



MAESTRÍA EN SISTEMAS EMBEBIDOS

MEMORIA DEL TRABAJO FINAL

Cámara IoT para detección facial con conectividad Wi-Fi

Autor:

Esp. Ing. Mauricio Barroso Benavides

Director:

Mg. Ing. Gonzalo Sanchez (FF.AA, FIUBA)

Jurados:

Mg. Ing. Edgardo Torrelli (FIUBA)

Mg. Lic. Leopoldo Zimperz (FIUBA)

Mg. Ing. Sebastián Guarino (FIUBA)

*Este trabajo fue realizado en la ciudad de Tupiza,
entre junio de 2021 y junio de 2023.*

Resumen

Esta memoria describe el proceso de desarrollo de un dispositivo electrónico compuesto principalmente por un módulo de procesamiento con conectividad Wi-Fi y una cámara, que puede capturar imágenes para procesarlas mediante algoritmos de Inteligencia Artificial y así detectar rostros humanos. Los datos generados por el dispositivo son transmitidos hacia servidores en la nube encargados de procesar, almacenar y facilitar su visualización para los usuarios finales. La principal aplicación de este dispositivo consiste en generar información sobre la presencia de personas en ambientes cerrados.

En la realización del presente trabajo se utilizaron conocimientos adquiridos a lo largo de la carrera como desarrollo de firmware, visión artificial, sistemas distribuidos, gestión de proyectos y gestión de tecnología.

Agradecimientos

A Gonzalo Sanchez, director de este trabajo, por sus valiosos consejos y criterios que se ven reflejados a lo largo de este desarrollo.

A los profesores de la Maestria en Sistemas embebidos, por contribuir en mi formación académica con sus conocimientos y experiencias.

Índice general

Resumen	I
1. Introducción general	1
1.1. Inteligencia artificial, aprendizaje automático y aprendizaje profundo	1
1.1.1. Inteligencia artificial	1
1.1.2. Aprendizaje automático	2
1.1.3. Aprendizaje profundo	2
1.2. Redes neuronales convolucionales	3
1.2.1. Capa de convoluciones	4
1.2.2. Capa de <i>pooling</i>	5
1.2.3. Capa <i>fully-connected</i>	5
1.3. Vision artificial	6
1.4. Servicios en la nube	7
1.4.1. Software como un servicio	7
1.4.2. Infraestructura como un servicio	7
1.4.3. Plataforma como un servicio	7
1.5. Motivación	7
1.6. Estado del arte	8
1.7. Objetivos y alcance	8
1.8. Requerimientos	9
2. Introducción específica	11
2.1. Funcionamiento general del sistema	11
2.1.1. Tarjeta de desarrollo	12
2.1.2. Sensor de movimiento	13
Sensor pasivo infrarrojo	13
Amplificador operacional	14
2.1.3. Cámara	16
2.2. Redes Convolucionales en Cascada Multitarea	16
2.2.1. Red de propuestas	17
2.2.2. Red de refinamiento	17
2.2.3. Red de salida	18
2.2.4. Tareas	18
2.3. Biblioteca TensorFlow	19
2.4. Servicios Web de Amazon	20
2.4.1. Servicio IoT Core	20
2.4.2. Servicio TimeStream	21
2.5. Plataforma Grafana	22
3. Diseño e implementación	23
3.1. Detección facial con TensorFlow y TensorFlow Lite	23
3.2. Desarrollo del firmware	27

3.2.1. Detección facial con TensorFlow Lite Micro	29
3.2.2. Comunicación con los servicios en la nube	32
3.2.3. Gestión del consumo energético	35
3.3. Procesamiento y visualización en la nube	38
3.3.1. Gestión de dispositivos con IoT Core	38
3.3.2. Bases de datos de series temporales con Timestream	40
3.3.3. Visualización de datos con Grafana	41
4. Ensayos y resultados	43
4.1. Banco de pruebas	43
4.2. Pruebas sobre los modelos	44
4.3. Pruebas sobre el sensor de movimiento	46
4.4. Pruebas de consumo energético sobre el sistema	47
4.5. Pruebas sobre los servicios en la nube	48
5. Conclusiones	51
5.1. Conclusiones generales	51
5.2. Próximos pasos	52

Índice de figuras

1.1. Diferencias entre AI, ML y DL	1
1.2. Arquitectura de una red neuronal artificial.	3
1.3. Nodo de una red neuronal artificial.	3
1.4. CNN para clasificar dígitos escritos a mano ¹	4
1.5. Convolución de una entrada de 3x3 con un <i>kernel</i> de 2x2 y <i>stride</i> de 1 ²	5
1.6. Tipos de <i>pooling</i> ³	5
1.7. Componentes de un sistema de visión artificial y un sistema de visión humana.	6
1.8. Imagen procesada por un sistema de detección facial ⁴	6
1.9. Diagrama en bloques del sistema propuesto en [soa_ref]	8
2.1. Diagrama en bloques del sistema.	11
2.2. Componentes del ESP32-S3-DevKitC-1 ⁵	12
2.3. Diagrama en bloques del sensor de movimiento PIR.	13
2.4. Fotografía del sensor IRA-S230ST01 ⁶	14
2.5. Fotografía del TLV8544 en un encapsulado TSSOP-14 ⁷	15
2.6. Componentes del módulo ESP-LyraP-CAM ⁸	16
2.7. <i>Pipeline</i> de MTCNN [mtcnn_info].	18
2.8. Arquitectura de P-Net [mtcnn_info].	19
2.9. Arquitectura de R-Net [mtcnn_info].	19
2.10. Arquitectura de O-Net [mtcnn_info].	20
2.11. Diagrama de conexión entre dispositivos IoT y AWS ⁹	21
3.1. <i>Pipeline</i> detallado de MTCNN.	23
3.2. Bloque de postprocesamiento.	24
3.3. Bloque de preprocesamiento.	24
3.4. Diagrama de flujo de trabajo para la conversión ¹⁰	25
3.5. Diagrama de árbol de decisiones para el proceso de cuantización ¹¹	26
3.6. Diagrama de capas del firmware.	28
3.7. Diagrama de flujo de la tarea de detección facial.	30
3.8. Captura de pantalla del detalle del modelo O-Net para TensorFlow Lite.	31
3.9. Diagrama de flujo de la tarea de comunicación con los servicios en la nube.	32
3.10. Diagrama representativo de la memoria del ESP32-S3.	33
3.11. Diagrama representativo de la memoria del ESP32-S3.	34
3.12. Diagrama de la secuencia de operación del coprocesador ULP del ESP32-S3 ¹²	36
3.13. Diagrama de la secuencia de operación del coprocesador ULP del ESP32-S3.	36
3.14. Máquina de estados para filtrar falsos positivos del sensor de movimiento.	37

3.15. Diagrama de transición de modos de consumo del dispositivo.	37
3.16. Arquitectura de los servicios en la nube.	38
3.17. Captura de pantalla de la creación de una base de datos en Times-tream.	41
3.18. Captura de pantalla de la creación de una tabla en Timestream.	41
3.19. Captura de pantalla del <i>dashboard</i>	42
4.1. Diagrama del banco de pruebas.	43
4.2. Imagen de prueba para los modelos.	44
4.3. Resultados para TensorFlow.	44
4.4. Resultados para TensorFlow Lite sin cuantización.	45
4.5. Resultados para TensorFlow Lite con cuantización a 8 bits.	45
4.6. Resultados para TensorFlow Lite Micro con cuantización a 8 bits.	45
4.7. Señales de salida del sensor de movimiento.	46
4.8. Gráfico de consumo de corriente del sistema.	47
4.9. Gráfico de consumo de corriente del sistema.	48
4.10. Mensaje de prueba recibido en MQTT Test Client.	49
4.11. Tabla faceCounter con los datos de prueba.	50
4.12. <i>Dashboard</i> con datos de prueba para 3 dispositivos.	50

Índice de tablas

2.1. ESP32-S3-DevKitC-1 especificaciones	13
2.2. IRA-S230ST01 especificaciones	14
2.3. TLV8544 especificaciones	15
2.4. OV2640 especificaciones	17
3.1. Modelos comparativa	27
3.2. Consumo de corriente del dispositivo	38
4.1. Resultados de las pruebas sobre los modelos	46
4.2. Consumo de corriente del dispositivo	48

Este trabajo se lo dedico a mi familia, eternas gracias por su apoyo incondicional en cada etapa de mi vida.

