señales de entrada para el modelo son obtenidas mediante la utilización de la IMU integrada que posee el dispositivo y el valor de salida es representado mediante texto y representaciones numéricas en el monitor serial del Arduino IDE, además de mostrar respuestas visuales a través del LED integrado. Se realizó como experimentación para ejemplificar recolección de datos y demostrar diferentes acercamientos para lograr resultados similares, como los logrados en el ejemplo "Magic Wand"de [8].

El flujo de trabajo empleado se describe a continuación:

1. Análisis de ejemplos previos.
2. Recolección de datos.
    a) Selección de patrones a reconocer.
    b) Selección de variables a recolectar.
    c) Captura serial de lecturas de IMU.
    d) División de los datos.
3. Desarrollo del modelo de aprendizaje automático.
    a) Diseño del modelo.

b) Entrenamiento del modelo.

1) Parámetros de entrenamiento.
2) Resultados del entrenamiento.

c) Inferencia en datos de prueba para evaluación del modelo.

4. Generación de un modelo de TensorFlow Lite.

a) Conversión del modelo.

b) Modelo resultante como archivo fuente en C.

5. Implementación del modelo en Arduino.

a) Creación de una aplicación para ejecución en microcontrolador.

b) Obtención y manejo de entradas y salidas al modelo.

6. Pruebas de la aplicación.

9.2. Análisis de ejemplos previos

Se procedió a analizar ejemplos de reconocimiento de patrones disponibles en las librerías Harvard_TinyMLx y Arduino_TensorFlowLite, llamados "Magic Wand". En el caso de la primera librería, Harvard_TinyMLx, el modelo utiliza valores de lecturas de aceleración y giroscopio para estimar posiciones de movimiento del dispositivo. Este modelo realiza inferencias para reconocer patrones de dígitos del 0 al 9 realizados en el aire por el usuario.

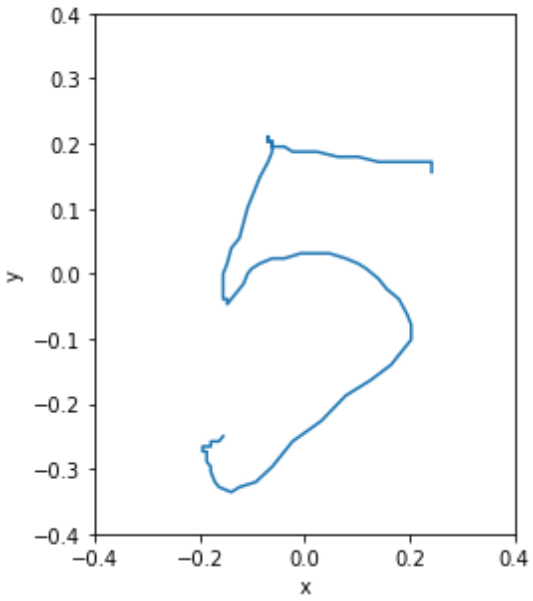


Figura 36: Lectura de datos del sensor transformado a posiciones en aplicación de "Magic Wand"de Harvard Tiny ML.

En la Figura 36, se puede observar la estimación de posiciones realizada en Computadora con los datos de entrenamiento del modelo. Este procedimiento, aunque caro computacionalmente, al momento de ejecutarlo en computadora para ser eficiente y dar resultados congruentes. Luego, para la utilización del modelo planteado por este ejemplo, se utilizan los datos de posiciones obtenidas para realizar un rasterizado, obteniendo una imagen de 32 x 32 bits.

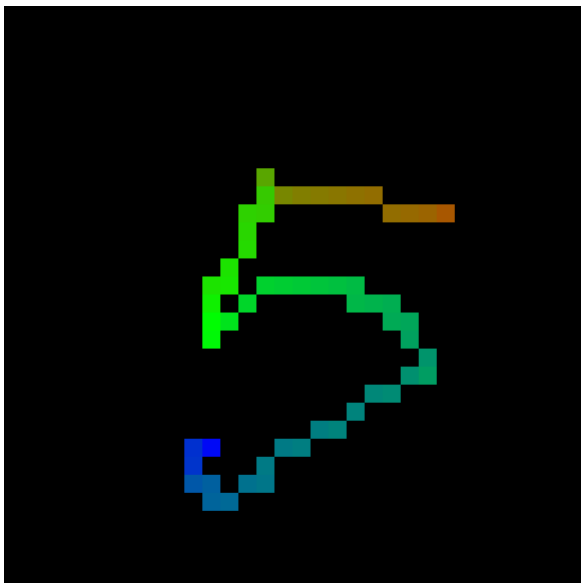


Figura 37: Rasterización de datos del sensor en aplicación de "Magic Wand"de Harvard Tiny ML.

La imagen resultante, mostrada en la Figura 37, contiene valores representativos de colores en los bits donde se encuentran datos de posición del gesto realizado con el dispositivo. De nuevo, la imagen resultante muestra resultados congruentes, aunque requiere de buena cantidad de operaciones. Esta imagen es utilizada para hacer inferencia en el modelo para el microcontrolador Arduino Nano 33 BLE Sense. El modelo que dicha librería plantea utiliza redes neuronales convolucionales para operar en la imagen y encontrar rasgos que brinden resultados distintivos según la imagen contenga uno u otro gesto.

Si bien el flujo de trabajo parece lógico y práctico, es necesario ver su funcionamiento al ejecutarlo en el microcontrolador. El modelo para implementación en el microcontrolador Arduino Nano 33 BLE Sense cuenta con un tamaño de 31256 bytes para reconocimiento de 10 gestos o patrones de movimiento. Al momento de la ejecución en el dispositivo, es notorio que la aplicación no cuenta con la sensibilidad deseada, pues no todos los gestos realizados son detectados por el dispositivo. Aún así, el dispositivo ejecuta el modelo sin errores, y a pesar de no ser el modelo más efectivo, realiza inferencias.

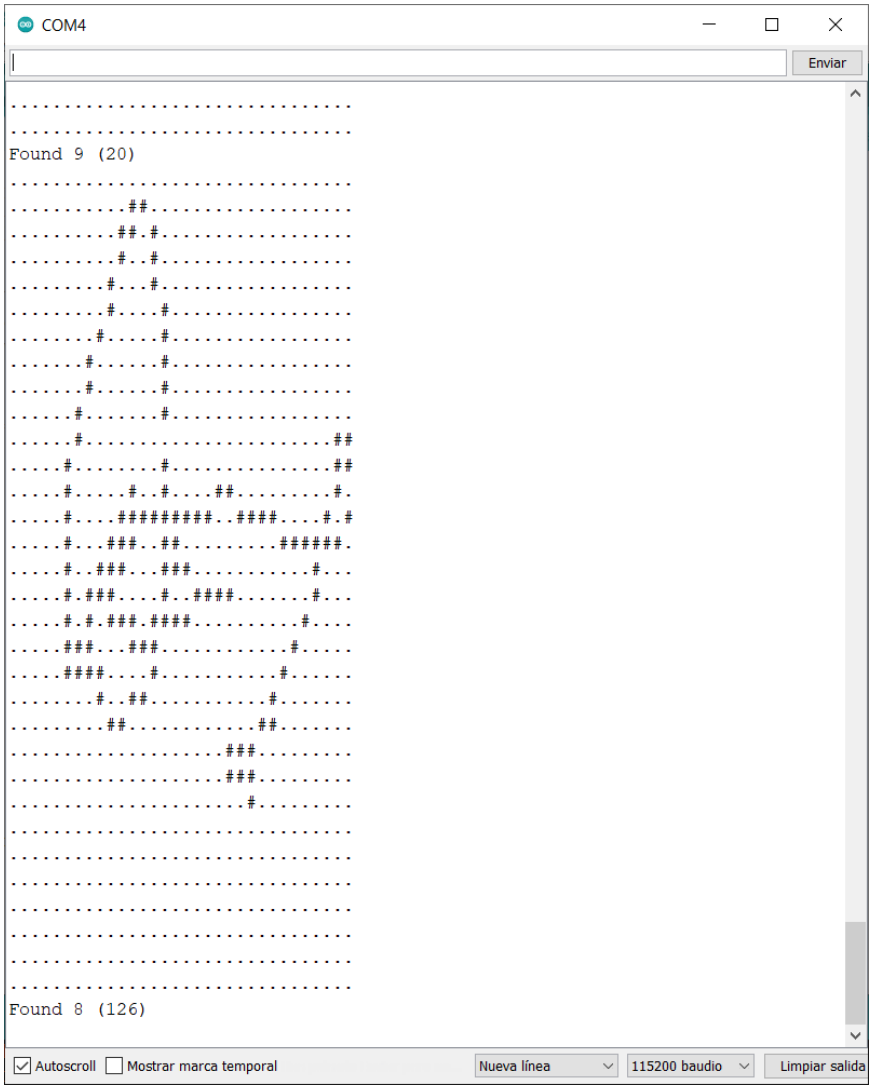


Figura 38: Lecturas e inferencias del modelo en aplicación de "Magic Wand"de Harvard Tiny ML, ejecutado en Arduino Nano 33 BLE Sense realizando el gesto representativo del dígito 2.

Al momento de realizar gestos con el dispositivo y observar los resultados y los datos con los que se infiere, es notorio que al dispositivo le toma un tiempo considerable realizar la inferencia, pues la respuesta no es inmediata. Esto se podría deber a la complejidad de las operaciones a que se utilizan en el modelo planteado y/o al procesamiento de datos que debe realizar el dispositivo previo a pasar a realizar inferencias. Aún así, la transformación de datos de aceleración a imagen con base en posiciones no es precisa, ya que al realizar gestos la imagen que interpreta el dispositivo no es cercana al movimiento real del dispositivo. Esto provoca que pocas veces se logren inferencias correctas. Obtener los resultados de imagen de gesto deseados luego del procesamiento de datos requeriría de mucha práctica de parte del usuario y requiere muchas operaciones para tan malos resultados, por lo que no se consideró un acercamiento adecuado a la aplicación que se busca realizar.

Luego, se analizó el ejemplo de "Magic Wand"desarrollado en la librería Arduino_-TensorFlowLite. En este caso, se implementa de nuevo un modelo que utiliza redes neuronales

convolucionales, aunque no se realiza rasterizado para obtener imágenes. La entrada del modelo utilizado en este ejemplo es una matriz con las lecturas del sensor, en este caso la IMU LSM9DS1 integrada en el Arduino Nano 33 BLE Sense. En este caso, el modelo no corrió en el primer intento. Luego de búsqueda exhaustiva, se encontraron blogs donde se planteaban modificaciones a la librería de control de la IMU, pues era ahí donde se daban los problemas de ejecución. Aún así, la documentación oficial de la librería y del sensor no plantean modificaciones oficialmente, quedando en evidencia la pobre documentación de esta tecnología. Aún luego de intentar dos modificaciones diferentes encontradas en la web, la aplicación no mostró una ejecución óptima. La aplicación muestra problemas de recolección de datos, pues no detecta los movimientos realizados por el usuario en todas las ocasiones, probablemente producto de las problemáticas de integración con las librerías del sensor.

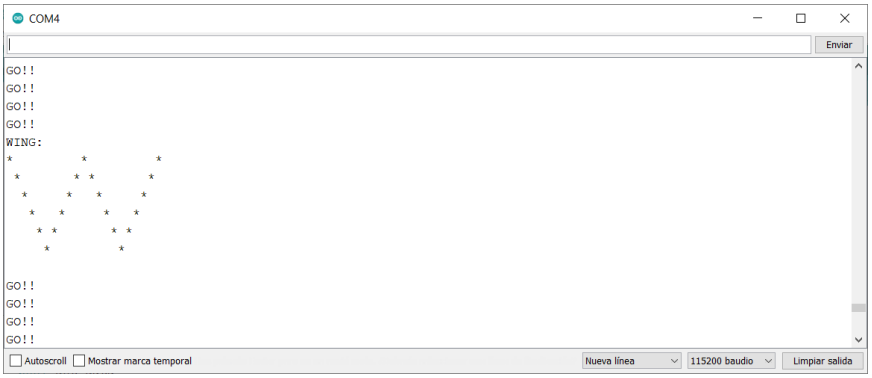


Figura 39: Lecturas e inferencias del modelo en aplicación de "Magic Wand"de Arduino TensorFlow Lite, ejecutado en Arduino Nano 33 BLE Sense.

Este segundo ejemplo contiene un modelo de 19616 bytes que reconoce tres gestos sencillos. Aunque un modelo más compacto que el primero, sigue utilizando operaciones complejas dentro del modelo, lo que promueven una ejecución lenta. Tomando en cuenta lo analizado, se decidió realizar un modelo que contase con operaciones sencillas y que contara con una aplicación robusta.

9.3. Recolección y división de datos

Esta sección presenta procedimientos para la selección de los patrones a reconocer, selección de variables a recolectar, captura serial de datos de Unidad de Medición Inercial (IMU) y división de los datos.

9.3.1. Selección de patrones a reconocer

La selección de patrones a reconocer consistió en definir movimientos que se desean reconocer con la aplicación a través de recolección de datos con una IMU con base en tres criterios: sencillos, rápidos de ejecutar y replicables. Seleccionó un gesto de golpe y otro de movimiento de muñeca, comunes en algunas aplicaciones de videojuegos.

Imágenes secuenciales del primer gesto (golpe):

Figura 40: Etapa inicial del primer gesto a reconocer (golpe).

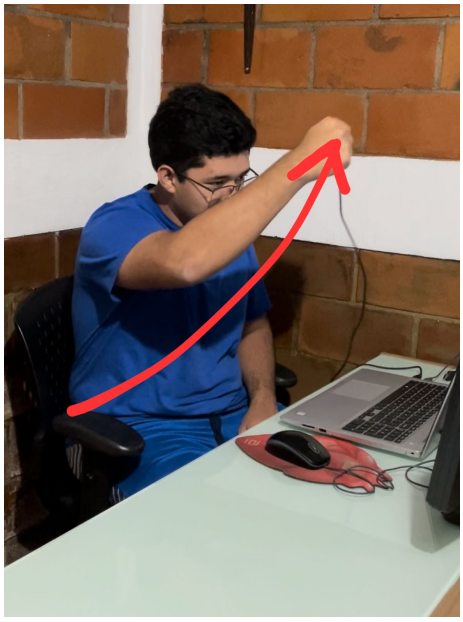


Figura 41: Etapa final del primer gesto a reconocer (golpe).

Imágenes secuenciales del segundo gesto (movimiento de muñeca):

Figura 42: Etapa inicial del segundo gesto a reconocer (movimiento de muñeca).

Figura 43: Etapa final del segundo gesto a reconocer (movimiento de muñeca).

9.3.2. Selección de variables recolectar

El dispositivo Arduino Nano 33 BLE Sense utilizado para esta aplicación cuenta con un Unidad de Medición Inercial (IMU) modelo LSM9DS2 que brinda datos de:

- Acelerómetro con salidas en el rango de $[-4, +4]$ G's ± 0.122 mG's.
- Giroscopio con salidas en el rango de $[-2000, +2000]$ grados/seg. ± 70 mgrados/seg.
- Magnetómetro con salidas en el rango de $[-400, +400]$ μT ± 0.014 μT [19].

La frecuencia de muestreo para el acelerómetro y el giroscopio está fijada inicialmente en 104 Hz, mientras que la del magnetómetro en 20 Hz, aunque pueden ser cambiadas mediante código.

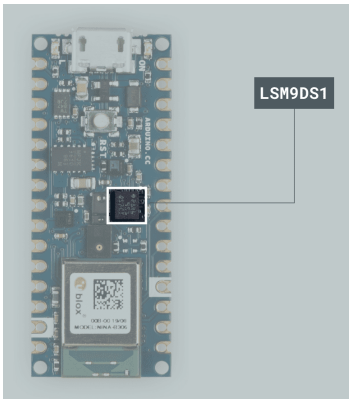


Figura 44: Unidad de medición incercial (IMU) integrada en el dispositivo Arduino Nano 33 BLE Sense [19]

.

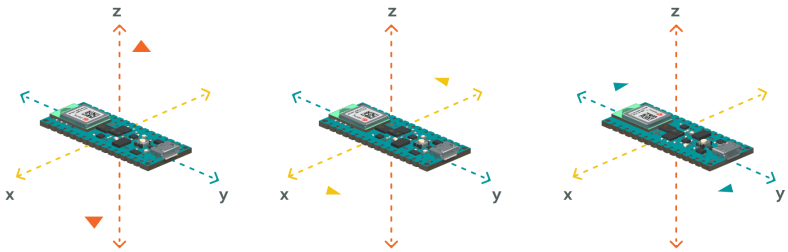


Figura 45: Orientación de ejes de unidad de medición incercial (IMU) integrada en el dispositivo Arduino Nano 33 BLE Sense [19].

De los datos disponibles en el dispositivo, se seleccionaron los obtenidos del acelerómetro y giroscopio, al considerarlos más relevantes para el muestre de movimientos en tres dimensiones con el dispositivos. El dispositivo brinda información de aceleración y giroscopio en tres ejes, por lo que el set de datos en cada muestra cuenta con seis componentes. Y se decidió tomar 119 muestras para cada gesto con una frecuencia de 119 Hz, con lo que se muestrea aproximadamente un segundo de tiempo. Además, con el fin de evitar una ejecución continua del modelo en momentos innecesarios, se estableció un threshold de aceleración para comenzar la toma de datos al detectar movimientos bruscos que superan dicho valor. Es importante mencionar que los datos fueron normalizados para el entrenamiento y durante la inferencia utilizando sus valores máximos y mínimos para obtener valores en el rango de -1 a 1.