Capítulo 1

Introducción general

En este capítulo se presentan conceptos básicos sobre las tecnologías y técnicas que fueron utilizadas en el desarrollo del trabajo. Se abordan nociones sobre inteligencia artificial, aprendizaje automático, aprendizaje profundo, redes neuronales convolucionales, visión artificial y servicios en la nube. También se citan trabajos anteriores que inspiraron a este, las motivaciones para llevarlo a cabo junto a sus objetivos y alcances.

1.1. Inteligencia artificial, aprendizaje automático y aprendizaje profundo

Inteligencia artificial (AI, *Artificial Intelligence*), aprendizaje automático (ML, *Machine Learning*) y aprendizaje profundo (DL, *Deep Learning*), son términos muy utilizados hoy en día en el mundo del desarrollo tecnológico [ai_ml_dl]. Aunque estos términos son muy parecidos, entre ellos existen dependencias que pueden ser visualizadas con ayuda de la figura 1.1.

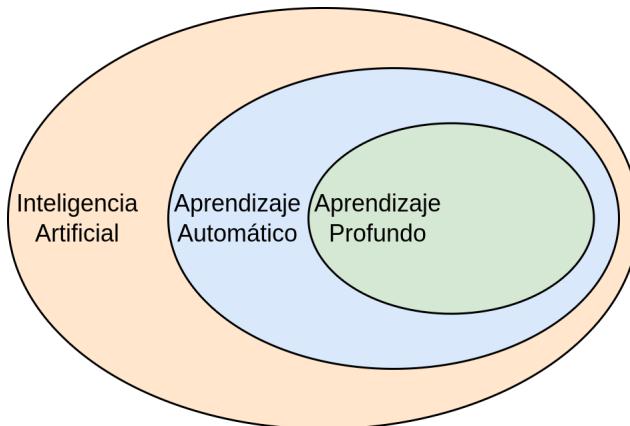


FIGURA 1.1. Diferencias entre AI, ML y DL.

1.1.1. Inteligencia artificial

La inteligencia artificial es un área de la computación que permite a los sistemas computacionales imitar la inteligencia humana para entender su entorno y tomar acciones que maximicen sus posibilidades de lograr sus objetivos [ai_def]. Sus aplicaciones más importantes se encuentran en las áreas de comercio, educación, robótica, salud, agricultura, automotriz y finanzas[ai_apps]. Todos los sistemas

de inteligencia artificial reales e hipotéticos pueden ser clasificados en alguno de los siguientes tipos [ai_types]:

- Inteligencia artificial estrecha: también conocida como inteligencia artificial débil, su objetivo es llevar a cabo un solo tipo de tarea. Estos sistemas no poseen conciencia y no son manejados por sentimientos como lo haría un humano. Algunos ejemplos son los *chatbots* o los automóviles autónomos.
- Inteligencia artificial general: también conocida como inteligencia artificial fuerte, es un concepto en el que las máquinas exhiben inteligencia humana. Estos sistemas tendrían la capacidad de aprender, entender y actuar de tal manera que sería indistinguible a un humano. Actualmente no existe, pero es utilizado conceptualmente en industrias como el cine.
- Super inteligencia artificial: también forma parte de la inteligencia artificial fuerte. Se le considera muy poderosa por ser capaz de volverse consciente y autónoma. No solo replica el comportamiento humano, sino que lo supera. Puede pensar mejor y tener más habilidades. Sin embargo, esta tecnología aún está en desarrollo.

1.1.2. Aprendizaje automático

El aprendizaje automático es un subconjunto de AI que utiliza algoritmos de aprendizaje estadísticos para construir sistemas con la habilidad de aprender automáticamente y mejorar a partir de experiencias previas sin ser explícitamente programados para esto [ml_def]. Muchos de los servicios de recomendación empleados por empresas como Netflix, YouTube o Spotify, utilizan ML para adaptarse a un usuario en particular y ofrecer una mejor experiencia más personalizada [ml_apps]. Estos algoritmos pueden ser clasificados de la siguiente manera [ml_types]

- Aprendizaje supervisado: se refiere al aprendizaje modelo a partir de un conjunto de datos, mejor conocidos como *dataset*, cuyas respuestas son conocidas con antelación y están asociadas a una etiqueta o *label*. Por ejemplo, el *dataset* pueden ser muchas fotografías de gatos y el *label* asociado el nombre de este animal. De esta manera el modelo es entrenado para generar predicciones de datos nuevos.
- Aprendizaje no supervisado: es usado cuando los datos utilizados para el aprendizaje no tienen *labels*. Su objetivo principal es aprender acerca de los datos e inferir patrones sin ningún tipo de referencia sobre las respuestas esperadas. Es mayormente empleado como parte del análisis exploratorio de datos [ai_ml_dl].
- Aprendizaje reforzado: es el aprendizaje mediante la interacción continua con el entorno con el método de prueba y error, y utiliza continuamente la retroalimentación de sus acciones y experiencias previas. Este tipo de aprendizaje emplea recompensas si se realizan acciones correctas y penalizaciones si son incorrectas.

1.1.3. Aprendizaje profundo

El aprendizaje profundo es una técnica de ML que está inspirada en la forma en la que el cerebro humano filtra información [dl_def]. Cómo DL procesa información

de manera similar al cerebro humano, sus aplicaciones son tareas que un humano generalmente realiza, como distinguir entre un peatón o un poste de luz en el caso de automóviles autónomos. El componente principal de DL son las redes neuronales artificiales, que son capas de nodos interconectados, donde existe una capa de entrada, una o varias capas ocultas y una capa de salida. En la figura 1.2 se puede observar la arquitectura de una red neuronal artificial.

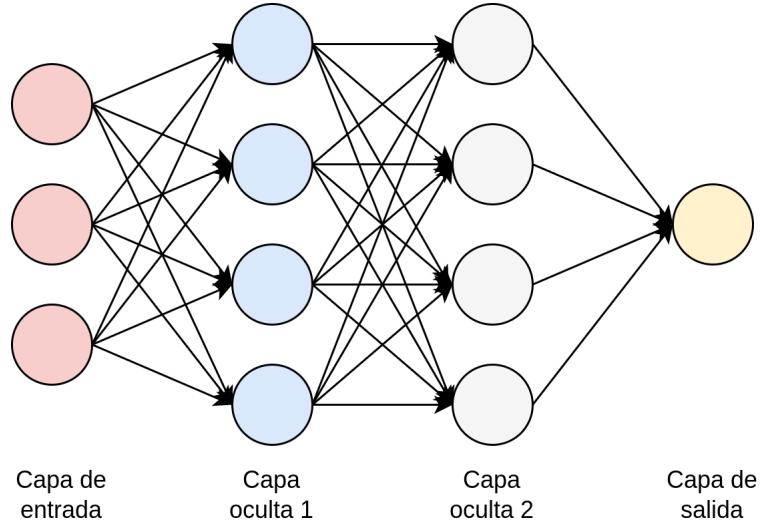


FIGURA 1.2. Arquitectura de una red neuronal artificial.

Cada uno de los nodos de las capas ocultas y de salida, tienen como entrada la salida de los nodos anteriores multiplicadas por unos términos denominados pesos o *weights* y que sumados junto a otro término llamado sesgo o *bias* pasan por una función de activación no lineal para generar su salida. En la figura 1.3 se visualiza un nodo de las capas ocultas o de salida.

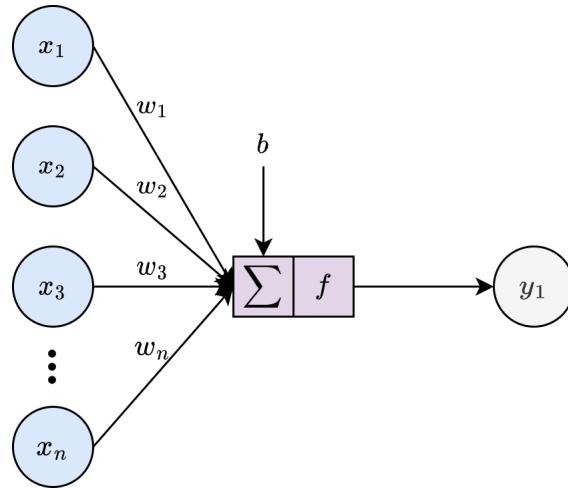


FIGURA 1.3. Nodo de una red neuronal artificial.

1.2. Redes neuronales convolucionales

También conocidas como CNN (*Convolutional Neural Networks*) por sus siglas en inglés, son un algoritmo de DL que están orientadas a recibir como entrada una

imagen digitalizada, asignarle *weights* y *biases* entrenables a varios aspectos/objetos en la imagen para poder diferenciarlas unas de otras [cnn_def]. Su uso reduce el preprocesamiento de las imágenes de entrada con respecto a otros modelos de clasificación, ya que los filtros necesarios son incorporados en su arquitectura y tienen la habilidad de ser entrenados.

Computacionalmente una imagen puede ser muy difícil de procesar, esto depende del espacio de colores donde se encuentra [cnn_colors] y las dimensiones que posee. Por ejemplo, una imagen RGB y de dimensiones 1920x1080 píxeles tiene un tamaño de 6220800 bytes. El objetivo principal de las CNN es reducir la dimensionalidad de las imágenes de entrada, de tal forma que sean más fáciles de procesar y no pierdan sus características o *features* principales que son críticas para obtener una buena predicción.

La arquitectura de una CNN es independiente del tipo de aplicación, donde las capas que lo componen son elegidas en función de los objetivos que se persiguen. En la figura 1.4 se puede observar la arquitectura de una CNN para clasificar dígitos escritos a mano.

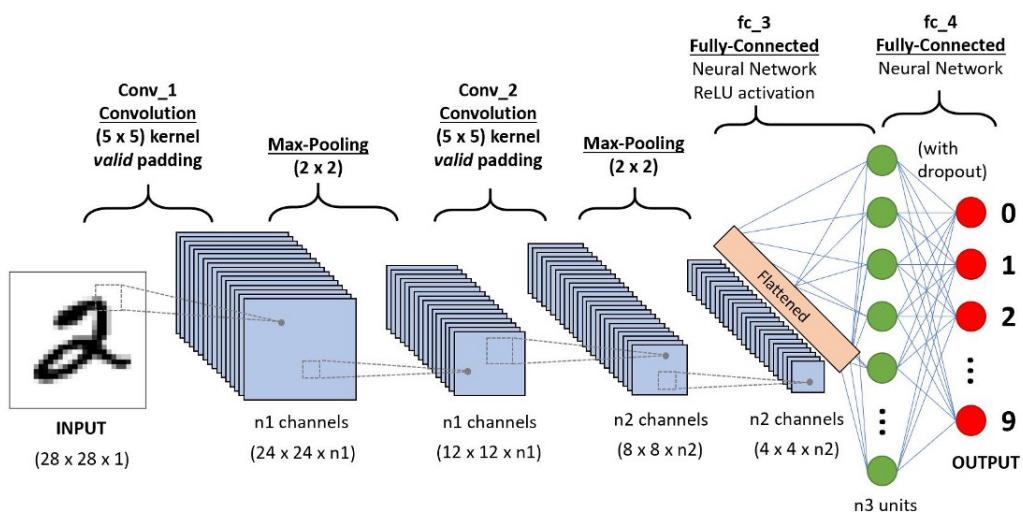


FIGURA 1.4. CNN para clasificar dígitos escritos a mano¹.

En la arquitectura de la figura 1.4 se pueden observar tres capas principales para construir una CNN: capa de convoluciones, capa de *pooling* y capa *fully-connected*.

1.2.1. Capa de convoluciones

Esta capa es la encargada de aplicar la operación de convolución sobre las imágenes de entrada para encontrar patrones que más adelante permitirán clasificarlas. La convolución de una imagen con un *kernel* no es más que la aplicación del operador punto entre ambos. Este tipo de capas se definen por:

- El número de los *kernels* o filtros que se aplican a la imagen, que es el número de matrices por las que se van a convolucionar las imágenes de entrada.

¹Imagen tomada de: <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>

- El tamaño de los *kernels*, donde casi siempre tienen dimensiones cuadradas e impares como 3x3 o 5x5.
- El *stride* o paso, se refiere a la forma en como el *kernel* recorre la imagen.

En la figura 1.5 se puede observar la operación de convolución de una entrada con dimensiones 3x3 y un *kernel* de 2x2.

Input	Kernel	Output
$\begin{array}{ c c c } \hline 0 & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array}$	$\begin{array}{ c c } \hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array}$	$=$ $\begin{array}{ c c } \hline 19 & 25 \\ \hline 37 & 43 \\ \hline \end{array}$

FIGURA 1.5. Convolución de una entrada de 3x3 con un *kernel* de 2x2 y *stride* de 1².

1.2.2. Capa de *pooling*

Similar a la capa de convoluciones, tiene el objetivo de reducir la dimensionalidad de los *features* obtenidos mediante las convoluciones aplicadas en la capa anterior, para reducir el poder computacional requerido en un principio. Existen dos tipos de dos tipos: *max pooling* y *Average pooling*. El primero retorna el valor máximo de una porción de la imagen cubierta por el *kernel* y el segundo el valor promedio o *average*. *Max pooling* también funciona como supresor de ruido al mismo tiempo que reduce la dimensionalidad. Mientras que *average pooling* solo sirve para reducir la dimensionalidad. En la figura 1.6 se pueden observar estos tipos de *pooling*.

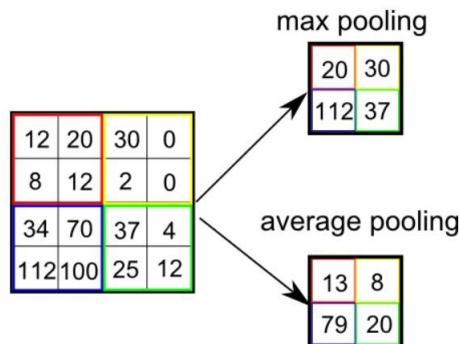


FIGURA 1.6. Tipos de *pooling*³.

1.2.3. Capa *fully-connected*

También conocida como capa lineal o FC por sus siglas en inglés, es simplemente una red neuronal artificial como la mostrada en la sección anterior y se utiliza

²Imagen tomada de: <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>

³Imagen tomada de: <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>

después de que las capas de convolución y *pooling* desglosan los *features* más importantes presentes en la imagen de entrada de la CNN. La capa FC brinda las probabilidades finales para cada *label* esperado.

1.3. Vision artificial

La visión artificial o *computer vision* es un campo científico interdisciplinario que se encarga de cómo los sistemas computacionales pueden obtener un entendimiento de alto nivel de imágenes y videos digitales para comprender y automatizar tareas como lo haría un sistema de visión humana. Las tareas que ejecuta un sistema de visión artificial son de adquisición, procesamiento, análisis y entendimiento de imágenes. En la figura 1.7 se pueden apreciar las similitudes de un sistema de visión artificial y un sistema de visión humana.

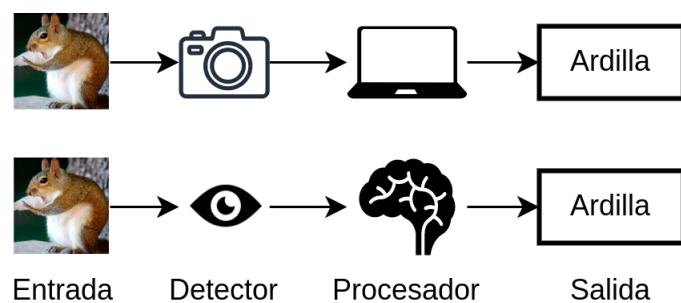


FIGURA 1.7. Componentes de un sistema de visión artificial y un sistema de visión humana.

Uno de los campos de estudio más importantes de la visión artificial es la detección facial. La detección facial puede ser considerada como un caso particular de la detección de objetos y tiene los objetivos de detectar y localizar todos los rostros humanos contenidos en una imagen digital. En la figura 1.8 se puede observar una imagen procesada por un sistema de detección facial.

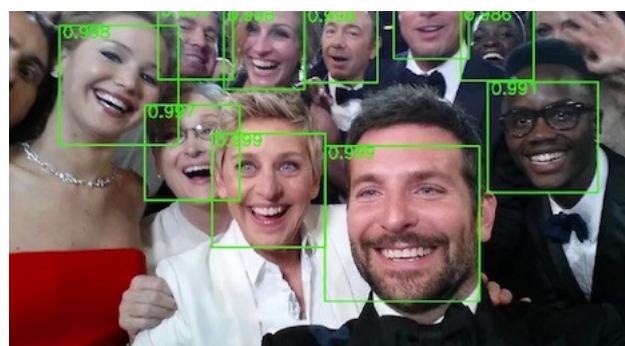


FIGURA 1.8. Imagen procesada por un sistema de detección facial⁴.