9.3.3. Captura serial de lecturas de IMU

Las lecturas de la IMU fueron impresas en el Monitor Serial de Arduino IDE. Se realizó, mediante un código de Captura de gestos en Arduino, la recolección de datos al hacer lecturas del sensor al momento de ejecutar los gestos. Los datos fueron impresos de forma serial en formato de valores separados por comas. Luego, durante la ejecución, se utilizó el software Realterm para realizar una captura de los datos seriales y guardar los datos de cada gesto en un archivo *.csv diferente.

Todas las repeticiones de un mismo gesto fueron grabadas en el mismo archivo y en este caso no se realizó normalización en el dispositivo, sino posteriormente en el código de desarrollo del modelo. Los primeros tres datos tomados para cada lectura corresponden a la aceleración en los ejes X, Y y Z. Luego, se observan tres datos correspondientes al giroscopio en los tres ejes. La Figura 46 muestra una fracción de los datos capturados para un gesto. Se recolectaron 28 conjuntos de datos representativos del gesto de golpe y 32 representativos del gesto de movimiento de muñeca.

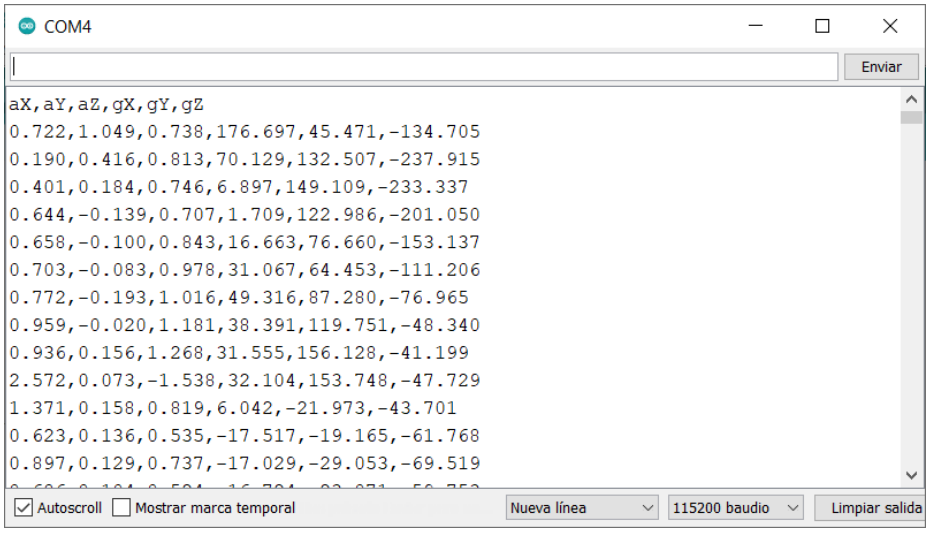


Figura 46: Captura de datos de gestos a reconocer mediante Monitor Serial de Arduino IDE.

Luego, mediante código en lenguaje Python en cuadernos Jupyter ejecutados utilizando Google Colaboratory, se procedió a graficar los datos obtenidos para cada uno de los gestos realizados mediante la recolección de datos.

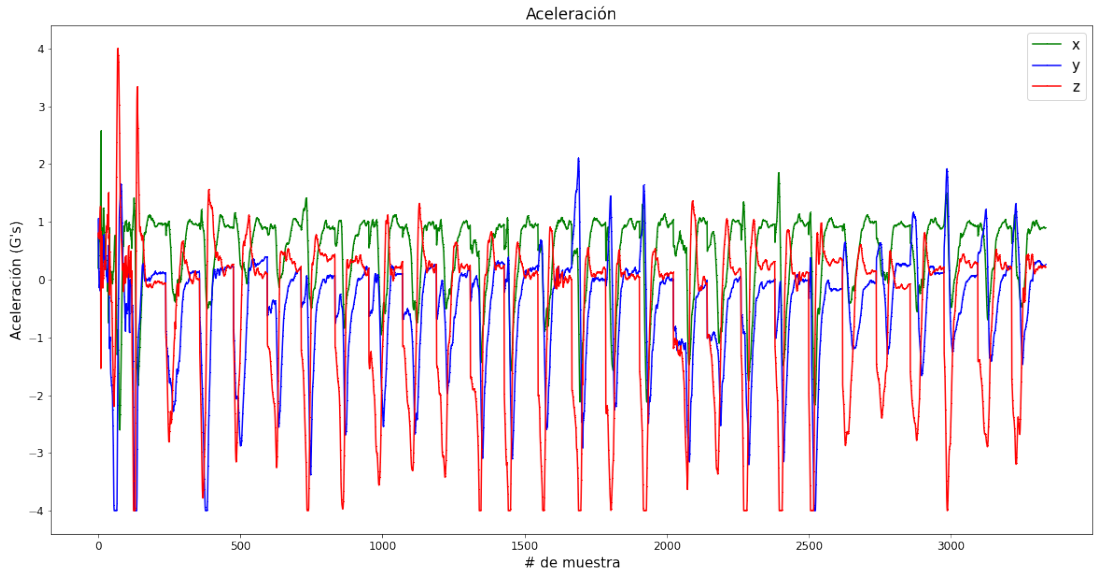


Figura 47: Datos de aceleración para capturas de gestos de golpe.

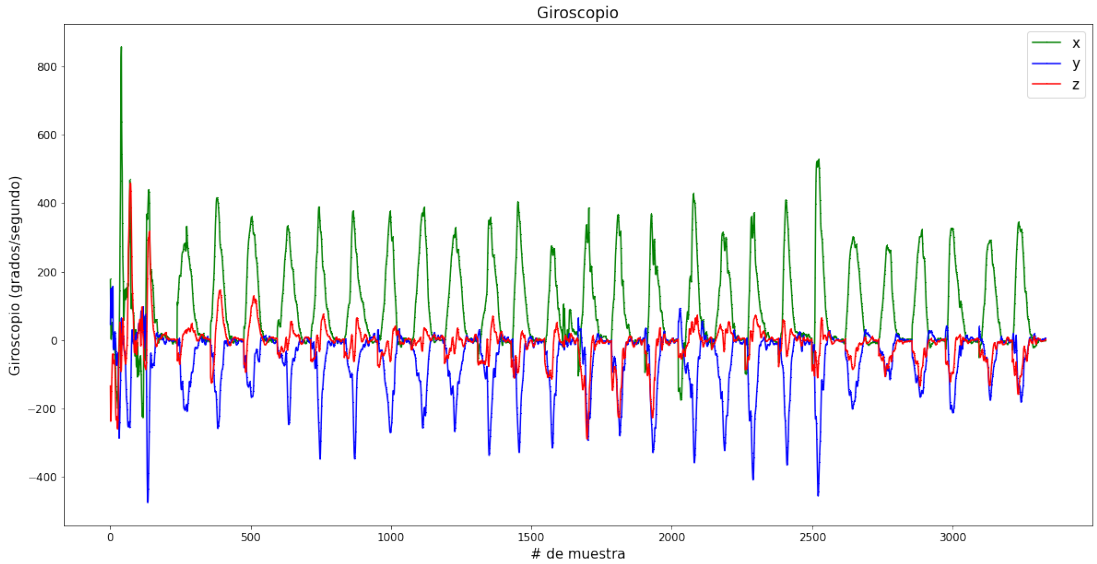


Figura 48: Datos de giroscopio para capturas de gestos de golpe.

Las Figuras 47 y 48, muestran con claridad todas las iteraciones recolectadas de los gestos de golpe en términos de aceleración y velocidad de giro en los tres ejes, respectivamente. En dichas figuras se pueden observar claros patrones repetitivos de los datos en las diferentes iteraciones que se realizaron para cada gesto. Aún así, los datos se encuentran en diferente escala, por lo que al realizar normalización previa al entrenamiento del modelo mostrarían comportamientos más uniformes y valores en un rango más cerrado.

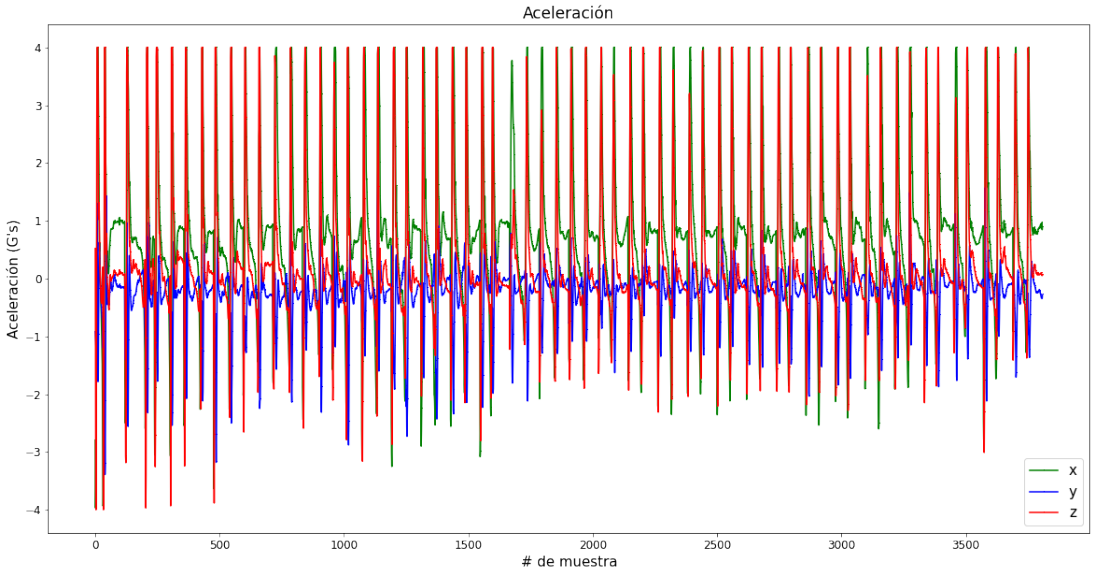


Figura 49: Datos de aceleración para capturas de gestos de muñeca.

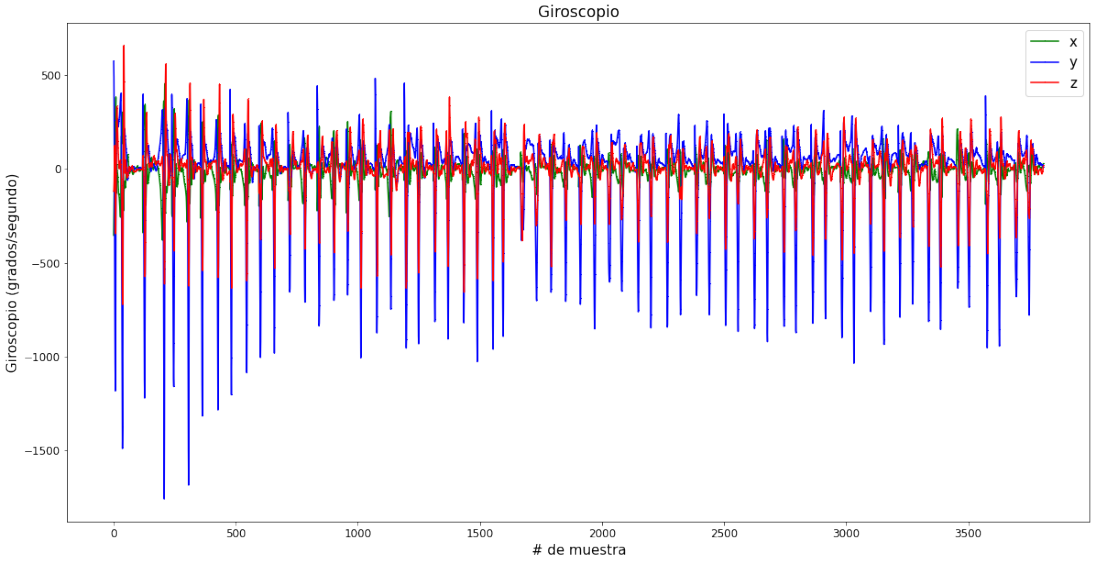


Figura 50: Datos de giroscopio para capturas de gestos de muñeca.

Las Figuras 49 y 50, muestran con claridad todas las iteraciones recolectadas de los gestos de movimiento de muñeca en términos de aceleración y velocidad de giro en los tres ejes, respectivamente. En dichas figuras también se pueden observar claros patrones repetitivos de una muestra del movimiento a otra, con lo que cobra sentido la recolección de datos y reconocimiento de dichos movimientos.

9.3.4. División de los datos

Dado que se buscaba entrenar una red neuronal para reconocimiento de patrones, se necesitó dividir los datos recolectados en tres conjuntos diferentes, uno de entrenamiento, otro de validación y un último de prueba. Los datos se dividieron de la siguiente forma, según recomendación de [8]:

- Entrenamiento: 60 %.
- Validación: 20 %.
- Prueba: 20 %.

9.4. Desarrollo del modelo de aprendizaje automático

En esta sección se aborda el diseño del modelo para reconocimiento de patrones, entrenamiento del modelo incluyendo parámetros y resultados del entrenamiento, así como resultados de inferencias del modelo en datos de prueba en una computadora. La ejecución de esta sección fue principalmente mediante Python en cuadernos Jupyter ejecutados utilizando Google Colaboratory, empleando como principal librería TensorFlow.

9.4.1. Características del sistema asignado en Google Colaboratory

Las etapas de procesamiento de datos, desarrollo del modelo de aprendizaje automático para reconocimiento de patrones y generación del modelo de TensorFlow Lite para reconocimiento de patrones fueron realizados utilizando el lenguaje Python en cuadernos ejecutados mediante Google Colaboratory. Todo el código se encuentra en un solo cuaderno, por lo que las características del sistema fueron las mismas en todos los procesos. Las características del sistema asignado por la plataforma para la ejecución del programa se listan a continuación.

- CPU: Intel Xeon @ 2.20GHz.
- GPU: No fue utilizada.
- RAM: 13.30 GB.
- Disco: 108.00 GB.

9.4.2. Diseño del modelo

El modelo seleccionado consiste únicamente en capas densas. Esto se debe a que este tipo de capas requieren de operaciones sencillas a comparación de las utilizadas en los ejemplos analizados, que incluyen capas convolucionales. Además, las funciones de activación de las primeras dos capas son también sencillas, aunque la de la capa de salida es softmax con la

finalidad de limitar las salidas y de interpretarlas como un porcentaje de confianza de que sea un gesto u otro.

No. de capa	Tipo de capa	Nodos	Función de activación	Parámetros entrenables
1	Densa	50	ReLU	35750
2	Densa	15	ReLU	765
3	Densa	2	Softmax	32

Cuadro 12: Hiper-parámetros de la arquitectura del modelo de reconocimiento de patrones.

En el Cuadro 12 se puede observar que la primera capa del modelo cuenta con 50 nodos, la segunda 15 nodos y la de salida con 2, con el fin de obtener una salida representativa de cada gesto que se desea reconocer. Es notorio que, en el apartado de parámetros entrenables, considerando la cantidad de entradas al modelo, se cuenta con una cantidad bastante grande de parámetros entrenables para el modelo. Esto significa que el modelo será de mayor tamaño que los presentados en los ejemplos, aunque no necesariamente más lento dadas las operaciones implementados. Aunque este aspecto sí podría afectar positivamente la capacidad de inferencia del modelo sobre los gestos a reconocer y el tiempo y complejidad del entrenamiento.

9.4.3. Entrenamiento del modelo

El entrenamiento fue realizado sin la utilización de aceleradores de hardware. A continuación se presentan sus parámetros y resultados.

Parámetros de entrenamiento

Modelo de reconocimiento de patrones	
Optimizador	RMSProp
Función de pérdida	Error cuadrático medio
Métrica	Error absoluto medio
Épocas	600
Tamaño de lote	1

Cuadro 13: Hiper-parámetros de entrenamiento del modelo de reconocimiento de patrones.

El Cuadro 13 muestra la selección de hiper-parámetros de entrenamiento para el modelo de reconocimiento de patrones. Las funciones de pérdida y métrica utilizadas fueron las mismas que para aplicaciones anteriores, aunque el optimizador cambió. En este caso se utilizó el optimizador conocido como RMSProp, que brinda un mejor desempeño que Adam al entrenar modelos complejos de forma más rápida y acertada. Se realizaron 600 épocas de entrenamiento, lo cual fue definido a prueba y error observando los resultados de entrenamiento con distintos parámetros. El tamaño de lote seleccionado fue 1 debido a la gran cantidad de datos y parámetros a entrenar. Aunque esta variación podría hacer más

lento el entrenamiento, asegura un mejor aprovechamiento de los datos para ajuste de los parámetros del modelo.

Resultados del entrenamiento

Luego de realizar el entrenamiento del modelo, se realizaron gráficas del historial de variaciones de la pérdida y métrica del modelo durante el entrenamiento. Los gráficos muestran valores para los conjuntos de datos de entrenamiento y validación a partir de la época 100 para una mejor visualización.