Hoy en día, muchos dispositivos comerciales y profesionales como *smartphones*, *tablets* y robots, utilizan la detección facial como primer paso para otro tipo de aplicaciones más complejas, entre las que destacan: reconocimiento facial, computación afectiva y grabación de video inteligente [fd_apps].

⁴Imagen tomada de: <https://pbarasia.medium.com/use-face-recognition-on-whatsapp-group-pictures-part-1-832c6a74b5e5>

1.4. Servicios en la nube

El término servicios en la nube hace referencia a un amplio rango de servicios ofrecidos bajo demanda a compañías y usuarios a través de internet. Estos servicios están diseñados para proveer de una manera fácil y asequible acceso a aplicaciones y recursos, sin la necesidad de una infraestructura o hardware propios [cs_def].

Los servicios en la nube son administrados totalmente por proveedores de computación en la nube o *cloud computing* [cs_def]. Estos se encuentran disponibles para los usuarios desde los servidores de los proveedores, por lo que no es necesario que una empresa aloje aplicaciones en sus propios servidores. De manera general, existen tres tipos básicos de servicios en la nube: Software como un servicio (SaaS, *Software as a Service*), Infraestructura como un servicio (IaaS, *Infrastructure as a Service*) y Plataforma como un servicio (PaaS, *Platform as a Service*).

1.4.1. Software como un servicio

En este servicio el proveedor solo proporciona el software o aplicaciones en la nube mediante internet. Los clientes tienen acceso a través de interfaces de aplicación o a través de la web, que les permite interactuar de manera sencilla, sin la necesidad de gestionar, instalar ni actualizar el software.

1.4.2. Infraestructura como un servicio

Este servicio implica la contratación de una infraestructura de hardware a un tercero, donde varios clientes comparten los recursos de una máquina física. El proveedor proporciona a sus clientes el acceso a los recursos computacionales necesarios para almacenar o ejecutar tareas que pueden incluir servidores, redes, *backup*, *firewalls*, entre otros.

1.4.3. Plataforma como un servicio

Es un servicio que se encuentra conceptualmente entre SaaS e IaaS al eliminar la parte física de la infraestructura y ofrece una plataforma donde los clientes pueden crear, desarrollar, gestionar y distribuir sus aplicaciones. El proveedor es el encargado de la gestión y mantenimiento de la plataforma y permite que los clientes se dediquen exclusivamente al desarrollo.

1.5. Motivación

Gracias a la amplia gama de plataformas de hardware y la disponibilidad de bibliotecas de código abierto para implementar AI, ML y DL, además de la difusión de información en foros y sitios web especializados, es posible desarrollar sistemas de visión artificial personalizados para distintos tipos de arquitecturas [mot_emb].

Normalmente las bibliotecas de código y los algoritmos para visión artificial no son aptos para dispositivos con poca cantidad de memoria y poder de computo reducido. Aunque gracias a los constantes avances en el desarrollo de herramientas para optimizar modelos de DL y arquitecturas de modelos más eficientes y

ligeras, es posible implementar estos algoritmos en dispositivos como microcontroladores.

La motivación principal de este trabajo fue desarrollar un sistema embebido de bajo costo económico, bajo consumo energético y de código abierto, que integre algoritmos de DL para visión artificial enfocados en cumplir eficientemente la tarea de detección facial.

Una motivación adicional fue integrar otra tecnología actual como es el internet de las cosas (IoT, *Internet of Things*), para trabajar en conjunto con los algoritmos de visión artificial. Así las aplicaciones que se pueden obtener son más versátiles a la hora de su implementación en entornos urbanos.

1.6. Estado del arte

Como antecedente existe el trabajo de Ilhan Aydin y Nashwan Adnan Othman, denominado “A new IoT combined face detection of people by using computer vision for security application” [soa_ref]. El *paper* donde se describe su trabajo presenta el desarrollo de un dispositivo electrónico que tiene como componentes principales una Raspberry Pi 3, un sensor pasivo infrarrojo (PIR, *Passive Infra Red*) y una cámara. Su objetivo principal es detectar personas con ayuda del sensor de movimiento PIR, fotografiarlas y aplicar el algoritmo de detección facial Haar Cascade [haar_cascade], para posteriormente guardar una imagen del rostro detectado y visualizarla en un teléfono móvil con ayuda de la aplicación Telegram. En la figura 1.9 se puede observar el diagrama en bloques del sistema descrito anteriormente.

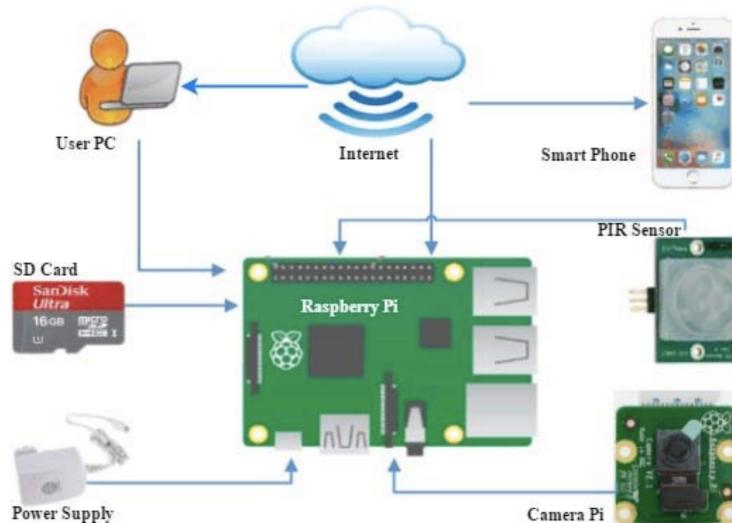


FIGURA 1.9. Diagrama en bloques del sistema propuesto en [soa_ref]

1.7. Objetivos y alcance

El objetivo principal de este trabajo fue desarrollar un sistema embebido con la capacidad de ejecutar modelos de AI para detectar y localizar rostros humanos de imágenes digitales capturadas por su cámara.

El alcance de este trabajo incluyó:

- Construir un prototipo de pruebas
- Desarrollar e implementar los modelos de AI necesarios
- Implementar los servicios en la nube necesarios

1.8. Requerimientos

Los requerimientos planteados para este trabajo fueron:

1. Requerimientos funcionales

- a) El sistema debe detectar y contar todos los rostros existentes de las imágenes obtenidas por su cámara con ayuda de las técnicas de procesamiento de imágenes *pyramid image* y *slidding window*, y modelos de DL que alcancen una precisión de al menos 80 %.
- b) El sistema debe conectarse a una red Wi-Fi existente a través de algún mecanismo de aprovisionamiento de credenciales de red.
- c) El sistema debe establecer comunicación con los servidores de AWS.
- d) El sistema debe ser alimentado mediante dos baterías AA de litio.
- e) El sistema debe poseer mecanismos de seguridad implementados tanto en hardware como en firmware para evitar su manipulación incorrecta.
- f) El sistema debe funcionar solamente si se detecta movimiento en el sector donde se encuentra instalado.

2. Requerimientos no funcionales

- a) El sistema debe tener un costo de desarrollo igual o menor a US\$200.
- b) El sistema debe tener documentación adecuada sobre su uso y desarrollo.

Capítulo 2

Introducción específica

Este capítulo expone una descripción detallada del sistema, del hardware utilizado y las herramientas de software necesarias en el desarrollo del trabajo. Se abarcan la descripción del sistema y sus componentes, los *frameworks* y modelos utilizados para detección facial y las herramientas utilizadas en la web.

2.1. Funcionamiento general del sistema

Este capítulo expone una descripción detallada del sistema, del hardware utilizado y las herramientas de software necesarias en el desarrollo del trabajo. Se abarcan la descripción del sistema y sus componentes, los *frameworks* y modelos empleados para detección facial y las herramientas utilizadas en la web.

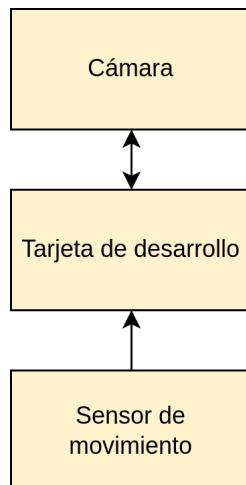


FIGURA 2.1. Diagrama en bloques del sistema.

En el sistema de la figura 2.1, cuando el sensor de movimiento detecta el movimiento de una persona genera una señal que notifica a la tarjeta de desarrollo sobre este evento. Entonces la tarjeta de desarrollo activa la cámara y obtiene una fotografía para procesarla. La imagen digital obtenida es procesada y utilizada como entrada para los modelos de DL. Cuando se obtienen las inferencias deseadas de los modelos, los resultados son procesados para transmitirlos hacia los servidores en la nube encargados de procesarlos y mostrarlos a los usuarios finales.

2.1.1. Tarjeta de desarrollo

El componente central del sistema es la tarjeta de desarrollo ESP32-S3-DevKitC-1-N8R8 de la empresa Espressif. Tiene como componente central el módulo ESP32-S3-WROOM-1-N8R8 basado en el sistema en chip (SoC, *Systems on Chip*) ESP32-S3 y varios otros componentes que simplifican el proceso de desarrollo de aplicaciones para IoT. En la figura 2.2 se observa una fotografía de la tarjeta con el detalle de sus componentes.

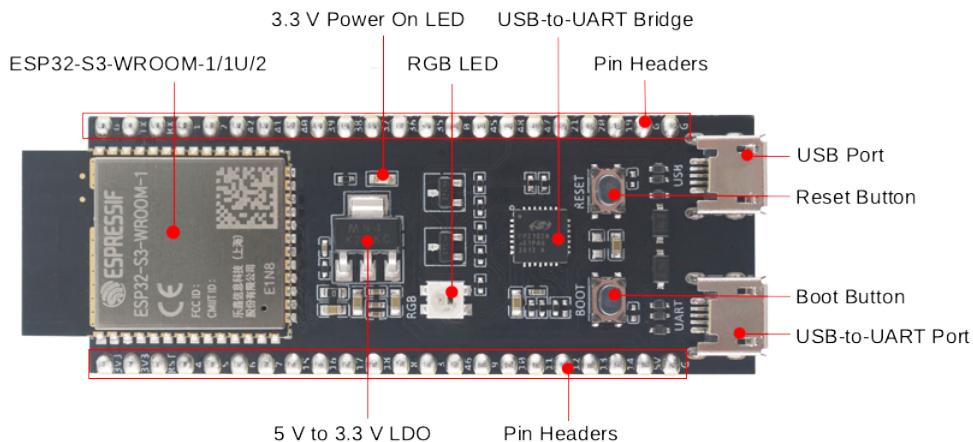


FIGURA 2.2. Componentes del ESP32-S3-DevKitC-1¹.

El módulo ESP32-S3-WROOM-1-N8R8 es una potente módulo de unidad de microcontrolador (MCU, *Microcontroller Unit*) de doble núcleo que incorpora Wi-Fi y Bluetooth de baja energía (BLE, *Bluetooth Low Energy*) y tiene un amplio conjunto de periféricos. Sus especificaciones técnicas más relevantes se detallan en la tabla 2.1.

En el mercado existen muchos fabricantes que ofrecen tarjetas de desarrollo de características técnicas que podrían haber sido utilizadas para el desarrollo de este trabajo. Sin ir muy lejos, Espressif, fabricante de la ESP32-S3-DevKitC-1-N8R8, tiene toda una familia de módulos y tarjetas muy similares entre sí. La elección de esta tarjeta en particular responde a los siguientes criterios:

- Costo: Espressif ofrece en todos sus SoCs, módulos y tarjetas de desarrollo un costo muy contenido por la gran cantidad de características ofrecidas.
- Redes neuronales: la serie de SoCs ESP32-S3 ofrece soporte para instrucciones vectoriales, que acelera las tareas de computación de redes neuronales. Esta fue la característica más importante al momento de la elección de esta tarjeta.
- Memoria: como el trabajo implicaba el uso de una cámara y, por tanto, el manejo de *buffers* de memoria de gran tamaño para manipular las imágenes obtenidas, la cantidad de memoria externa que ofrece esta tarjeta la hizo óptima para la aplicación.

¹Imagen tomada de: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/hw-reference/esp32s3/user-guide-devkitc-1.html>

TABLA 2.1. Tabla de especificaciones del ESP32-S3-DevKitC-1
[devkit_info]

Característica	Descripción
SoC embebido	ESP32-S3R8
Procesador	Xtensa LX7 doble núcleo de 32 bits
Frecuencia	Hasta 240 MHz
ROM	384 KB
SRAM	512 KB
Pines	41
Flash	8 MB
PSRAM	8 MB
Tipo de antena	PCB
Wi-Fi	802.11 b/g/n hasta 150 Mbps
Bluetooth	Bluetooth 5 y Bluetooth mesh
Periféricos	GPIO, I2C, SPI, interfaz LCD, interfaz de cámara, UART, I2S, USB, PWM, ADC, sensor táctil, sensor de temperatura, timer y watchdogs
Rango de temperatura	-40 °C a 65 °C

2.1.2. Sensor de movimiento

Un sensor de movimiento PIR basa su funcionamiento al detectar diferencias en la energía IR (*Infrared, Infrarrojo*) en el campo de visión del sensor. Debido a que la señal de salida generada por el sensor es muy pequeña, es necesario aplicar etapas de amplificación y filtrado para elevar el nivel de tensión de la señal de salida y al mismo tiempo filtrar el ruido que puede generar eventos falsos positivos. Esta salida analógica luego se debe convertir en una señal digital mediante la operación de comparación de ventanas y se puede utilizar, por ejemplo, como una interrupción en un MCU. En la figura 2.3 se muestra el diagrama en bloques del sensor de movimiento PIR.



FIGURA 2.3. Diagrama en bloques del sensor de movimiento PIR.

Sensor pasivo infrarojo

El IRA-S230ST01 es un sensor PIR fabricado por la empresa Murata. Posee una alta sensibilidad y un rendimiento confiable gracias a la tecnología cerámica y la técnica de IC (*Integrated Circuit, Circuito Integrado*) híbrida de Murata. Tiene además una sensibilidad mejorada a la interferencia de RF (*Radio Frequency, Radiofrecuencia*). Sus aplicaciones más comunes incluyen sistemas de seguridad, aparatos de iluminación, electrodomésticos, entre otros [pir_info]. En la figura

2.4 se puede observar una fotografía del IRA-S230ST01.



FIGURA 2.4. Fotografía del sensor IRA-S230ST01².

En la tabla 2.2 se detallan sus características técnicas más importantes.

TABLA 2.2. Tabla de especificaciones del IRA-S230ST01 [pir_info]

Característica	Descripción
Rango de temperatura	-40 °C a 70 °C
SNR	40 dB
Campo de vision	theta1=theta2=45 grados
Electrodo	(2.0x1.0mm)x2
Responsividad	4.6 mV
Filtro óptico	5 μm paso alto
Fuente de alimentacion	2 V a 15 V

La elección del IRA-S230ST01 como sensor PIR responde a los siguientes criterios:

- Marca: Murata es una marca muy reconocida en el mundo de los semiconductores y ofrece productos de muy alta calidad.
- Documentación: el IRA-S230ST01 cuenta con documentación muy clara sobre sus características técnicas.

Amplificador operacional

El TLV8544 es un amplificador operacional cuádruple de ultra bajo consumo de la empresa Texas Instruments, de costo optimizado para aplicaciones de detección en equipos inalámbricos y cableados de bajo consumo. El diseño del TLV8544 minimiza el consumo en dispositivos como sensores de movimiento para sistemas de seguridad, donde el tiempo de vida de la batería que los alimenta es crítico. Su uso más común es en configuraciones de amplificadores de transimpedancia con resistencias de *feedback* en el orden de los Mega ohms. Adicionalmente, tiene protección contra EMI (*Electromagnetic Interference*, Interferencia Electromagnética) que reduce la sensibilidad a las señales de RF no deseadas de fuentes como

²Imagen tomada de: <https://www.murata.com/en-sg/products/productdetail?partno=IRA-S230ST01>

teléfonos móviles, Wi-Fi y transmisores de radio [opamp_info]. En la figura 2.5 se puede observar una fotografía del TLV8544 en un encapsulado TSSOP-14.



FIGURA 2.5. Fotografía del TLV8544 en un encapsulado TSSOP-14³.

Las características técnicas más importantes del TLV8544 se presentan en la tabla 2.3.