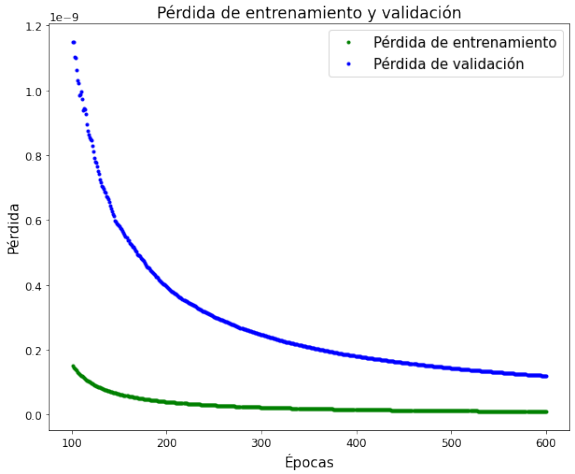


Figura 51: Evolución de pérdida del modelo durante el entrenamiento.

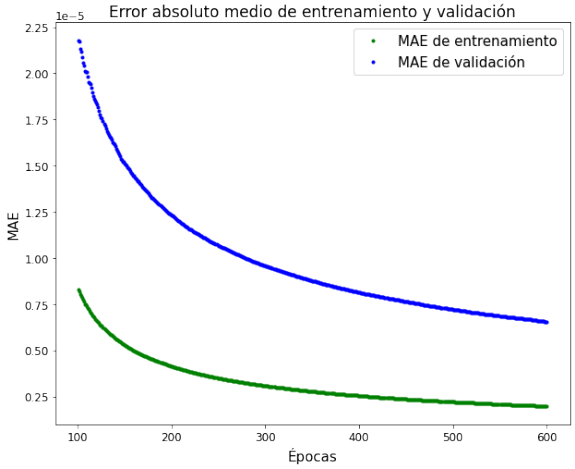


Figura 52: Evolución de métrica del modelo durante el entrenamiento.

Las Figuras 51 y 52 muestran valores descendentes para la pérdida y la métrica en los conjuntos de datos de entrenamiento y validación durante todas las épocas del entrenamiento. Esto sugiere que no existe sobre-ajuste, ya que no hubo estancamiento en los parámetros presentados ni pobres inferencias en la validación en comparación con el entrenamiento.

Además, considerando el orden de magnitud de los valores de pérdida y métrica, además de la complejidad del modelo, se puede sospechar que el modelo resultante es bastante bueno para realizar inferencias y reconocer los patrones para los que fue entrenado.

9.4.4. Inferencia en datos de prueba para evaluación del modelo

Luego de entrenar el modelo, se utilizaron los parámetros resultantes para realizar inferencias sobre el conjunto de datos de prueba. Es importante mencionar que el modelo brinda a la salida dos valores entre 0 y 1, que además suman 1. Estos valores son interpretados como la probabilidad de que el gesto de entrada al modelo sea uno u otro gesto. Tomando esto en cuenta, fue considerada como la predicción final del modelo el gesto para el que mayor valor de salida brindara la inferencia. El conjunto de datos de prueba cuenta con 12 datos, 7 de gesto de golpe y 5 de gesto de movimiento de muñeca.

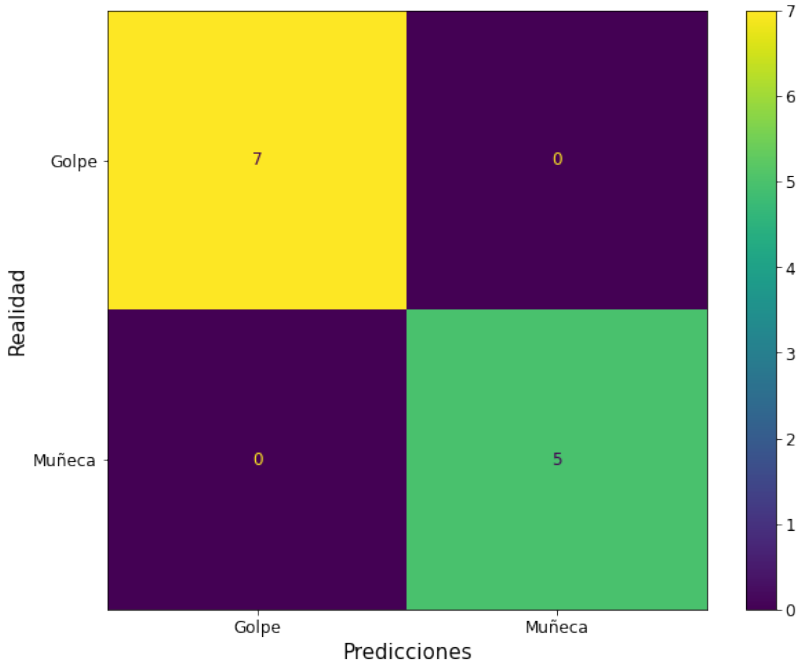


Figura 53: Matriz de confusión de inferencias del modelo en datos de prueba.

La Figura 53 muestra evidencia de la buena inferencia realizada por el modelo en los datos de prueba a través de una matriz de confusión. El modelo mostró 100 % de efectividad en las inferencias realizadas a los datos de prueba al no mostrar confusiones de gestos. Esto indicó que el modelo está listo para ser convertido e implementado en un microcontrolador.

Aún así, es necesario evaluar el comportamiento del modelo al realizar su conversión a un modelo de TensorFlow Lite e implementarlo en el dispositivo Arduino Nano 33 BLE Sense.

9.5. Generación de un modelo de TensorFlow Lite

En esta sección se aborda la conversión del modelo resultante a un formato interpretable por microcontroladores, mediante la utilización de Google Colaboratory y la librería TensorFlow.

9.5.1. Conversión del modelo

La conversión de este modelo a formato TensorFlow Lite no incluyó optimización. En este caso se utilizó el modelo en formato de punto flotante de 32 bits para los valores de entrada y salida. Esto se debe a que la cuantificación de un modelo de múltiples entradas es más complejo que de una sola entrada y no forma parte del alcance de este trabajo, por lo que se evitó. Aún así, es interesante notar las variaciones de tamaño que presenta la conversión del modelo a un formato interpretable por microcontroladores.

Modelo	Tamaño (bytes)
TensorFlow	84111
TensorFlow Lite	147856

Cuadro 14: Comparación de los modelos de reconocimiento de patrones obtenidos a partir del tamaño.

El Cuadro 14 muestra que el modelo utilizado para microcontroladores posee un tamaño mayor que el modelo original de TensorFlow. Es posible que esto se deba a las funciones de activación utilizadas, ya que en ocasiones es necesario guardar en el modelo información para aproximarlas en los microcontroladores, además de la gran cantidad de parámetros que componen el modelo que en el modelo original se encuentran en un formato comprimido diferente.

9.5.2. Modelo resultante como archivo fuente en C

A continuación, se muestra un extracto del modelo resultante como archivo fuente en C ejecutable en microcontroladores.

```
unsigned char g_model[] = {
  0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x14, 0x00, 0x20, 0x00,
  0x04, 0x00, 0x08, 0x00, 0x0c, 0x00, 0x10, 0x00, 0x14, 0x00, 0x00, 0x00,
  ...                                                                 ...
  0x30, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff,
  0xca, 0x02, 0x00, 0x00, 0xfc, 0xff, 0xff, 0xff, 0x04, 0x00, 0x04, 0x00,
  0x04, 0x00, 0x00, 0x00
};
unsigned int g_model_len = 147856;
```

9.6. Implementación del modelo en Arduino

Esta sección aborda la implementación del modelo de reconocimiento de patrones desarrollado en el dispositivo Arduino Nano 33 BLE Sense mediante código en Arduino IDE.

9.6.1. Creación de una aplicación para ejecución en microcontrolador

La creación de una aplicación para la ejecución del modelo requirió de varias acciones y segmentos de código que se listan y describen a continuación. Muchas de ellas son idénticas a las utilizadas en la aplicación mostrada anteriormente por lo que cuentan con una explicación más breve.

1. Instalación de bibliotecas:
   - *a*) Arduino_TensorflowLite.
   - *b*) Arduino_LSM9DS1: instalada mediante el administrador de bibliotecas para controlar el sensor.
2. Inclusión de librerías.
   - *a*) TensorFlowLite.h
   - *b*) tensorflow/lite/micro/all_ops_resolver.h
   - *c*) tensorflow/lite/micro/micro_error_reporter.h
   - *d*) tensorflow/lite/micro/micro_interpreter.h
   - *e*) tensorflow/lite/schema/schema_generated.h
   - *f*) tensorflow/lite/version.h
3. Definición de variables globales.
   - *a*) Definición de puntero de clase ErrorReporter.
   - *b*) Definición de puntero de clase Model.
   - *c*) Definición de puntero de clase MicroInterpreter.
   - *d*) Definición de puntero TfLiteTensor para las entradas y salidas del modelo.
   - *e*) Definición de valor de tamaño de la arena de tensores.
   - *f*) Definición de arena de tensores.
   - *g*) Definición de array de gestos.
   - *h*) Definición de número de gestos.
4. Función de *setup*.
   - *a*) Registro de configuración.
   - *b*) Obtener operaciones de implementación necesarias mediante AllOpsResolver.
   - *c*) Construir un intérprete para correr el modelo.
   - *d*) Asignar memoria de la arena de tensores para los tensores del modelo.

e) Obtener punteros a los tensores de entrada y salida del modelo.

f) Inicialización de puerto serial.

g) Inicialización de IMU.

5. Función de *loop*.

a) Obtención de variables de entrada al modelo mediante lectura de sensor.

b) Colocar la entrada en el tensor de entrada del modelo.

c) Inferir utilizando el modelo.

d) Reportar cualquier error.

e) Obtener la salida del modelo del tensor de salida.

f) Interpretación del valor de salida para definir resultado.

Es importante mencionar que las secciones del código mencionadas en el listado pueden tener modificaciones y añadidos dependiendo de las funciones que se desean realizar con la aplicación.

9.6.2. Obtención y manejo de entradas y salidas al modelo

Las entradas al modelo fueron obtenidas mediante lecturas de acelerómetro y giroscopio. Los seis datos obtenidos en cada lectura son adjuntados a un array unidimensional ordenados hasta completar la cantidad de datos necesario para cada gesto, en este caso 119. Los datos son normalizados según sus magnitudes máximas o mínimas antes de ser ingresados en el tensor de entrada del modelo. Luego, a la salida, se evalúa cuál de las dos componentes de la salida es mayor, para definir dicha inferencia como el resultado final. En el caso de la aplicación serial, los resultados son mostrados en el Monitor Serial de Arduino IDE como se muestran en el tensor de salida.

9.7. Pruebas de la aplicación

Luego de desarrollar la implementación del modelo, se procedió a ejecutar la aplicación para verificar su funcionamiento. Los resultados se muestran en las imágenes siguientes.

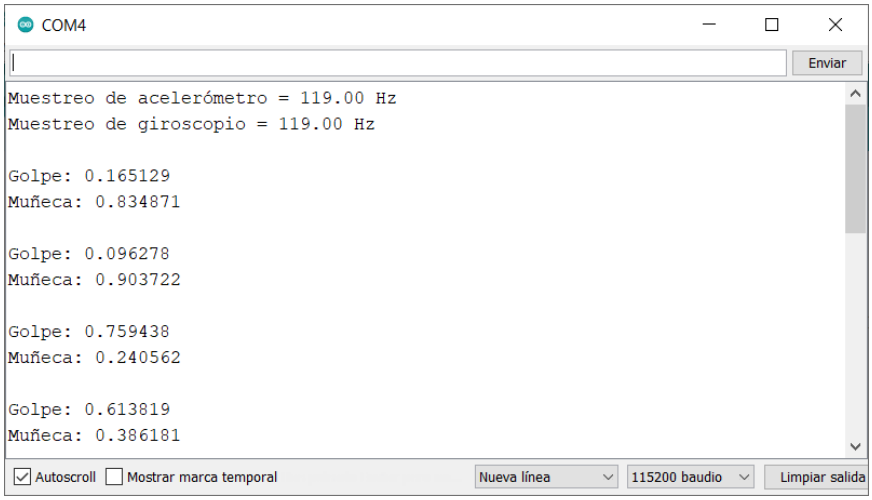


Figura 54: Captura de Monitor Serial con Inferencias del modelo y frecuencia de muestreo de sensores.

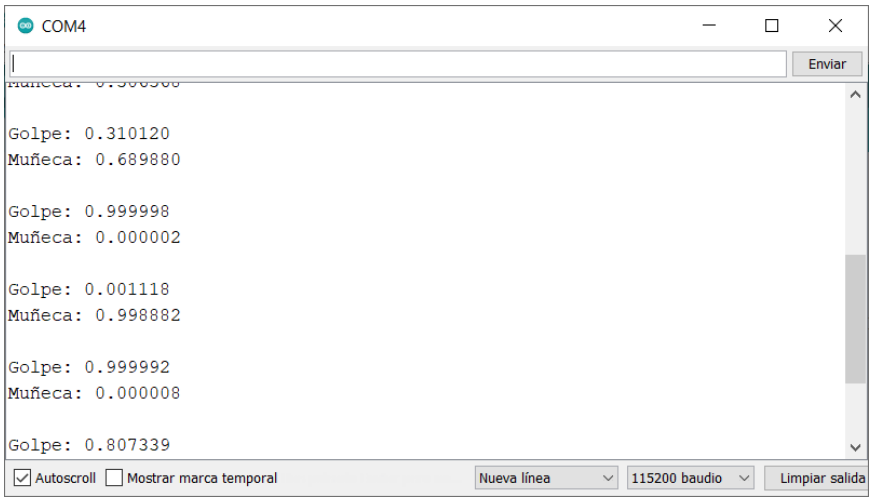


Figura 55: Captura de Monitor Serial con Inferencias del modelo.

Las Figuras 54 y 55 muestran evidencia de las inferencias realizadas por el modelo y las salidas impresas en el Monitor Serial de Arduino IDE. Durante esta prueba fue satisfactorio notar que el modelo no realiza ejecuciones en todo momento o con movimientos muy leves, sino solo cuando el usuario realiza los gestos intencionalmente. Además, la inferencia sugiere un tiempo de respuesta rápido, aunque ese parámetro no se evaluó cuantitativamente. Esto podría atribuirse a la sencilla recolección de datos que hace el dispositivo y a las capas más sencillas en la arquitectura del modelo que reducen las operaciones necesarias para su ejecución.

Es importante mencionar que los gestos para ser reconocidos por el dispositivo deben ser imitaciones de los gestos recolectados para entrenamiento, ya que variaciones de ritmo u otros podrían hacer que el modelo no reconozca el gesto como lo que en realidad es. Además, es importante mantener la misma posición y orientación de sujeción para el dispositivo durante

las inferencias que para los datos de entrenamiento, ya que las mediciones variarían de eje y provocaría que el modelo no realizase las inferencias correctas. Esta problemática sugiere que sería útil fabricar un sujetador o estuche que facilite el agarre del dispositivo y mantener su posición y orientación.

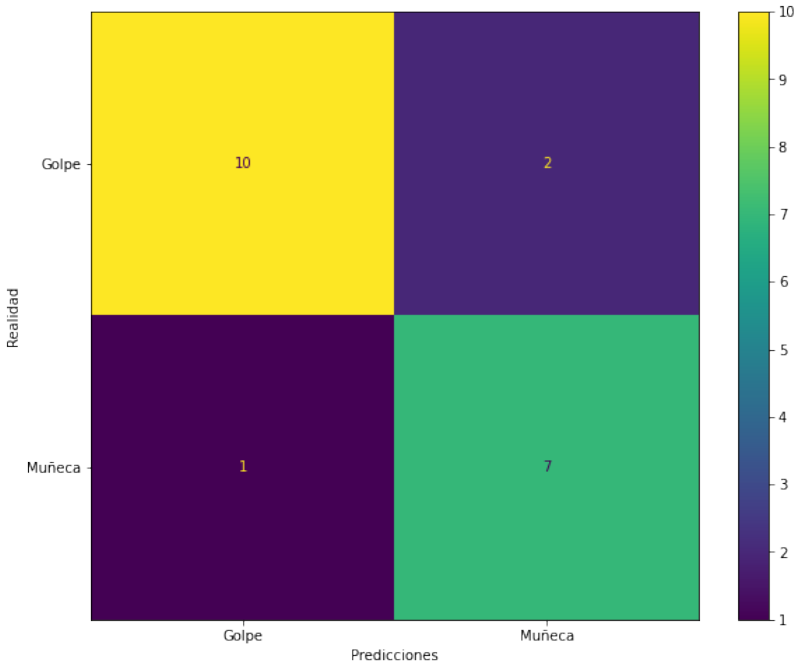


Figura 56: Matriz de confusión de inferencias del modelo en ejecución real de aplicación.

La Figura 56 muestra resultados de las inferencias realizadas por el modelo al ejecutarlo en el microcontrolador seleccionado. El modelo no mostró un 100 % de efectividad en las inferencias realizas, aunque mostró buena cantidad de respuestas correctas. El modelo confundió dos gestos de golpe con gestos de movimiento de muñeca y un único gesto de movimiento de muñeca con golpes. Aún así, esto podría atribuirse a la complejidad e incomodidad de sujetar el dispositivo sin alguna clase de estuche y/o a las variaciones de ritmo de los gestos respecto de los datos utilizados para el entrenamiento, ya que en general la aplicación mostró una buena respuesta.

CAPÍTULO 10

Visión por computadora utilizando aprendizaje automático a través de módulo de visión y microcontrolador: Arduino, JeVois A33 y YOLO

10.1. Diseño experimental

Este experimentó consistió en realizar una aplicación de visión por computadora a través de la ejecución de una red neuronal que implementa un algoritmo conocido como YOLO. Se buscó que módulo de visión JeVois A33 ejecutase el algoritmo YOLO en imágenes de vídeo en tiempo real, obteniendo resultados de detección y clasificación de objetos en la imagen que capta el dispositivo. Los resultados obtenidos serán enviados de forma serial hacia un dispositivo Arduino Micro que utilizará la información recibida para ejecutar un controlador de servomotores para centrar el objeto detectado en el plano de imagen de la cámara. Es importante mencionar que se desarrollaron diferentes controladores y se probaron con diferentes formas de visión por computadora a manera de obtener comparaciones.

El flujo de trabajo empleado se describe a continuación:

1. Selección de objeto a detectar y clasificar mediante YOLO.
2. Recolección de datos.
    a) Datos de entrenamiento.
    b) Datos de prueba.
3. Desarrollo de algoritmo YOLO.
    a) Algoritmo YOLO v3.
    b) Algoritmo Tiny YOLO v3.
    c) Comparación de algoritmos y resultados.

4. Implementación del algoritmo YOLO en módulo de visión JeVois A 33.

5. Diseño y fabricación de brazo para giro e inclinación de módulo de visión.

6. Comunicación entre dispositivos.

7. Desarrollo de controladores para brazo robótico.

    a) Controlador PD usando ArUco Markers.

        1) Desarrollo del controlador
        2) Parámetros seleccionados
        3) Pruebas del controlador

    b) Controlador Image Based Visual Servoing (IBVS) usando ArUco Markers.

        1) Desarrollo del controlador
        2) Parámetros seleccionados
        3) Pruebas del controlador

    c) Controlador Image Based Visual Servoing (IBVS) usando algoritmo YOLO

        1) Desarrollo del controlador
        2) Parámetros seleccionados
        3) Pruebas del controlador

10.2. Selección de objeto a detectar y clasificar mediante YOLO

La selección del objeto a detectar en imágenes y clasificar mediante la utilización del algoritmo YOLO tomó en cuenta varios aspectos:

- Objeto de geometría sencilla, aunque con variación de imagen según la perspectiva.
- Objeto común para mayor facilidad de acceso a fotografías.
- Objeto en posesión del desarrollador para poder realizar pruebas.

Tomando en cuenta los aspectos listados anteriormente, se decidió utilizar una calculadora CASIO fx-991LAX CLASSWIZ. Este objeto se compone de un cuerpo con geometría básica de prisma rectangular, que brinda diferentes imágenes según la perspectiva desde la que se observa por los botones y otros accesorios que lo componen. Además, varios compañeros de grado cuentan con el mismo modelo, lo que facilita el acceso a fotografías para entrenamiento y el modelo físico para pruebas.

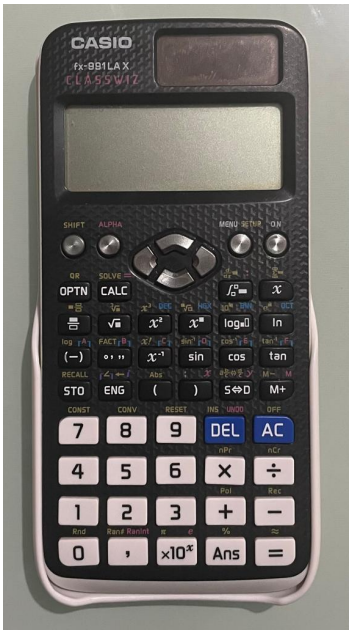


Figura 57: Calculadora CASIO fx-991LAX CLASSWIZ.

10.3. Recolección de datos

Esta sección aborda la recolección de datos para esta aplicación, que consistió en obtener una cantidad grande de imágenes del objeto seleccionado para detectar. Luego, en el caso de los datos de entrenamiento fue necesario procesarlos al realizar un etiquetado de los objetos que se busca que el algoritmo reconozca.

10.3.1. Datos de entrenamiento

Los datos de entrenamiento se obtuvieron de dos fuentes principales. La primera fue mediante descarga automatizada de Google, utilizando [20] como guía, y la segunda fue recolección de imágenes de calculadoras del modelo seleccionado propiedad de compañeros de grado. A ellos les fue solicitada su cooperación para hacer la recolección de los datos necesarios. La primera fuente brindó 80 imágenes al conjunto de datos de entrenamiento y la segunda brindó 90 imágenes. En total se utilizaron 170 imágenes para el entrenamiento.

Luego, se verificó que todas las imágenes obtenidas se encontrasen en el mismo formato, en este caso *.JPG. Las imágenes que no se encontraban al momento en ese formato, se convirtieron utilizando la aplicación en línea 11zon. Lo siguiente fue hacer un etiquetado de las imágenes de entrenamiento, para lo que se usó la aplicación LabelImg. En ella, se hicieron recuadros en cada una de las imágenes, seleccionando el objeto que se desea detectar y clasificar. En dicha aplicación se selecciona también la categoría en la que se desea clasificar cada uno de los objetos seleccionados en las imágenes.

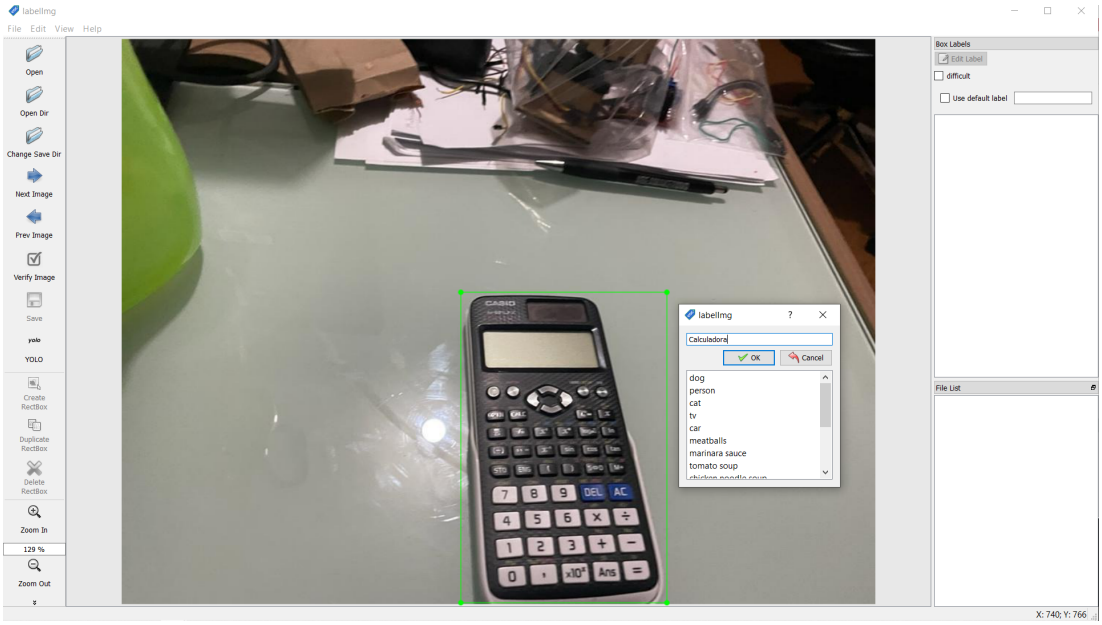


Figura 58: Etiquetado de objetos en imágenes.

En cada imagen fue necesario crear un recuadro que encierre cada uno de los objetos que se desean detectar y clasificar, para luego presionar guardar y generar automáticamente un archivo de texto que contiene información en el formato que se muestra a continuación.

```
C     X         Y         W         H
0 0.141667 0.491071 0.210000 0.803571
```

Los datos obtenidos representan información de la categoría del objeto, su ubicación en la imagen y el tamaño del recuadro que lo encierra. El formato YOLO indica que se obtienen las variables C X Y W H, donde C es un entero representativo de la categoría o nombre del objeto a encerrado en el recuadro, X es la coordenada x normalizada del centro del recuadro que encierra el objeto, de forma similar Y, W representa el ancho del recuadro que encierra el objeto y H representa la altura, también en valores normalizados.

Luego, los archivos de imagen de entrenamiento junto con su correspondiente documento de texto con las coordenadas, fueron comprimidos en una carpeta y cargados en Google Drive, para poder accesarlos de manera fácil a través de cuadernos de Google Colaboratory.

10.3.2. Datos de prueba

Se obtuvieron imágenes de prueba en un período de tiempo posterior al de recolección de datos de entrenamiento. Se solicitó a compañeros su colaboración para juntar diferentes fotografías y vídeos donde se pudiera observar la fotografía en cuestión. A continuación se muestra una imagen de prueba. Además, uno de los vídeos de prueba puede ser observado en este vínculo.

Figura 59: Imagen de prueba recolectada.

10.4. Desarrollo de algoritmos YOLO

El desarrollo de los algoritmos YOLO se realizó con la guía de [21], incluyendo la preparación del set de entrenamiento, el entrenamiento y la ejecución del modelo en imágenes de prueba. En esta sección se aborda el desarrollo de cada uno de los algoritmos utilizados. En esa aplicación se utilizaron las versiones número 3, puesto que son las últimas versiones desarrolladas completamente y probadas. Existe una versión 4, aunque aún sigue en desarrollo.

10.4.1. Algoritmo YOLO v3

El algoritmo YOLO v3 fue desarrollado mediante código en lenguaje Python ejecutado mediante cuadernos Jupyter en Google Colaboratory. Las características del sistema asignado fueron las siguientes:

- CPU: Intel Xeon @ 2.30GHz.
- GPU: Tesla K80 de 12 GB.
- RAM: 13.30 GB.
- Disco: 79.00 GB.

El desarrollo del algoritmo YOLO consistió en el entrenamiento de una versión predesarrollada y original del algoritmo a través del framework de Darknet [22]. Este algoritmo

base es entrenado para un set de datos personalizado de forma que realice las inferencias necesarias. El flujo de trabajo utilizado se presenta a continuación.

1. Montaje de Google Drive en máquina virtual.
2. Clonación de repositorio de Darknet.
3. Compilación de Darknet utilizando OpenCV y GPU Nvidia.
4. Configuración de la red Darknet para entrenamiento del modelo YOLO v3.
    a) Obtención de archivo .cfg.
    b) Cambios en archivo .cfg.
    c) Configuración de respaldo de archivos de pesos.
    d) Creación de archivos .names y .data.
    e) Descarga de pesos del modelo 53 de Darknet.
5. Extracción de imágenes
    a) Descompresión de archivos de imágenes y texto para entrenamiento.
    b) Parsing de archivos de texto para obtención de coordenadas y archivos.
    c) Obtención de listado de imágenes.
    d) Creación de archivo train.txt.
6. Entrenamiento del modelo.
    a) Copia de archivos relevantes del modelo en Google Drive.
    b) Entrenamiento de algoritmo YOLO v3 con datos personalizados utilizando los archivos desarrollados.

El archivo .cfg utilizado fue el yolov3.cfg, ya contenido en la carpeta /cfg/ del repositorio clonado. Esta configuración contiene la arquitectura del modelo, así como algunos parámetros de entrenamiento. En este archivo fue estrictamente necesario modificar la cantidad de clases a detectar con el modelo. Dicho parámetro fue reemplazado de 80 clases a 1 clase, dado que solo se buscaba reconocer un objeto. Luego, considerando que solo se detectaría una clase, hubo que modificar la cantidad de filtros en las capas denominadas YOLO. La cantidad de filtros en dicha capa pasó de 255 a 18, dado que el desarrollador recomienda utilizar $3 \times (n + 5)$, donde n representa la cantidad de clases a detectar. En este caso el modelo contiene 3 capas de este tipo y los cambios fueron realizados en el archivos .cfg mediante el comando sed -i para edición de archivos de texto.

Luego, otra parte importante del flujo de trabajo fue la creación de los archivos obj.names y obj.data. El archivo obj.names contiene los nombres de las categorías a detectar con el modelo. En este caso únicamente se incluyo la palabra calculadora, puesto que es el único objeto a detectar. Por otro lado, el archivo obj.data almacena la cantidad de clases a detectar por el modelo y la ubicación de los archivos que contienen datos de entrenamiento, ubicación del archivo obj.names y ubicación de la carpeta para respaldo, que en este caso fue en una carpeta de Google Drive.

El archivo de pesos pre-entrenados utilizados para el modelo es llamado darknet53.conv.74, con un peso de 154.96 MB. El entrenamiento duró un total de 1000 épocas y duró más de 4 horas. Otros resultados importantes serán discutidos más adelante. Además, fue necesaria una exploración de [23] como guía para interpretar las salidas de entrenamiento del modelo. El entrenamiento fue detenida cuando la pérdida dejó de mostrar mejorías con el avance de las épocas.

10.4.2. Algoritmo Tiny YOLO v3

El algoritmo Tiny YOLO v3 también fue desarrollado mediante código en lenguaje Python ejecutado mediante cuadernos Jupyter en Google Colaboratory. Las características del sistema asignado en este caso fueron las mismas que en el algoritmo YOLO v3 y son mostradas a continuación.

- CPU: Intel Xeon @ 2.30GHz.
- GPU: Tesla K80 de 12 GB.
- RAM: 13.30 GB.
- Disco: 79.00 GB.

El desarrollo del algoritmo Tiny YOLO también consistió en el entrenamiento de una versión pre-desarrollada y original del algoritmo a través del framework de Darknet, tomando como guía el artículo mostrado en [24]. Este algoritmo base es entrenado para el mismo set de datos personalizados que para el algoritmo YOLO v3 de forma que realice las inferencias necesarias. El flujo de trabajo utilizado tiene la misma base que para el algoritmo YOLO v3, aunque con leves modificaciones que son abordadas a continuación.

El archivo .cfg utilizado fue el yolov3-tiny.cfg, ya contenido en la carpeta /cfg/ del repositorio clonado. Esta configuración contiene la arquitectura del modelo tiny, así como algunos parámetros de entrenamiento. Esta arquitectura contiene menor cantidad de capas, lo que provoca una disminución en la precisión, aunque con una disminución positiva en el espacio en memoria que ocupan los pesos y en la velocidad de inferencia del modelo. En este archivo fue estrictamente necesario modificar la cantidad de clases a detectar con el modelo de 80 a 1, igual que con el algoritmo anterior. Luego, considerando que solo se detectaría una clase, hubo que modificar la cantidad de filtros en las capas denominadas YOLO. La cantidad de filtros en dicha capa también pasó de 255 a 18, siguiendo las recomendaciones del desarrolador. En este caso el modelo contiene únicamente 2 capas de este tipo.

El archivo de pesos pre-entrenados utilizados para el modelo es llamado yolov3-tiny.weights, con un peso de 33.79 MB. El entrenamiento duró un total de 2054 épocas y duró alrededor de 1 hora. Otros resultados importantes serán discutidos más adelante. El entrenamiento fue detenida cuando la pérdida dejó de mostrar mejorías con el avance de las épocas.

10.4.3. Comparación de algoritmos y resultados

Luego de realizar el entrenamiento, se procedió a comparar los resultados. Los parámetros y resultados de comparación se muestran en el Cuadro 15. A partir del entrenamiento y algunas inferencias, se compararon los parámetros listados a continuación:

- Duración del entrenamiento,
- Épocas de entrenamiento,
- Pérdida mínima alcanzada durante el entrenamiento,
- Tamaño del archivo de pesos del modelo resultante del entrenamiento,
- Velocidad de inferencia en vídeo, y
- Confianza utilizada en la inferencia.

Parámetro	**YOLO v3**	**Tiny YOLO v3**
Duración (s)	14319	3451
Épocas	1000	2054
Pérdida mínima	0.209666	0.291194
Tamaño de pesos (MB)	234.9	33.1
Velocidad de inferencia (FPS)	14.1 - 14.4	62.9 - 79.1
Confianza	0.5	0.5

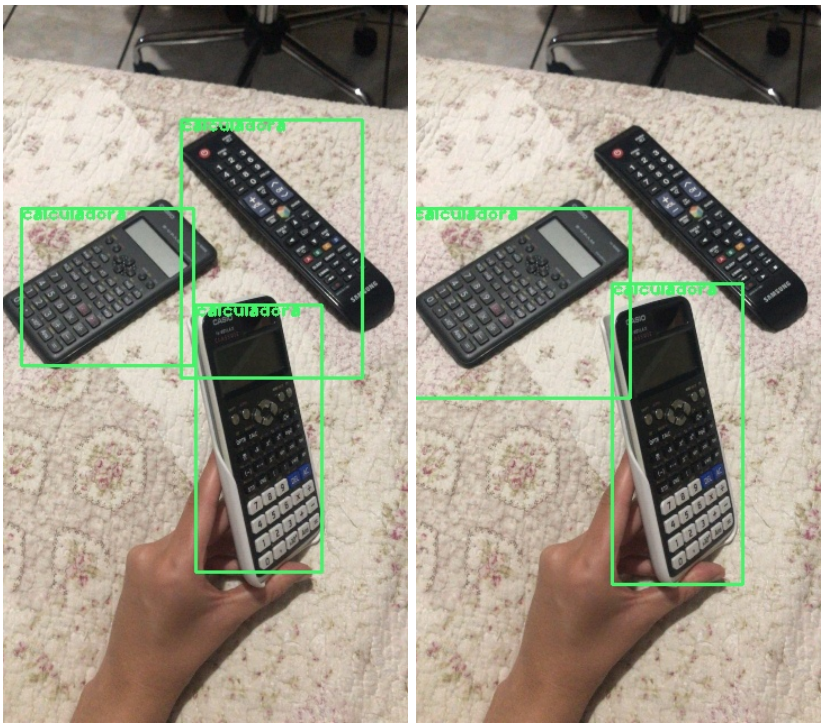
Cuadro 15: Comparación de algoritmos YOLO v3 y Tiny YOLO v3.