TABLA 2.3. Tabla de especificaciones del TLV8544 [opamp_info]

Característica	Descripción
Número de canales	4
Fuente de alimentación	1.7 V a 3.6 V
Corriente de salida por canal	30 mA
Corriente de operación	500 nA
CMMR (<i>Common Mode Rejection Ratio</i> , Relación de Rechazo del Modo Común)	90 dB
Rango de temperatura	-40 °C a 125 °C
Corriente de polarización	100 fA
Ancho de banda de ganancia	8 kHz

Desde hace muchos años los amplificadores operacionales son dispositivos muy utilizados por su gran cantidad de aplicaciones, en el mercado existen una gran variedad de modelos y son fabricados por muchas empresas de semiconductores. Estos fueron los criterios de elección del TLV8544 para el presente trabajo:

- Aplicación: por sus características técnicas, el TLV8544 está diseñado para ser parte de las etapas de amplificación y filtrado en el diseño de un sensor de movimiento PIR.
- Documentación: Texas Instruments, además del correspondiente *datasheet* del TLV8544, ofrece varios documentos técnicos con ejemplos de diseño para el TLV8544.
- Costo: es un dispositivo de precio muy razonable por todas las características que ofrece.
- Consumo energético: con sus 500 nA de corriente de funcionamiento por canal, el TLV8544 es una opción ideal para aplicaciones que requieran el uso de baterías.

³Imagen tomada de: <https://www.ti.com/product/TLV8544>

2.1.3. Cámara

Otro de los componentes principales del sistema es la cámara, que permite obtener imágenes en un formato digital que posteriormente deben ser procesadas por los algoritmos de DL. Para este trabajo se utilizó el módulo ESP-LyraP-CAM. Este módulo integra un CCM (*Compact Camera Module*, Módulo de Cámara Compacto) con un sensor OV2640 en conjunto con dos reguladores de tensión para su correcto funcionamiento. En la figura 2.6 se puede observar unas fotografías del módulo ESP-LyraP-CAM y sus componentes.

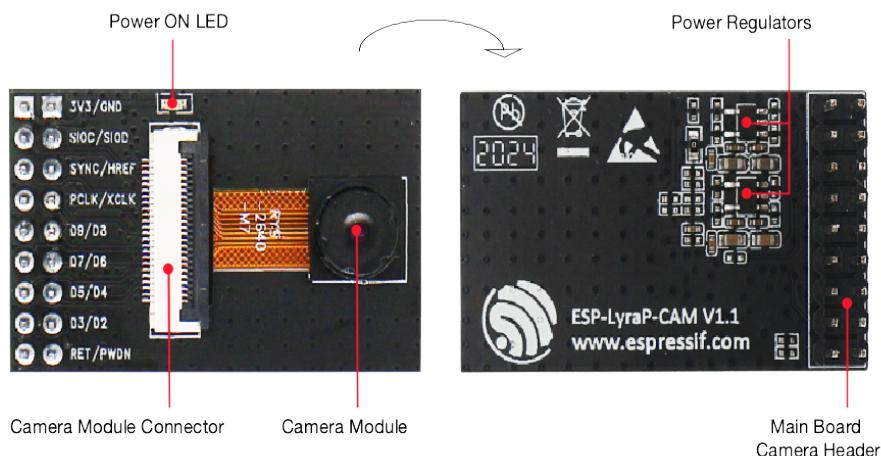


FIGURA 2.6. Componentes del módulo ESP-LyraP-CAM⁴.

El OV2640 de la empresa OmniVision es un sensor semiconductor de óxido metal complementario (CMOS, *Complementary Metal-Oxide-Semiconductor*) de 2 MP, cuenta con una interfaz de comunicación compatible con DVP (*Digital Video Port*, Puerto de Video Digital), soporta codificación JPEG y es de bajo consumo energético. En la tabla 2.4 se muestran las características técnicas más importantes del OV2640.

Los criterios para utilizar este módulo como cámara del sistema son los siguientes:

- Costo: los módulos con el sensor OV2640 tienen un costo muy reducido en comparación con otros disponibles en el mercado.
- Bajo consumo energético: como se mostró en la tabla 2.4 el consumo energético del módulo en modo *standby* es lo suficientemente bajo como para funcionar alimentado por baterías.
- Disponibilidad de código: al ser un módulo que ya lleva mucho tiempo en el mercado existen muchas bibliotecas de código para utilizarlo, lo que simplifica en gran medida el tiempo de desarrollo de *firmware*.

2.2. Redes Convolucionales en Cascada Multitarea

Las redes convolucionales en cascada multitarea (MTCNN, *Multi-task Cascaded Convolutional Networks*) son un *framework* basado en el *paper* [[mtcnn_info](#)], están

⁴Imagen tomada de: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32s2/hw-reference/esp32s2/user-guide-esp-lyrap-cam-v1.1.html>

TABLA 2.4. Tabla de especificaciones del OV2640 [camera_info]

Característica	Descripción
Tamaño de matriz	1600x1200 (UXGA) Core: 1.3 V ± 5 %
Fuente de alimentación	Analog 2.5 3.0 V I/O: 1.7 V - 3.3 V
Consumo energético	Free running: 125 mW Standby: 600 µA
Formato de imagen del sensor	1/4"
Tasa de transferencia maxima	1600x1200 a 15 fps SVGA a 30 fps CIF a 60 fps
Sensibilidad	0.6 / Lux-sec
SNR	40 dB
Rango dinámico	50 dB
Tamano de pixel	x2.2x2.2 µm
Formato de salida	YUV/RGB/MJPEG

desarrolladas para integrar las tareas de detección facial y alineamiento facial con ayuda de CNNs en cascada mediante aprendizaje multitarea. El proceso consta de tres etapas de CNNs que puede detectar rostros humanos, sus posiciones y las posiciones de sus rasgos faciales (nariz, ojos y boca). En la figura 2.7 se muestra el *pipeline* utilizado en MTCNN.

2.2.1. Red de propuestas

También conocida como P-Net (*Proposal Network*), esta etapa está compuesta de una red totalmente convolucional (FCN, *Fully Convolutional Network*). La diferencia entre una FCN y una CNN es que la FCN no utiliza una capa FC como parte de su arquitectura. Tiene la función de obtener ventanas candidatas y sus vectores de regresión de *bounding box* a partir de varias escalas de la imagen original. La regresión de *bounding box* es una técnica para predecir la localización de un cuadro delimitador en el que se encuentra el objeto que quiere ser detectado, en este caso rostros humanos. Una vez que se obtienen estos vectores, se realiza una operación conocida como NMS (*Non Max Suppression*, Supresión no Máxima) para combinar las regiones superpuestas entre sí. Finalmente, las ventanas candidatas resultantes pasan a la siguiente etapa. En la figura 2.8 se muestra la arquitectura de P-Net.

2.2.2. Red de refinamiento

Esta es una CNN denominada R-Net (*Refine Network*). Los candidatos provenientes de P-Net son la entrada de esta red. La arquitectura de R-Net reduce aún más el número de candidatos, realiza la calibración con regresión de *bounding box* y emplea NMS para fusionar candidatos superpuestos. Para cada candidato de entrada, R-Net obtiene la probabilidad de si es un rostro o no, un vector de 4 elementos que es el *bounding box* y un vector de 10 elementos que representan la localización de rasgos faciales. En la figura 2.9 se muestra la arquitectura de R-Net.

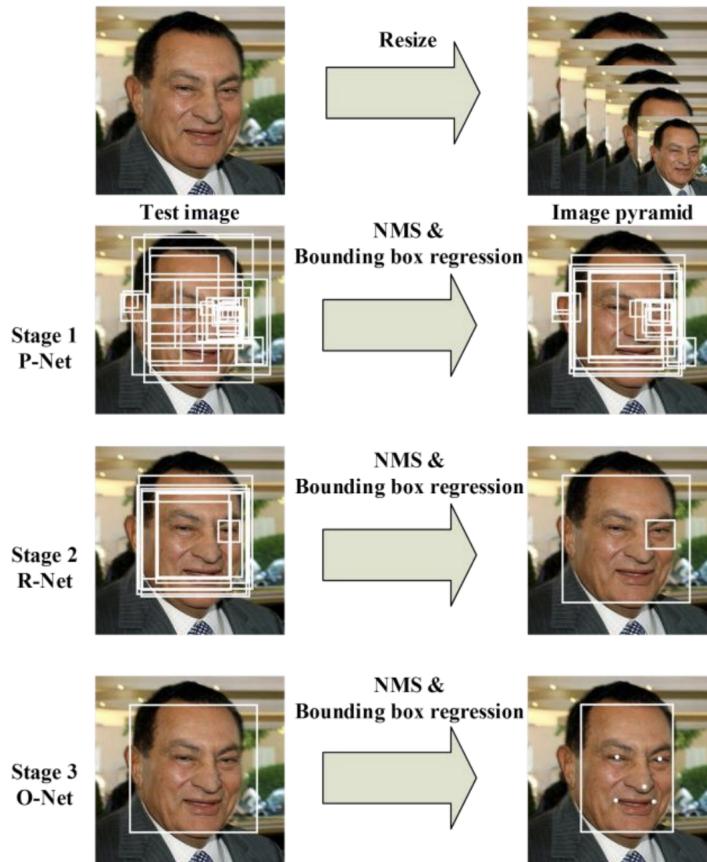


FIGURA 2.7. Pipeline de MTCNN [mtcnn_info].