En los resultados observados en el cuadro anterior, es notorio que la duración del entrenamiento del algoritmo Tiny YOLO v3 es considerablemente menor, lo que es positivo y se debe a su menor complejidad en cuanto a la arquitectura utilizada. Aún así, el algoritmo YOLO v3 requirió una menor cantidad de épocas que el algoritmo Tiny YOLO v3 para llegar a una pérdida mínima similar de entre 0.20 y 0.29 unidades. Un parámetro importante a considerar, dado que se busca implementar el modelo en un microcontrolador que cuenta con memoria limitada es el tamaño del archivo de pesos. Los pesos resultantes del algoritmo Tiny YOLO v3 muestran un mejor resultado, ya que tienen un peso menor, por una diferencia de más de 200 MB.

Luego, al momento de realizar inferencias en vídeos, que es la aplicación que se busca, el algoritmo Tiny YOLO v3 mostró un mejor rendimiento en términos de velocidad al cuadruplicar la velocidad de inferencia del algoritmo YOLO v3. Es importante mencionar que al momento de realizar inferencias se utiliza un parámetro de confianza. Es un parámetro entre 0 y 1 que actúa como threshold para tomar una inferencia del modelo como válida. El modelo brinda salidas entre 0 y 1 como probabilidad de que sea uno de los objetos que se detectan con el modelo. El parámetro de confianza utilizado para todos los modelos desarrollados fue de 0.5, significando que las inferencias del modelo menores a 0.5 no son representadas en las imágenes como resultados. Esto se hace ya que al considerar todas las inferencias como válidas, se pueden generar recuadros o detecciones no deseadas. Se utilizó

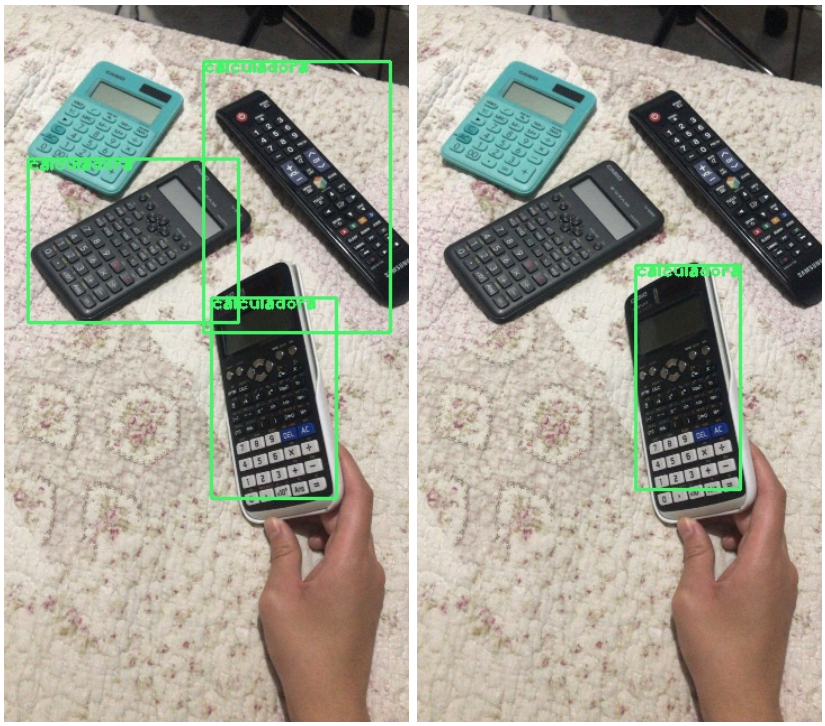
el mismo parámetro de confianza en todas las inferencias para facilitar una comparación más objetiva y se determinó el parámetro mediante prueba y error al observar las inferencias del modelo con diferentes valores.

Las inferencias en imágenes mostraron algunas resultados interesantes mostrados a continuación. El resto de resultados puede ser observado en anexos. Las inferencias en imágenes se realizaron mediante la utilización de la librería OpenCV en lenguaje Python. En caso distinto, las inferencias sobre vídeos pre-grabados se realizaron utilizando Darknet, donde se utilizó tarjeta gráfica y se aprovechan comandos en C y CUDA. Una muestra de inferencia en vídeo para el algoritmo YOLO v3 puede ser observado accediendo a este vínculo. Los resultados de inferencia del algoritmo Tiny YOLO v3 puede observarse en este vínculo.



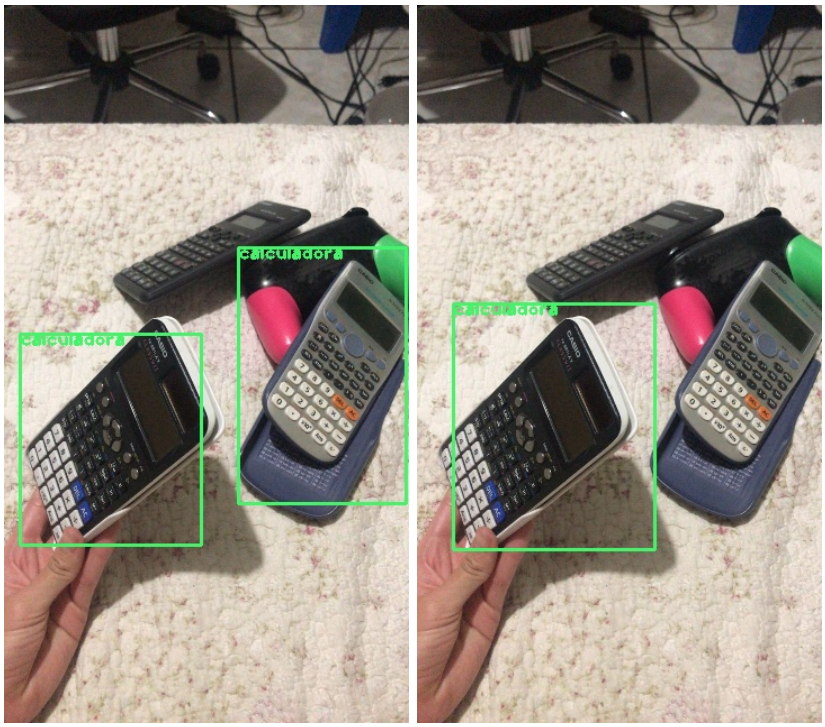
(a) YOLO v3. (b) Tiny YOLO v3.

Figura 60: Inferencia de algoritmos YOLO desarrollados sobre imagen de prueba #14.



(a) YOLO v3. (b) Tiny YOLO v3.

Figura 61: Inferencia de algoritmos YOLO desarrollados sobre imagen de prueba #15.



(a) YOLO v3. (b) Tiny YOLO v3.

Figura 62: Inferencia de algoritmos YOLO desarrollados sobre imagen de prueba #16.

Las Figuras 60, 61 y 62 son compuestas por las imágenes que mostraron los resultados más interesantes al hacer inferencias con los modelos desarrollados. En las comparaciones mostradas, se puede observar que el modelo YOLO v3 es más generalizable, ya que detecta algunas calculadores de diferente tipo. También se puede observar que ambos modelos presentan inferencias con diferentes tamaños de recuadros en las ocasiones en las que detectan en mismo objeto. Aún así, las inferencias obtenidas con el algoritmo Tiny YOLO v3, son suficientemente buenas ya que detectan la calculadora que se desea en la mayoría de las imágenes de prueba. Además, considerando la reducción de tamaño del archivo de pesos del modelo Tiny YOLO v3, las diferencias en las inferencias son asumibles ya que brinda ventajas en la implementación en dispositivos de memoria limitada. Por esta razón, se seleccionó el modelo Tiny YOLO v3 desarrollado como el algoritmo a implementar en el módulo de visión JeVois A33.

10.5. Implementación de algoritmo YOLO en módulo de visión JeVois A33

Se buscó realizar implementación del algoritmo YOLO seleccionado en el módulo de visión JeVois A33. Inicialmente, se utilizó este dispositivo ya que ofrecía en su comercialización la implementación de modelos propios de visión por computadora utilizando aprendizaje automático. Aún así, el dispositivo mostró complicaciones. El dispositivo se maneja utilizando JeVois Inventor, un software que permite la selección del módulo que ejecuta el dispositivo en la toma de la cámara para obtener la imagen de salida. La primera complicación encontrada al momento de la implementación del modelo YOLO fue que el dispositivo no permite la edición, compilación y/o creación de código en C++ para la ejecución de modelos.

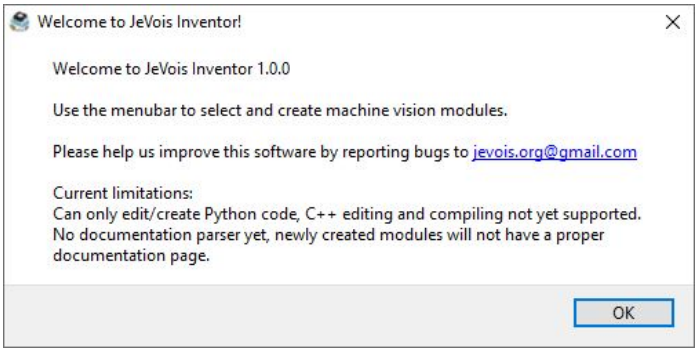


Figura 63: Aviso de limitaciones de edición y creación en JeVois A33.

Al momento de la inicialización del módulo se muestra el aviso de la Figura 63. Este aviso provocó que la ejecución del modelo tuviera que implementarse en Python, generando una inferencia demasiado lento por la naturaleza del lenguaje y de la librería OpenCV utilizada en dicho lenguaje. Aún así, al ejecutar el modelo en el módulo de visión, el dispositivo muestra complicaciones al realizar inferencias, ya que son escasas las ocasiones en que se muestran recuadros de detección en la toma cuando se enfoca el objeto y el módulo colapsa luego de un tiempo de ejecución. Sin embargo, la implementación se consideró correcta ya que el dispositivo cuenta con un módulo en el que se implementa YOLO utilizando Python

y muestra el mismo comportamiento que en la implementación personalizada, aunque no muestra el comportamiento necesario para integrar en el sistema de control. Los problemas en la implementación se explican con las limitaciones del dispositivo para ejecución y desarrollo de nuevos módulos.

El desarrollo de la aplicación se continuó utilizando un módulo de ejemplo en el que se implementa usando C++ un algoritmo YOLO que detecta múltiples objetos.

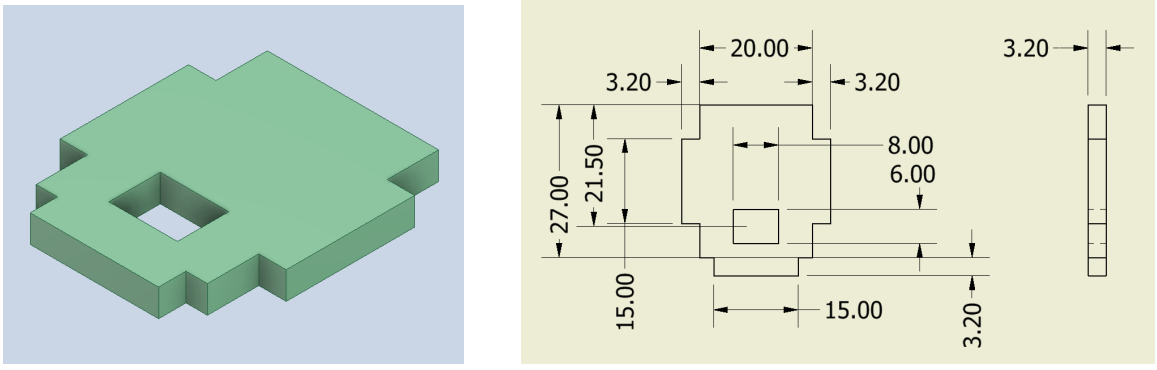
Es importante mencionar que a partir de la sección a continuación, el trabajo fue realizado en conjunto con Héctor Kleé para desarrollar el brazo robótico, los sistemas de control y su implementación.

10.6. Diseño y fabricación de brazo para giro e inclinación de módulo de visión

Se emprendió el diseño de un brazo para giro e inclinación de un módulo de visión JeVois A33 tomando en cuenta los aspectos siguientes:

1. El brazo necesita girar sobre dos ejes, uno vertical y uno horizontal.
2. El brazo es motorizado con dispositivos Hitec HS-8360TH.
3. El brazo debía sostener la cámara en un extremo para realizar movimientos de giro e inclinación.
4. Se contó con planchas de MDF de 3.2 mm de espesor.
5. Se debía fabricar con corte láser preferiblemente.

El cumplimiento de dichos requerimientos se dio mediante el modelado 3D de piezas de espesor uniforme en Inventor. Se diseñaron estructuras más complejas a partir de piezas planas sencillas que facilitaron la fabricación. Las piezas tomaron en cuenta mediciones de los motores y del módulo de visión.



(a) Modelo 3D.

(b) Dimensiones.

Figura 64: Dimensiones de pieza modelada en Inventor para estructura de brazo para giro e inclinación de módulo de visión.

Las piezas diseñadas se evaluaron aún en la etapa de diseño mediante ensamblaje en Inventor para verificar su funcionalidad y tamaños adecuados. Los modelos 3D representativos de los motores fueron obtenidos de una librería pública de modelos 3D conocida como GrabCad y solo se verificó que las mediciones del modelo coincidieran con las reales [25].

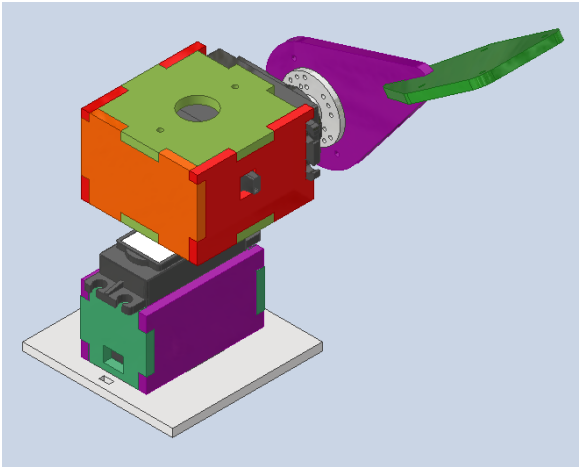
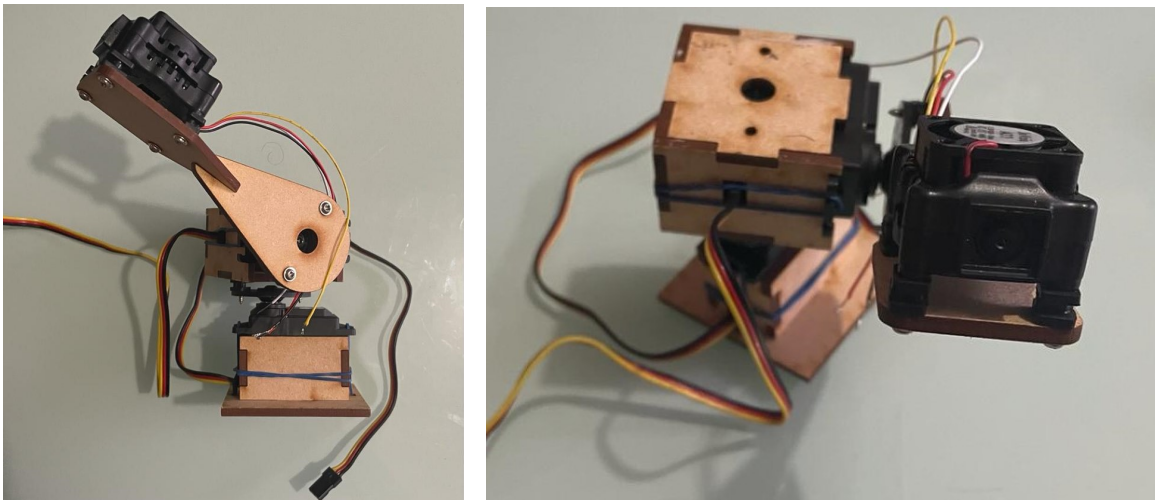


Figura 65: Ensamblaje de piezas modeladas en 3D en Inventor.

Luego de corroborar digitalmente el diseño, se procedió a la realización de un plano para corte láser. Las líneas de bordes de las piezas fueron especialmente parametrizadas en color y espesor para que la máquina las reconociera como líneas de corte.

Después de la fabricación mediante corte láser, se procedió al ensamblaje de las piezas. Durante el ensamblaje se utilizaron los accesorios del kit de piezas del motor, que incluye tornillos, separadores y adaptadores de transmisión de movimiento. Además, se usaron ambos motores y el módulo de visión. El resultado se muestra en las siguientes figuras.



(a) Vista lateral. (b) Vista frontal.

Figura 66: Visualización del brazo robótico ensamblado.

10.7. Comunicación entre dispositivos

El sistema de visión por computadora ejecutado por el módulo de visión JeVois A33 envía de forma serial los datos resultantes de la inferencia en un formato estandarizado. Dada la naturaleza del modelo a ejecutar, se utilizó el formato *Terse*, un formato de mensaje de resultados que muestra coordenadas para ubicación en dos dimensiones en el plano de Figura [26]. Las datos recibidos en un Arduino Micro son procesados mediante un *parser* que extrae los datos necesarios del array de texto recibido. El parser utilizado fue extraído de un ejemplo desarrollado por JeVois [27]. Este es implementado como una máquina de estados finitos para extraer la información útil de las coordenadas del centro del objeto detectado en el plano de imagen. Dichas coordenadas son mencionadas más adelante, haciendo referencia a ellas como $\overline{u}$ y $\overline{v}$, donde u es el eje horizontal en el plano de imagen y v el eje vertical.

La transmisión de información se realiza mediante el puerto serial de los dispositivos en cuestión. La velocidad de comunicación fue fijada a un *baudrate* de 115200.

10.8. Desarrollo de controladores para brazo robótico

Esta sección muestra el desarrollo de los sistemas de control a implementar en el brazo robótico con retroalimentación de las coordenadas de objetos en el plano de imagen obtenido mediante el módulo de visión JeVois A33. Es importante mencionar que 2 de los 3 controladores presentados fueron desarrollados para visión por computadora mediante algoritmos clásicos para detectar ArUco Markers, principalmente trabajado por Hector Kleé en su trabajo de graduación. Estos controladores fueron desarrollados para mostrarlos como evidencia de funcionamiento de los sistemas de control previo a la implementación utilizando YOLO como sistema de visión por computadora. Las coordenadas deseadas para el objeto en el plano de imagen son $(u, v) = (0, 0)$, puesto que se desea centrar el objeto en el plano de imagen.

10.8.1. Controlador PD usando ArUco Markers

Se desarrolló un Controlador PD para centrar en el plano de imagen el centroide de un ArUco Marker detectado por un algoritmo clásico de visión por computadora. Este controlador fue desarrollado como línea base de lo que se espera de un sistema de control, ya que se tienen expectativas superiores para el otro algoritmo de control desarrollado.

Desarrollo del controlador

Se implementó un sistema de control independiente para cada motor del brazo robótico. El controlador para q_1 brinda salidas para controlar el motor que gira sobre el eje vertical para ajustar la coordenada horizontal del objeto en el plano de imagen. El controlador para q_2 brinda salidas para controlar el motor que gira sobre el eje horizontal para ajustar la coordenada Vertical del objeto en el plano de imagen. Las ecuaciones utilizadas se repiten para cada controlador, aunque utilizaron diferentes constantes.

Ecuaciones utilizadas para controlador de motor 1.

$$e_u = u_{ref} - \overline{u} \tag{24}$$
$$eD_u = e_u - e_{u-ant} \tag{25}$$
$$q_1 = q_{1-ant} + (K_{P1} \cdot e_u + K_{D1} \cdot eD_u) \tag{26}$$

Ecuaciones utilizadas para controlador de motor 2.

$$e_v = v_{ref} - \overline{v} \tag{27}$$
$$eD_v = e_v - e_{v-ant} \tag{28}$$
$$q_2 = q_{2-ant} + (K_{P2} \cdot e_v + K_{D2} \cdot eD_v) \tag{29}$$

En las ecuaciones presentadas, e representa el error de la coordenada correspondiente respecto de la coordenada deseada, eD representa el cambio del error respecto del error anterior y q representa la nueva salida a ingresar en el motor luego de aplicar el sistema de control.

Parámetros seleccionados

Los parámetros seleccionados para el controlador se muestran en el Cuadro 16.

Constante	**Motor 1** (q_1)	**Motor 2** (q_2)
K_P	0.006	0.007
K_D	0.005	0.009

Cuadro 16: Constantes de controlador PD.

Pruebas del controlador

La demostración del funcionamiento de este controlador puede ser observado en la primera parte del vídeo incluido este vínculo. Se muestra una imagen del momento de pruebas. El controlador mostró resultados positivos al centrar el objeto en el plano de imagen en la mayoría de situaciones evaluadas. Aún así, existen momentos en los que el controlador muestra movimientos oscilantes al estabilizar el objeto en un punto. Esto se debe a que los controladores son independientes y tratan de compensar el comportamiento del contrario con movimientos, ya que no consideran el efecto del controlador complementario en su modelo.

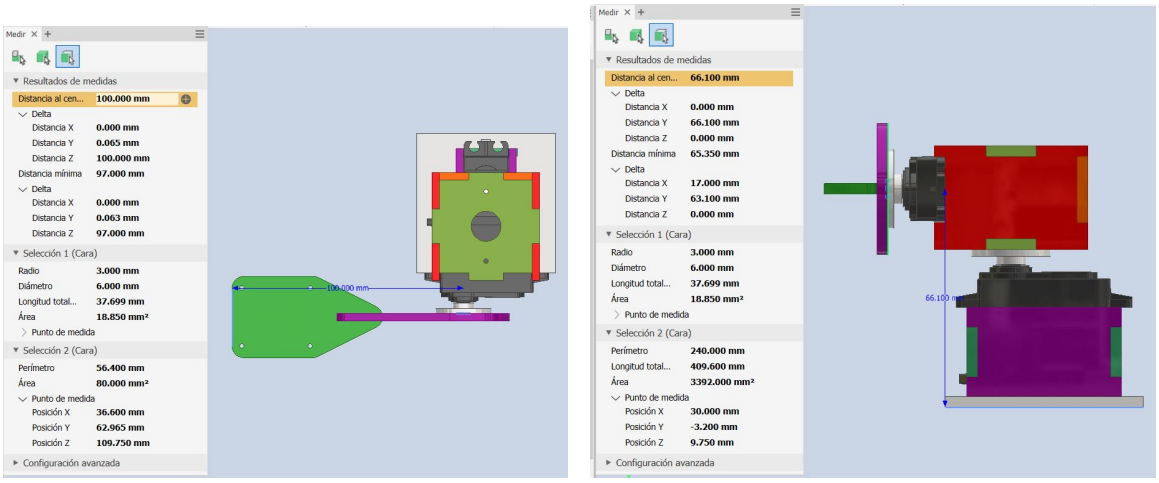
Figura 67: Pruebas de controlador PD usando ArUco Markers.

10.8.2. Controlador Image Based Visual Servoing (IBVS) usando ArUco Markers

Se desarrolló un Controlador Image Based Visual Servoing (IBVS) para centrar en el plano de imagen el centroide de un ArUco Marker detectado por un algoritmo clásico de visión por computadora. Este controlador fue desarrollado como demostración del funcionamiento del sistema de control desarrollado, previo a implementarlo con el sistema de visión por computadora utilizando YOLO.

Desarrollo del controlador

Este controlador considera la cinemática del brazo robótico que controla. Por esta razón, fue necesario tomar medidas de los eslabones que componen el sistema robótico. El brazo diseñado fue modelado como un robot de dos eslabones con dos juntas revolutas. Considerando esto, fue necesario tomar medidas de los eslabones. El primer eslabón fue compuesto desde la base del robot hasta la junta horizontal, mientras que el segundo eslabón fue considerado desde la junta horizontal hacia el extremo donde se ubica el módulo de visión instalado. Se muestran imágenes de las mediciones, donde se hace referencia a ellas como parámetros de la matriz de Denavit - Hartenberg.



(a) Medida a1. (b) Medida d1.

Figura 68: Medidas de eslabones del brazo robótico.

Considerando el ensamblaje físico del robot y la instalación de sus motores, fue necesario considerar matrices de transformación de base TB y transformación de efector final TEF para considerarlas en la cinemática directa. La matriz de transformación de base es pre-multiplicada a la cinemática directa obtenida de la matriz de parámetros de Denavit - Hartenberg DH, que es luego post-multiplicada por la matriz de transformación de efector final. Las matrices obtenidas se muestran a continuación.

$$TB = trotx(180) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{30}$$

$$DH = \begin{bmatrix} q_1 & -0.0661 & 0 & \frac{\pi}{2} \\ q_2 - \frac{\pi}{2} & 0 & -0.1 & 0 \end{bmatrix} \tag{31}$$

$$TEF = troty(-90) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{32}$$

Dichas matrices permitieron obtener la cinemática directa del manipulador fabricado. El brazo robótico fue simulador mediante la utilización del Robotics Toolbox, mostrado en [28], obteniendo resultados positivos al replicar con el modelo digital los movimientos del modelo físico.

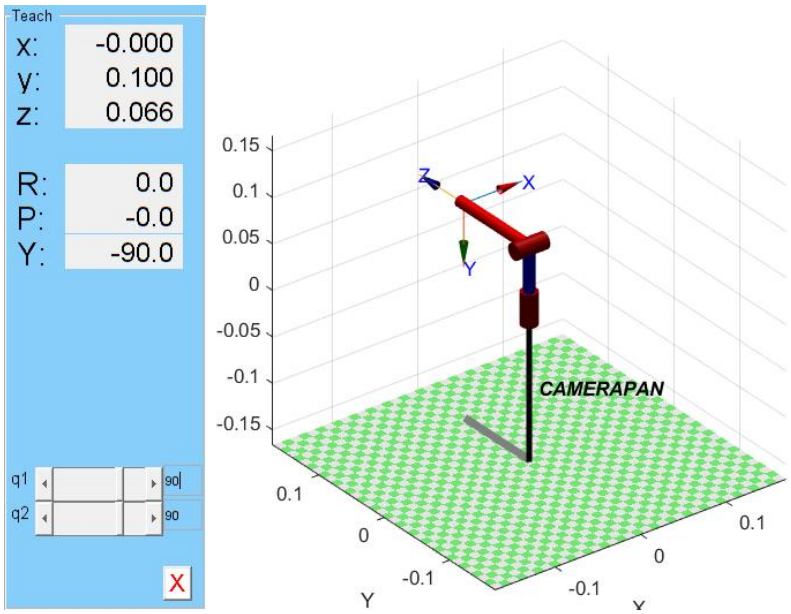


Figura 69: Imagen de simulación de cinemática directa del modelo del brazo robótico.

Luego, fue desarrollado el modelo del controlador IBVS, donde las principales ecuaciones son las siguientes.

$$\dot{e}_s = -J_L J_L^{\dagger} K e_s \tag{33}$$

$$q_{k+1} = q_k + J_L^{\dagger}[k] K e_s[k] \tag{34}$$

Esta teoría muestra que la derivada del error $\dot{e}_s$ utiliza el jacobiano de imagen J_L y su inversa J_L, de forma que se cancela la dinámica del sistema, obteniendo una matriz positiva definida como resultado. Al utilizar esto, la derivada del error resultante es básicamente una matriz de constantes K multiplicada por el error e_s, donde K debe ser una matriz positiva definida. Es importante mencionar que el algoritmo teórico utiliza la pseudo inversa, mediante el algoritmo de Moore-Penrose en el caso de MATLAB, para aproximar la inversa del jacobiano de imagen. Esto provocó resultados negativos, al no lograr implementar este algoritmo en el que se ejecuta el controlador y utilizar la inversa del jacobiano de imagen mediante el método de la adjunta. En este caso el algoritmo mostró comportamientos singulares y sobre-actuación de los motores, de forma que no se alcanzó el objetivo de control.

Considerando esto, se realizaron modificaciones al algoritmo mediante pruebas experimentales. En el caso funcional experimental se utilizaron las ecuaciones mostradas a continuación.

$$\dot{e}_s = -J_L K J_L^{\dagger} e_s \tag{35}$$

$$q_{k+1} = q_k + K J_L^{\dagger}[k] e_s[k] \tag{36}$$

Las ecuaciones anteriores muestran que la dinámica del sistema ya no se cancela mediante el jacobiano de imagen y su inversa, debida a que se alteró el orden de la multiplicación. Aún así, la matriz K utilizada es positiva definida. Además, para el cálculo de la inversa del Jacobiano de imagen, se utilizó el algoritmo de Levenberg-Marquardt. Estas alteraciones provocaron que no se tenga garantía matemática de tener un sistema L.T.I. G.A.S. asintóticamente estable, como se tiene con la propuesta teórica. Aún así, observando los resultados experimentales, se sospecha de tener un sistema asintóticamente estable con las modificaciones del algoritmo. Para probar la estabilidad del sistema se debería utilizar funciones de Lyapunov y el principio de invarianza de LaSalle, aunque dicha prueba no se realizó ya que escapa del alcance de este trabajo de graduación.

Parámetros seleccionados

Este sistema de control requirió de la estimación de dos parámetros, uno de ellos fue la profundidad del punto de interés en el plano de imagen c_z, que se decidió mantener constante debido a que su estimación es demasiado compleja matemáticamente para el microcontrolador utilizado y no toma demasiada relevancia en esta aplicación. Otro parámetro estimado fue la matriz de constantes de control K, que fue estimada mediante prueba y error haciendo variaciones hasta encontrar una que mostrase el comportamiento adecuado. A continuación se muestran los parámetros seleccionados.

$$c_z = 10 \tag{37}$$

$$K = \begin{bmatrix} 0.1 & 0 \\ 0 & 1 \times 10^{10} \end{bmatrix} \tag{38}$$

Pruebas del controlador

La demostración del funcionamiento de este controlador puede ser observado en la segunda parte del vídeo incluido este vínculo. Se muestra una imagen del momento de pruebas. Este controlador mostró un comportamiento estable y cumplió el objetivo de control. Muestra una velocidad de respuesta adecuada y demuestra el funcionamiento del algoritmo de control desarrollado. Además, brinda una solución más robusta que el controlador PD, ya que se toma en cuenta la dinámica del sistema en el controlador y ambos motores se controlan a partir del mismo sistema. Es importante mencionar que en este caso, el algoritmo de visión por computadora brinda información con una frecuencia bastante elevada al sistema de control, de forma que este muestra un comportamiento bastante suave en sus transiciones para seguir y centrar el objeto en el plano de imagen.

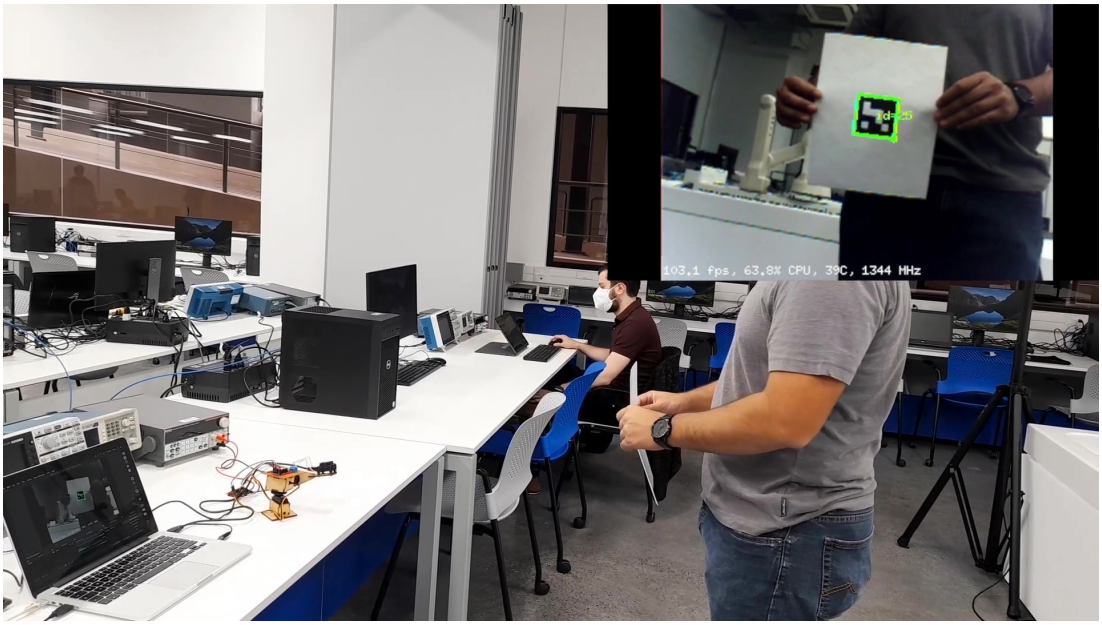


Figura 70: Pruebas de controlador IBVS usando ArUco Markers.