2.2.3. Red de salida

Conocida como O-Net (*Output Network*), es muy similar a R-Net, pero está enfocada a describir el rostro con más detalle y generar las cinco localizaciones para ojos, boca y nariz. Su arquitectura se muestra en la figura 2.10.

2.2.4. Tareas

Como se explicó en los puntos anteriores, MTCNN se compone de tres etapas que filtran ventanas candidatas y con la ayuda de NMS y calibración con los vectores de regresión de *bounding box*, se detectan rostros y sus rasgos. Entonces, el propósito de MTCNN es cumplir con las siguientes tareas:

- Clasificación rostro/no rostro: este es un problema de clasificación binaria que utiliza una función de pérdida de entropía cruzada.
- Regresión de *bounding box*: el objetivo de aprendizaje es un problema de regresión. Para cada candidato, se calcula el *offset* entre el candidato y el *ground truth* [**ground_truth**] más cercano. La función de pérdida Euclidiana es usada para esta tarea.
- Localización de rasgos faciales: es formulada como un problema de regresión en el que la función de perdida es la distancia Euclidiana.

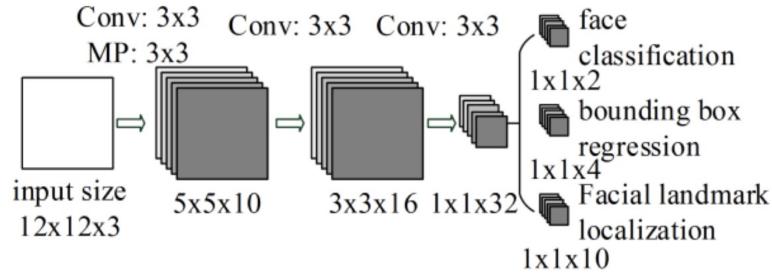


FIGURA 2.8. Arquitectura de P-Net [mtcnn_info].

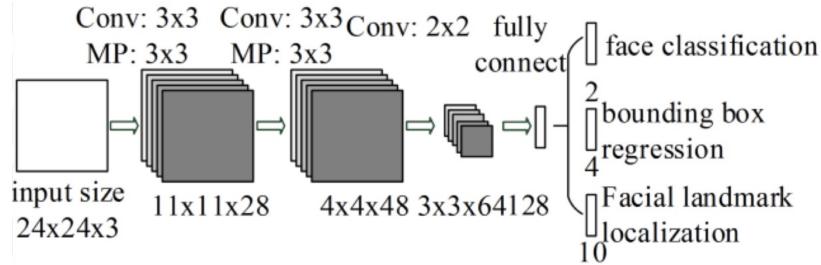


FIGURA 2.9. Arquitectura de R-Net [mtcnn_info].

2.3. Biblioteca TensorFlow

TensorFlow es una plataforma de código abierto para ML. Tiene un ecosistema completo y flexible de herramientas, bibliotecas y recursos comunitarios que permite a los desarrolladores crear e implementar fácilmente aplicaciones basadas en ML. Fue originalmente desarrollado por investigadores e ingenieros que trabajaban en el equipo Google Brain dentro de la organización de investigación de inteligencia artificial de Google y la versión inicial fue lanzada en 2015 bajo la licencia Apache License 2.0 [tf_info]. TensorFlow proporciona APIs estables y oficiales para Python y C++, aunque también existen APIs para otros lenguajes de programación que no están garantizadas de manera oficial.

Sus características principales son:

- Autodiferenciación: es el proceso de cálculo automático del vector gradiente de un modelo respecto a cada uno de sus parámetros. Ejecución ansiosa: significa que las operaciones se evalúan de manera inmediata en lugar de agregarse a un gráfico computacional que se ejecuta más tarde.
- Distribuido: TensorFlow proporciona una API para distribuir el cómputo en múltiples dispositivos tanto para ejecución ansiosa como para gráficos computacionales.
- Funciones de pérdida: TensorFlow proporciona un conjunto de funciones de pérdida, también conocidas como funciones de costo.
- Métricas: TensorFlow brinda acceso a una API de métricas de uso común que se utilizan para evaluar el rendimiento de los modelos de ML. Optimizadores: TensorFlow ofrece un conjunto de optimizadores para entrenar redes neuronales, algunos son ADAM, ADAGRAD y SGD (*Stochastic Gradient Descent*, Descenso de Gradiente Estocástico).

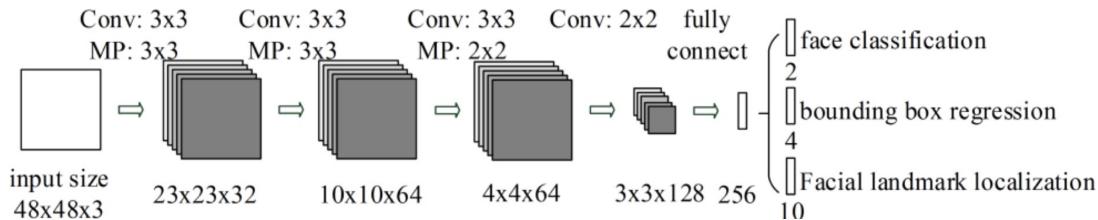


FIGURA 2.10. Arquitectura de O-Net [mtcnn_info].

Para el desarrollo de aplicaciones de ML existen varias otras bibliotecas, algunas de las más populares son: PyTorch, Caffe Computer Vision Library, Deeplearning, Neuroph, OpenNN, Theano, Torch y MXNet. Los criterios de elección de TensorFlow en este trabajo sobre las anteriores bibliotecas citadas fueron:

- Experiencia: este fue el criterio más fuerte en elección de TensorFlow como *framework* para el desarrollo de modelos de ML. El autor de este trabajo ya poseía experiencia trabajando con TensorFlow.
- Documentación: TensorFlow tiene mucha documentación oficial sobre su API y una gran variedad de tutoriales de utilización.
- Herramientas para cuantización: TensorFlow cuenta con herramientas de cuantización de datos para optimizar el tamaño y tiempos de ejecución de modelos de ML.

2.4. Servicios Web de Amazon

Mejor conocido como AWS (*Amazon Web Services*) por sus siglas en inglés, es una plataforma de *cloud computing* provista por Amazon que incluye una combinación de IaaS, PaaS y SaaS. Los servicios de AWS pueden ofrecer herramientas de poder cómputo, almacenamiento de datos y servicios de entrega de contenido [aws_info].

AWS está dividido en distintos tipos de servicios que pueden ser configurados según las necesidades de cada usuario. Estos servicios pueden dividirse en las siguientes categorías: computación, almacenamiento, bases de datos, administración de datos, migración, redes, herramientas de desarrollo, monitoreo, administración de *big data*, analíticas, AI, desarrollo móvil, mensajería y notificaciones.

De toda la extensa cantidad de servicios que ofrece AWS, para este trabajo se necesitaron solo los servicios IoT Core y Timestream.

2.4.1. Servicio IoT Core

Proporciona los servicios en la nube necesarios para conectar dispositivos IoT entre sí y a los otros servicios de AWS [iot_info]. AWS IoT proporciona software que puede ayudar a integrar dispositivos IoT en soluciones basadas en las herramientas de AWS. En la figura 2.11 se puede observar un diagrama de interconexión de dispositivos IoT y los servicios de AWS mediante IoT Core.