10.8.3. Controlador Image Based Visual Servoing (IBVS) usando algoritmo YOLO

Se desarrolló un Controlador Image Based Visual Servoing (IBVS) para centrar en el plano de imagen el centroide de una persona detectada por un algoritmo de visión por computadora moderno que utiliza aprendizaje automático (YOLO). Este controlador fue desarrollado para comprobar la interacción del sistema de control IBVS con las inferencias del algoritmo YOLO. El desarrollo de este sistema de control fue el mismo que para el controlador anterior, al igual que los parámetros, ya que lo único que cambió en estos ámbitos fue la forma de obtención de datos de entrada al sistema.

En este caso de control, fue necesario implementar el sistema de control IBVS de forma distinta. Las salidas del algoritmo de visión de computadora YOLO ejecutado por el módulo de visión JeVois A33 son demasiado lentas, haciendo inferencias a una velocidad de entre 2 y 3 cuadros por segundo. Esto llevó a utilizar el algoritmo de control IBVS de forma iterativa. Por cada dato de entrada del sistema de visión de computadora, se realizaron 3 iteraciones del algoritmo IBVS del sistema de control, buscando hacer converger al sistema a una configuración que colocase el objeto detectado en el centro del plano de imagen.

Pruebas del controlador

La demostración del funcionamiento de este controlador puede ser observado en el vídeo incluido este vínculo. Se muestra una imagen del momento de pruebas. Este caso de aplicación del algoritmo IBVS de control mostró un comportamiento adecuado, aunque no óptimo. Aún así, el controlador mostró un comportamiento suficientemente bueno en el que realizaba movimientos adecuados para centrar el objeto detectado, aunque no logró centrar

en ningún momento el objeto en el plano de imagen. Tomando en cuenta esto, se consideran resultados positivos, ya que el controlador mostró un comportamiento que puede ser útil en algunas aplicaciones, como detección de personas y ajuste de cámaras en espacios pequeños para observar mejor su comportamiento.

Entre las principales problemáticas que afectaron el comportamiento del sistema se encuentra que la velocidad de inferencia del sistema de visión por computadora es demasiado lento en el modelo de visión, debido a que no es una implementación nativa, sino a través de la librería OpenCV en C++. Habría sido de gran utilidad un módulo de visión en el que se pueda implementar de forma nativa este tipo de algoritmos, como el OpenMV Cam. Aún así, el controlador mostró versatilidad al lograr implementarlo realizando varias iteraciones a partir de un único dato de entrada, situación que no funcionaría con un Controlador PD, por la naturaleza del sistema. Además, la variación de la velocidad de inferencia del sistema sería difícil de modelar o considerar en un sistema de control tradicional como el controlador PD, mientras que en IBVS se puede combatir realizando iteraciones para converger a una configuración de los motores que logren el objetivo, por lo que el algoritmo IBVS muestra mayor versatilidad y los resultados son considerados positivos.

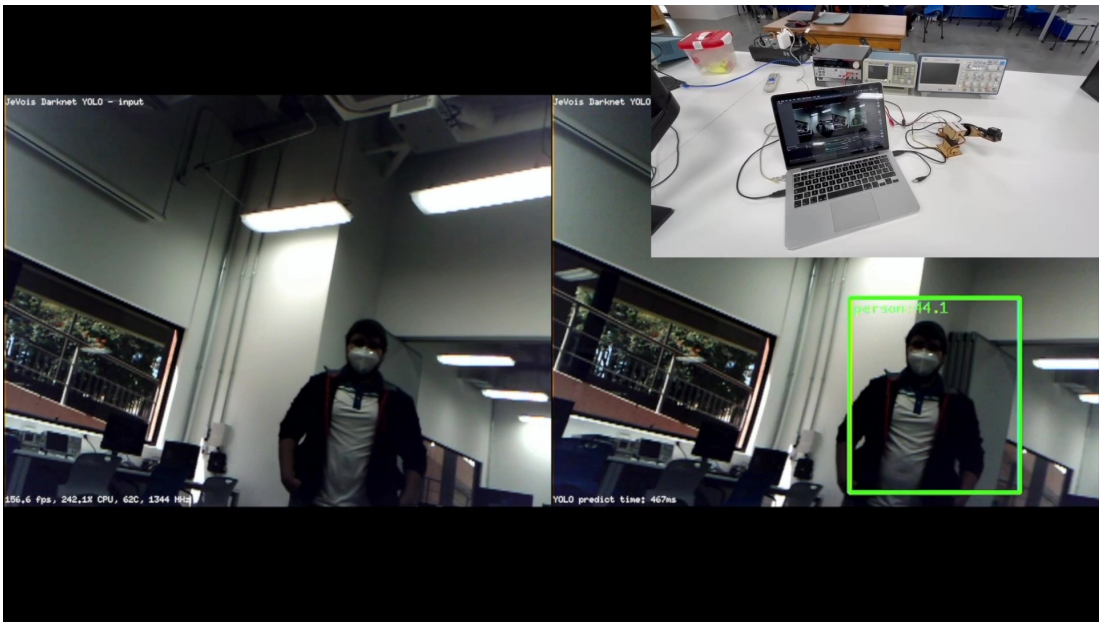


Figura 71: Pruebas de controlador IBVS usando algoritmo YOLO.

CAPÍTULO 11

Discusión

Es importante mencionar que los resultados obtenidos en la aplicación final, y en general en el trabajo de graduación, fueron en exceso difíciles de obtener. Esto se debió a que la documentación para esta tecnología y este tipo de aplicaciones es escasa y limitada en su contenido. Además, la integración de sistemas de aprendizaje automático en microcontroladores es compleja, ya que requiere de muchos recursos de los dispositivos y no todos los dispositivos son capaces de ejecutarlos aún realizando una buena implementación en código, y su desarrollo requiere de participación de los fabricantes del dispositivo como de los desarrolladores de la herramienta, por lo que es difícil de lograr. Este tipo de situaciones demuestran que esta tecnología es muy reciente y que le falta madurar para alcanzar su máximo potencial, que promete ser alto. En este momento, los desarrolladores comercializan la tecnología como algo completamente desarrollado y sin limitaciones, pero la realidad es bastante diferente, ya que existen casos donde ni siquiera los desarrolladores principales han encontrado solución a las problemáticas y evitan revelar sus limitaciones.

En retrospectiva, los resultados obtenidos fueron muy positivos y demostrativos del gran potencial de la tecnología que deberá ser más conocida para integrar una comunidad más grande que ayude a desarrollar y ampliar las aplicaciones que se pueden realizar. Esta es la razón por la que se considera conveniente la continuación de esta línea de investigación, ya que abre las puertas a nuevas aplicaciones mediante sistemas embebidos que utilicen la tecnología presentada, lo que permitirá avances y exploración que fortalecerán su utilización y entendimiento de parte de la comunidad guatemalteca.

CAPÍTULO 12

Conclusiones

1. El flujo de trabajo de Tiny ML mediante TensorFlow muestra facilidad para implementación de modelos de regresión, aunque cuenta con problemas de implementación al utilizar sensores.

2. La implementación de un modelo de TFLite utilizando cuantificación fue satisfactoria al mostrar reducciones de tamaño del modelo de 1600 y 2200 bytes en diferentes casos, lo que es favorable para implementación en microcontroladores.

3. El control basado en visión implementado mediante IBVS mostró mayor robustez y versatilidad al interactuar con diferentes técnicas de visión por computadora y ejecución en tiempo real e iterativa.

CAPÍTULO 13

Recomendaciones

1. Evaluar la utilización de MicroAI, CMSIS-NN u otros flujos de trabajo que permitan implementación de RNs en microcontroladores.
2. Evaluar la utilización de IMU externas los dispositivos que ejecutan los modelos para recolección de datos.
3. Desarrollar una librería propia para el manejo de la IMU a utilizar.
4. Fabricar un estuche o sujetador para el disposito Arduino Nano 33 BLE Sense en la aplicación de reconocimiento de patrones, de forma que se facilite su sujeción y mantener el mismo agarre durante los movimientos.
5. Utilizar módulos de visión que permitan la implementación de forma nativa del algoritmo YOLO, como OpenMV CAM.

CAPÍTULO 14

Bibliografía

[1] J. Schmidhuber, «My First Deep Learning System of 1991 + Deep Learning Timeline 1962-2013,» *CoRR*, vol. abs/1312.5548, 2013. arXiv: 1312.5548. dirección: http://arxiv.org/abs/1312.5548.

[2] M. Kranz, *Internet of things*. LID Editorial Empresarial, S.L., 2017, ISBN: 9788416894895. dirección: https://books.google.com.gt/books?id=Szz1DwAAQBAJ.

[3] M. Merenda, C. Porcaro y D. Iero, «Edge Machine Learning for AI-Enabled IoT Devices: A Review,» *Sensors*, vol. 20, n.º 9, 2020, ISSN: 1424-8220. DOI: 10.3390/s20092533. dirección: https://www.mdpi.com/1424-8220/20/9/2533.

[4] W. Li y M. Liewig, «A Survey of AI Accelerators for Edge Environment,» en *Trends and Innovations in Information Systems and Technologies*, Á. Rocha, H. Adeli, L. P. Reis, S. Costanzo, I. Orovic y F. Moreira, eds., Cham: Springer International Publishing, 2020, págs. 35-44, ISBN: 978-3-030-45691-7.

[5] O. Guzide y S. Sloboda, «Is Apple's new M1 chip a gamechanger in computing?» *Proceedings of the West Virginia Academy of Science*, vol. 93, n.º 1, 2021.

[6] Y. Wang, Q. Wang, S. Shi, X. He, Z. Tang, K. Zhao y X. Chu, «Benchmarking the Performance and Energy Efficiency of AI Accelerators for AI Training,» en *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, págs. 744-751. DOI: 10.1109/CCGrid49817.2020.00-15.

[7] Google, *Frequently asked questions on TPUs*, 2020. dirección: https://coral.ai/docs/edgetpu/faq/.

[8] P. Warden y D. Situnayake, *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-low-power Microcontrollers*. O'Reilly Media, Incorporated, 2020, ISBN: 9781492052043. dirección: https://books.google.com.gt/books?id=sB3mxQEACAAJ.

[9] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens,

Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu y Xiaoqiang Zheng, *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, Software available from tensorflow.org, 2015. dirección: https://www.tensorflow.org/.

[10] R. Szeliski, *Computer vision : algorithms and applications*. London New York: Springer, 2021, ISBN: 9781848829343.

[11] J. McCarthy, «What is artificial intelligence,» 2007.

[12] X. Zhang, *A matrix algebra approach to artificial intelligence*. Singapore: Springer, 2020, ISBN: 9789811527708.

[13] I. Goodfellow, Y. Bengio, A. Courville e Y. Bengio, *Deep learning*, 2. MIT press Cambridge, 2016, vol. 1.

[14] S. Ma y M. Belkin, *Diving into the shallows: a computational perspective on large-scale shallow learning*, 2017. arXiv: 1703.10622 [stat.ML].

[15] J. Redmon, S. K. Divvala, R. B. Girshick y A. Farhadi, «You Only Look Once: Unified, Real-Time Object Detection,» *CoRR*, vol. abs/1506.02640, 2015. arXiv: 1506.02640. dirección: http://arxiv.org/abs/1506.02640.

[16] B. Siciliano, L. Sciavicco, L. Villani y G. Oriolo, *Robotics*. Springer London, 2009. DOI: 10.1007/978-1-84628-642-1. dirección: https://doi.org/10.1007/978-1-84628-642-1.

[17] P. Corke, *Robotics, Vision and Control*. Springer International Publishing, 2017. DOI: 10.1007/978-3-319-54413-7. dirección: https://doi.org/10.1007/978-3-319-54413-7.

[18] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, R. Rhodes, T. Wang y P. Warden, «TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems,» *CoRR*, vol. abs/2010.08678, 2020. arXiv: 2010.08678. dirección: https://arxiv.org/abs/2010.08678.

[19] T. A. Team, *Accessing accelerometer data on Nano 33 BLE Sense: Arduino documentation*, 2021. dirección: https://docs.arduino.cc/tutorials/nano-33-ble-sense/imu_accelerometer.

[20] H. Y. Pratama, *How to automatically download bulk images for your dataset using Python*, mayo de 2021. dirección: https://python.plainenglish.io/how-to-automatically-download-bulk-images-for-your-dataset-using-python-f1efffba7a03.

[21] S. Canu, *Train yolo to detect a custom object (online with free GPU)*, oct. de 2020. dirección: https://pysource.com/2020/04/02/train-yolo-to-detect-a-custom-object-online-with-free-gpu/.

[22] J. Redmon, *Darknet: Open Source Neural Networks in C*, http://pjreddie.com/darknet/, 2013-2016.

[23] N. Tijtgat, *Understanding YOLOv2 training output*, jun. de 2017. dirección: https://timebutt.github.io/static/understanding-yolov2-training-output/.

[24] Rafi, *Train your own tiny yolo V3 on google colaboratory with the custom dataset*, jul. de 2019. dirección: https://medium.com/@today.rafi/train-your-own-tiny-yolo-v3-on-google-colaboratory-with-the-custom-dataset-2e35db02bf8f.

[25] M. Samylov, *Free CAD Designs, Files & 3D Models: The GrabCAD Community Library*, mar. de 2018. dirección: https://grabcad.com/library/hitec-hsr-2645cr-1.

[26] JeVois, *Standardized serial messages formatting*, ene. de 2017. dirección: http://jevois.org/doc/UserSerialStyle.html.

[27] ——, *Making a motorized pan-tilt head for JeVois and tracking objects*, ene. de 2017. dirección: http://jevois.org/tutorials/UserPanTilt.html.

[28] P. Corke, *Robotics toolbox*, abr. de 2020. dirección: https://petercorke.com/toolboxes/robotics-toolbox/.

CAPÍTULO 15

Anexos

15.1. Documentación: Enlace para acceso a desarrollo del trabajo de graduación

Enlace a carpeta de Google Drive.

15.2. Aplicación 3: Imagen de plano para corte de piezas de corte láser

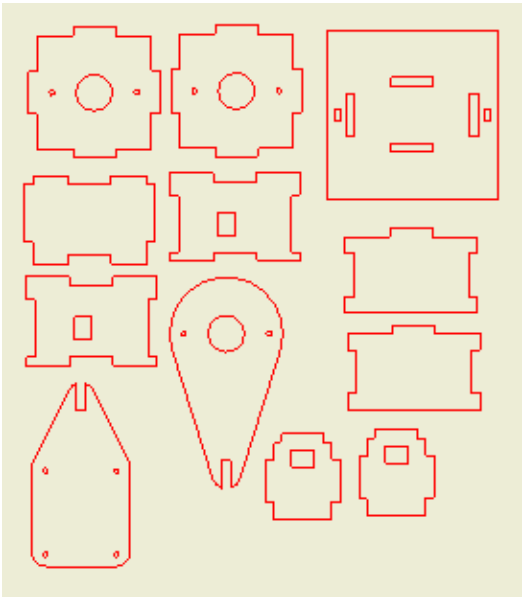
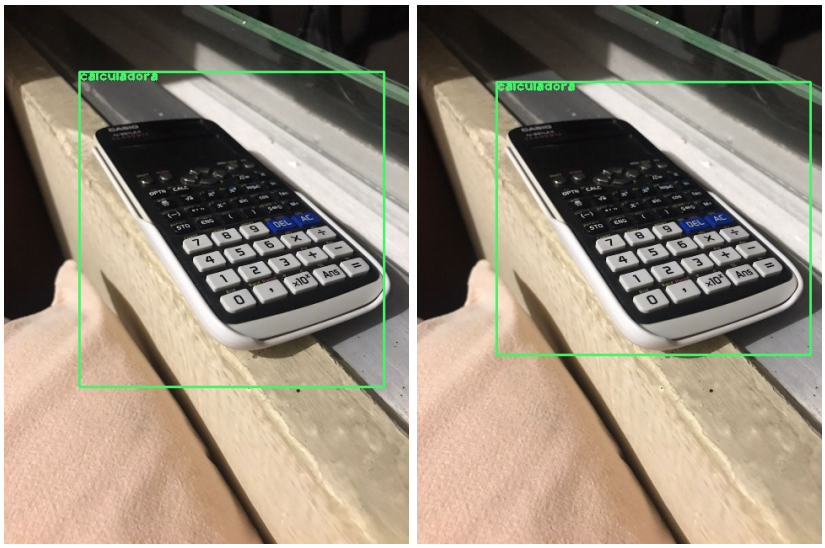


Figura 72: Plano de piezas para estructura para brazo en formato para corte láser.

15.3. Aplicación 3: Inferencias de algoritmos YOLO en imágenes de prueba

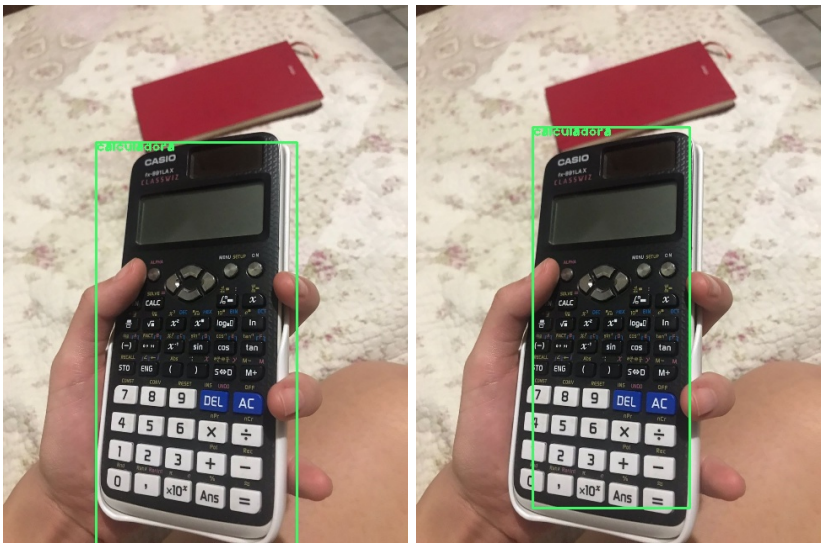


(a) YOLO v3. (b) Tiny YOLO v3.

Figura 73: Inferencia de algoritmos YOLO desarrollados sobre imagen de prueba #1.

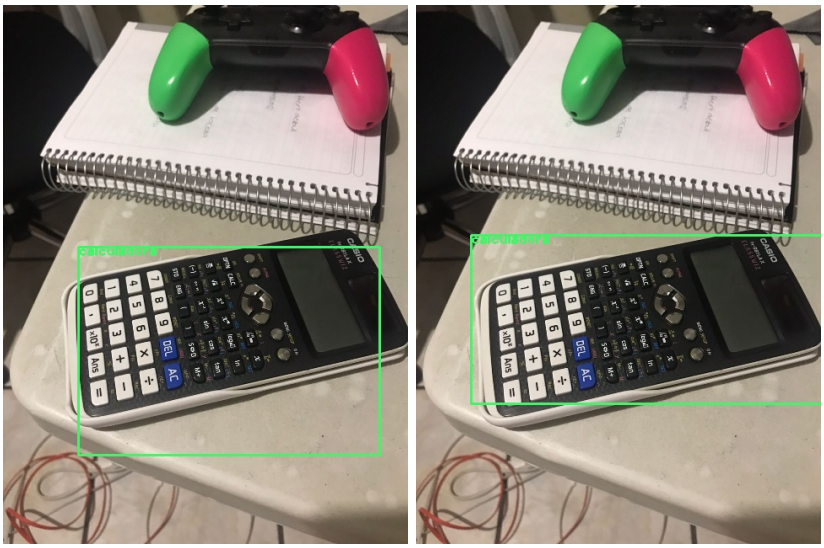
(a) YOLO v3. (b) Tiny YOLO v3.

Figura 74: Inferencia de algoritmos YOLO desarrollados sobre imagen de prueba #2.



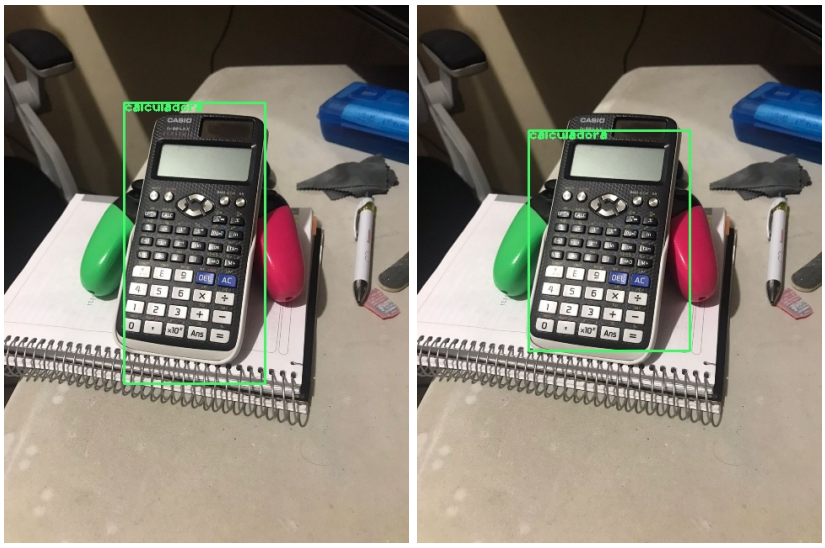
(a) YOLO v3. (b) Tiny YOLO v3.

Figura 75: Inferencia de algoritmos YOLO desarrollados sobre imagen de prueba #3.



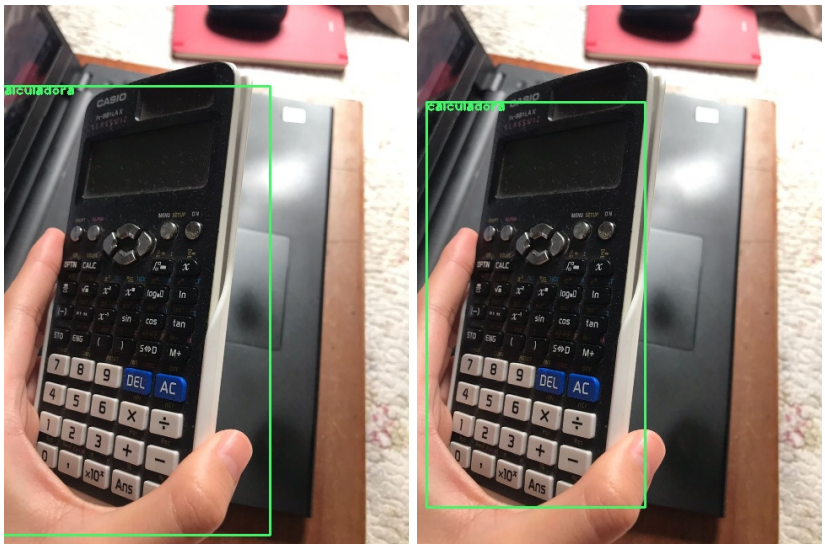
(a) YOLO v3. (b) Tiny YOLO v3.

Figura 76: Inferencia de algoritmos YOLO desarrollados sobre imagen de prueba #4.



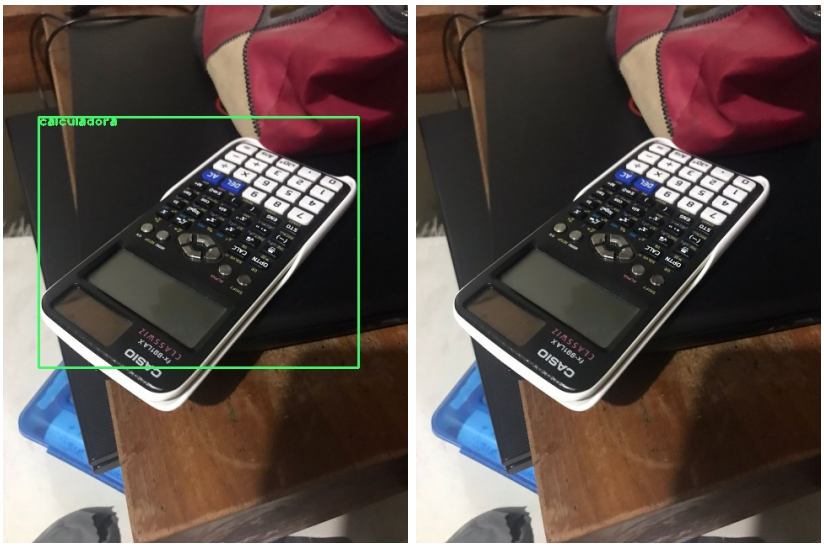
(a) YOLO v3. (b) Tiny YOLO v3.

Figura 77: Inferencia de algoritmos YOLO desarrollados sobre imagen de prueba #5.



(a) YOLO v3. (b) Tiny YOLO v3.

Figura 78: Inferencia de algoritmos YOLO desarrollados sobre imagen de prueba #6.

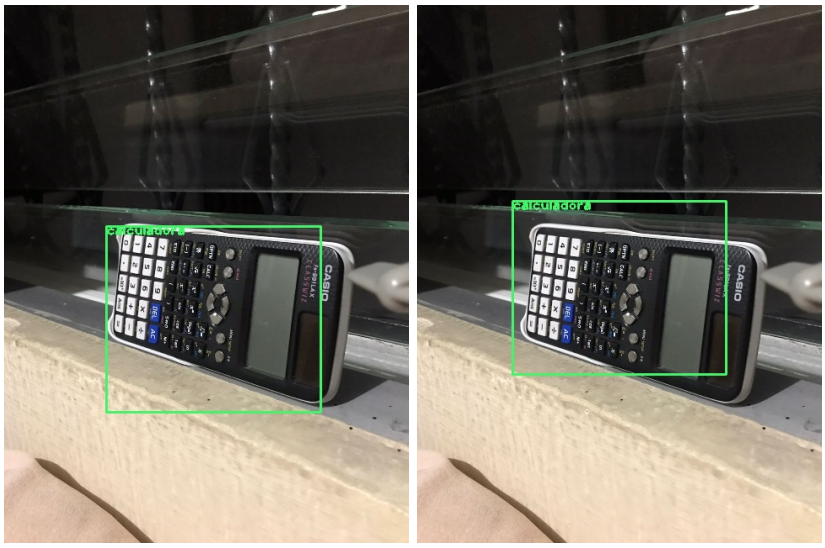


(a) YOLO v3. (b) Tiny YOLO v3.

Figura 79: Inferencia de algoritmos YOLO desarrollados sobre imagen de prueba #7.

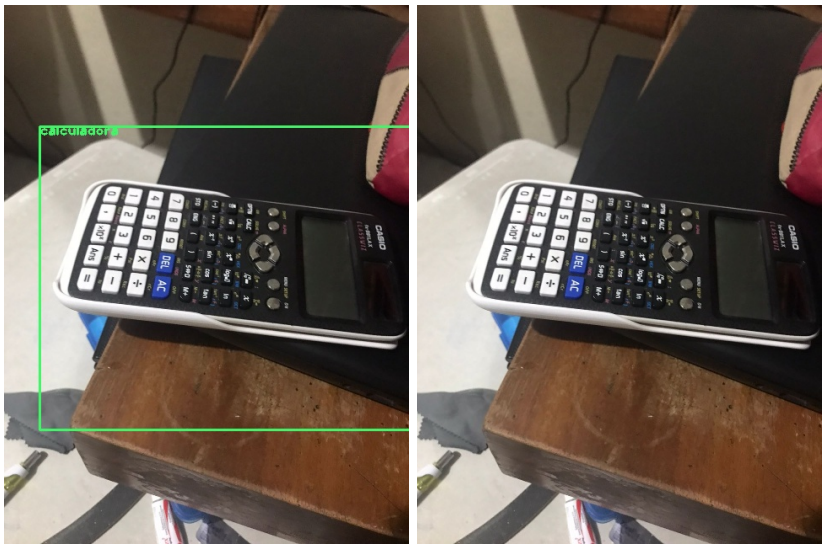
(a) YOLO v3. (b) Tiny YOLO v3.

Figura 80: Inferencia de algoritmos YOLO desarrollados sobre imagen de prueba #8.



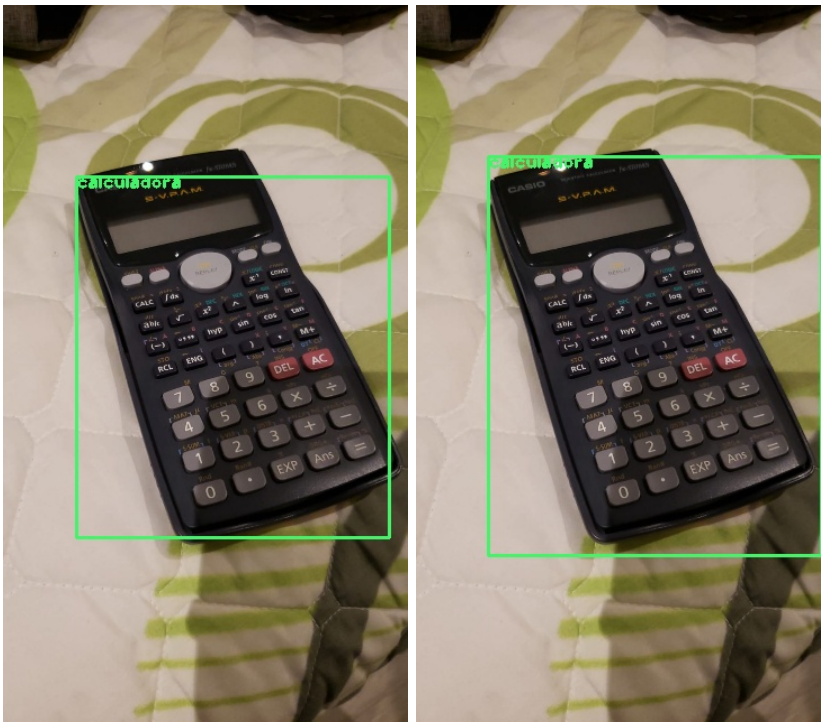
(a) YOLO v3. (b) Tiny YOLO v3.

Figura 81: Inferencia de algoritmos YOLO desarrollados sobre imagen de prueba #9.



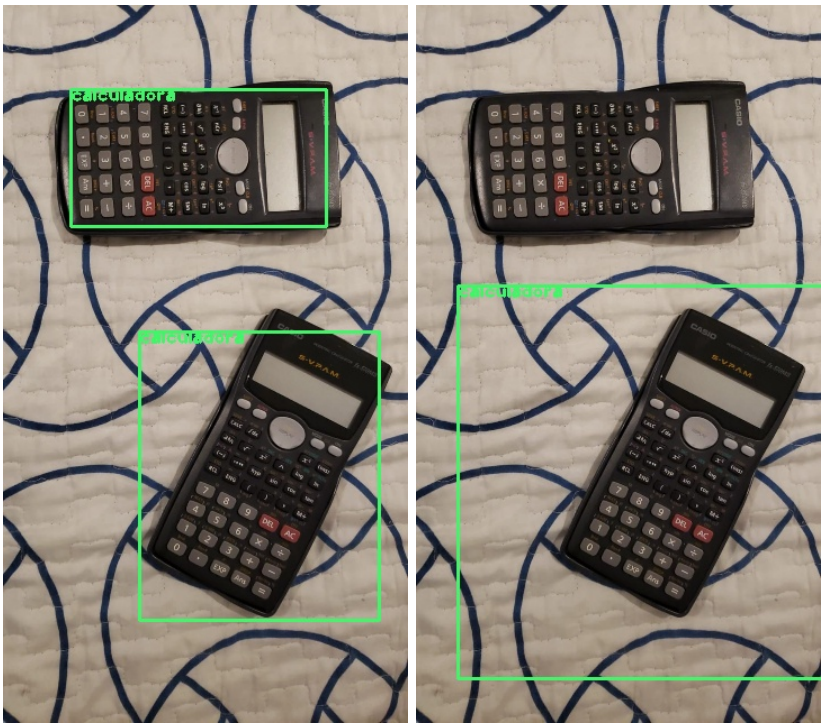
(a) YOLO v3. (b) Tiny YOLO v3.

Figura 82: Inferencia de algoritmos YOLO desarrollados sobre imagen de prueba #10.



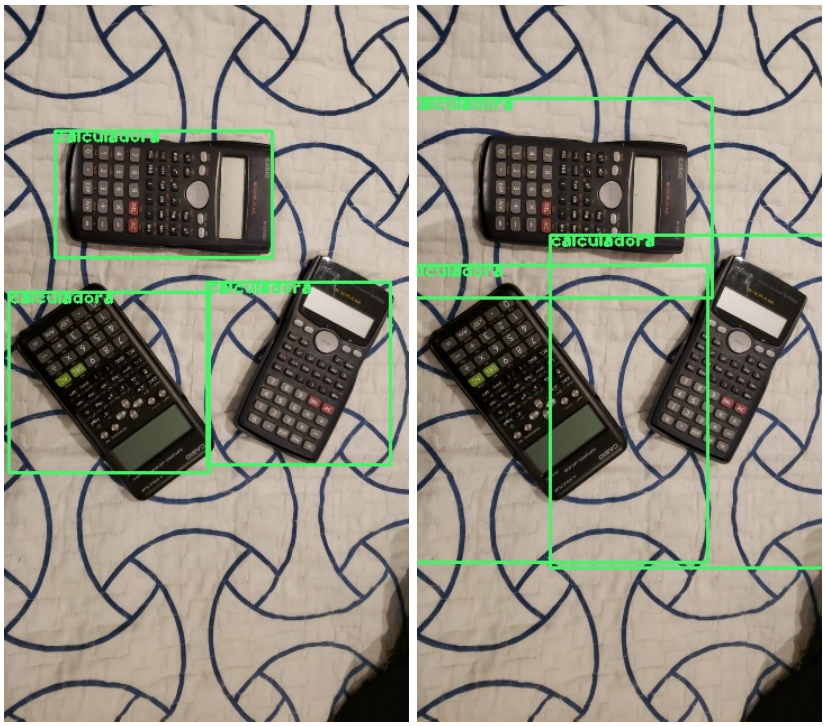
(a) YOLO v3. (b) Tiny YOLO v3.

Figura 83: Inferencia de algoritmos YOLO desarrollados sobre imagen de prueba #11.



(a) YOLO v3. (b) Tiny YOLO v3.

Figura 84: Inferencia de algoritmos YOLO desarrollados sobre imagen de prueba #12.



(a) YOLO v3. (b) Tiny YOLO v3.

Figura 85: Inferencia de algoritmos YOLO desarrollados sobre imagen de prueba #13.