⁵Imagen tomada de: <https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html>

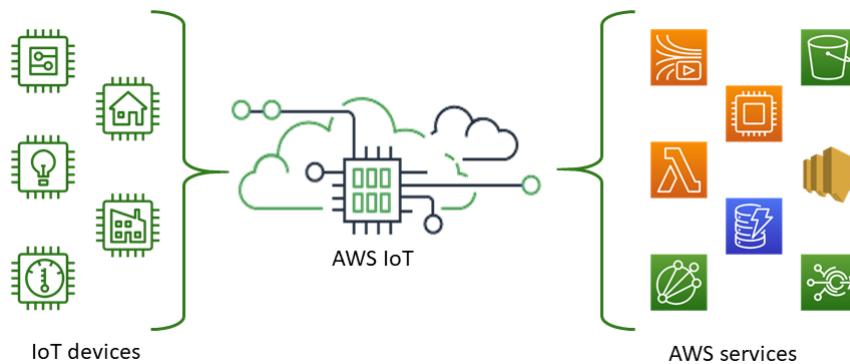


FIGURA 2.11. Diagrama de conexión entre dispositivos IoT y AWS⁵.

IoT Core permite seleccionar las tecnologías más adecuadas y actualizadas para interconectar dispositivos IoT. Los protocolos de comunicación soportados son: MQTT, MQTT sobre WSS, HTTPS y LoRaWAN.

El *broker* de IoT Core admite dispositivos y clientes que utilizan MQTT y MQTT sobre WSS para publicar y suscribirse a algún tópico. También es compatible con dispositivos y clientes que utilizan HTTPS para publicar mensajes.

2.4.2. Servicio TimeStream

Timestream es una base de datos de series temporales rápida, escalable y totalmente administrada, que facilita el almacenamiento y el análisis de billones de datos de series temporales al día. Timestream ahorra tiempos y costos con su capacidad de administrar los ciclos de vida de los datos de series temporales, donde mantiene los datos recientes en la memoria y mueve los datos históricos a un nivel de almacenamiento optimizado según las políticas definidas previamente por el usuario. El motor de consultas de Timestream permite acceder y analizar datos recientes e históricos al mismo tiempo. No necesita servidor y su tamaño se acomoda automáticamente para ajustar la capacidad y el rendimiento requeridos [[timestream_info](#)].

Los beneficios más notables que ofrece Amazon Timestream son:

- Sin servidor con escalado automático: a medida que cambian las necesidades de la aplicación, Timestream escala automáticamente para ajustar la capacidad.
- Administración de los ciclos de vida de los datos: ofrece niveles de almacenamiento, con un almacenamiento de memoria para datos recientes y un almacenamiento magnético para datos históricos. Timestream automatiza el proceso de transferencia entre ambos almacenamientos.
- Acceso simplificado a los datos: el motor de consultas de Timestream permite acceder a los datos de forma transparente, sin la necesidad de especificar el nivel de almacenamiento.
- Diseñado para series temporales: puede analizar datos de series de tiempo con SQL, con funciones integradas de series de tiempo para suavizar, aproximar e interpolar.

- Siempre cifrado: garantiza que los datos de series de tiempo siempre están cifrados. Timestream permite especificar una clave administrada para encriptar datos en el almacenamiento magnético.

2.5. Plataforma Grafana

Es una aplicación web multiplataforma de análisis y visualización interactiva. Proporciona tablas, gráficos y alertas a través de la web cuando se conecta a alguna fuente de datos compatible. Los usuarios pueden crear *dashboards* de monitoreo de datos complejos con ayuda de generadores de consultas interactivos [grafana_info].

Como herramienta de visualización, Grafana es muy popular gracias a las siguientes características:

- Se conecta a muchas fuentes de datos populares como Graphite, Prometheus, Influxdb, ElasticSearch, MySQL, PostgreSQL, entre otros.
- Es de código abierto y distribuida bajo la licencia AGPL-3.0, que permite desarrollar complementos desde cero para integrar con otras fuentes de datos.
- Ayuda a estudiar, analizar y monitorear datos durante un periodo de tiempo configurable por el usuario.
- Puede ser implementado localmente por organizaciones que quieran mantener sus datos confidenciales sin acceso a internet.
- Se pueden configurar alertas que se envían por otros medios de comunicación bajo ciertas condiciones preestablecidas.

Capítulo 3

Diseño e implementación

Este capítulo brinda una explicación detallada del proceso de implementación de los algoritmos de detección facial, diseño de firmware y la integración de los servicios en la nube con el prototipo de pruebas.

3.1. Detección facial con TensorFlow y TensorFlow Lite

Como se explicó en el capítulo 1, el objetivo principal de este trabajo es detectar rostros humanos con ayuda de algoritmos de AI. Para esto se deben obtener imágenes digitales con ayuda de una cámara, procesarlas y utilizarlas como entrada de una red de modelos de DL capaces de realizar la tarea de detección facial. Esta red de modelos fue descrita en el capítulo 2 y se denomina MTCNN.

Para implementar MTCNN adecuadamente no basta con alimentar P-Net con las imágenes obtenidas por la cámara, R-Net con las ventanas candidatas de P-Net y O-Net con las ventanas candidatas de R-Net. Los datos de entrada de cada uno de los modelos de MTCNN deben ser procesados para conseguir el mejor resultado posible, como se muestra en el diagrama de la figura 3.1.

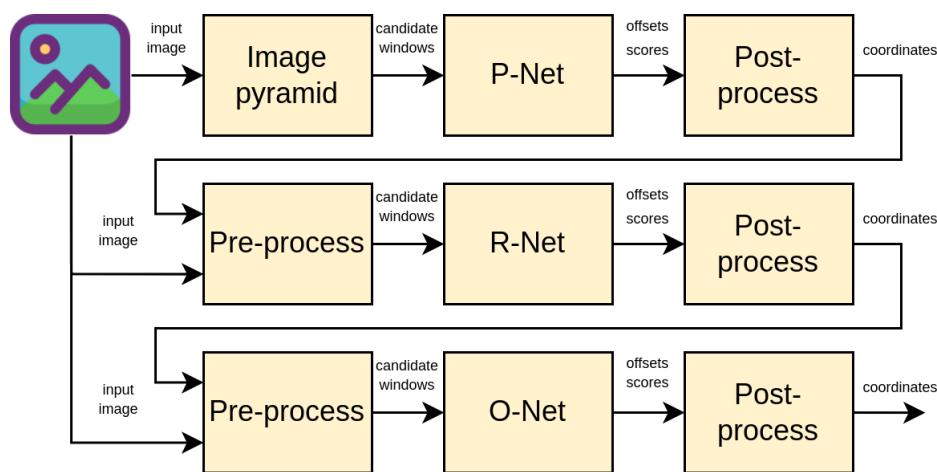


FIGURA 3.1. *Pipeline* detallado de MTCNN.

El diagrama de la figura 3.1 muestra el *pipeline* detallado de la red MTCNN, donde se pueden observar varios bloques de procesamiento, estos son:

- *Image pyramid*: genera a partir de la imagen de entrada otras imágenes de escalas inferiores, lo que permite detectar objetos de distintos tamaños. Cada nivel de escala se obtiene mediante la reducción de la escala anterior, por

lo que las imágenes en niveles superiores tienen una escala más baja que las imágenes en niveles inferiores. Después de generadas las imágenes escaladas requeridas de la imagen de entrada, estas sirven para alimentar P-Net y así detectar rostros de distintos tamaños.

- *Post-process*: en este bloque se procesan los datos de salida generados por P-Net, R-Net y O-Net. El primer subbloque realiza la operación de NMS para reducir la cantidad de ventanas candidatas que tienen solapamiento entre ellas. El segundo subbloque aplica un proceso de calibración que utiliza los *offsets* generados por los modelos para determinar de manera más precisa las coordenadas de las ventanas candidatas. Finalmente, el último subbloque corrige las coordenadas de las ventanas candidatas para que posean dimensiones cuadradas y estén dentro de los límites de la imagen original.

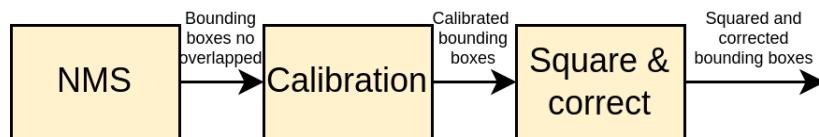


FIGURA 3.2. Bloque de postprocesamiento.

- *Pre-process*: tiene la función de procesar los datos de entrada para las redes R-Net y O-Net. El primer subbloque genera recortes de la imagen original en función de las coordenadas obtenidas del bloque *post-process*. En el segundo subbloque las imágenes recortadas de entrada son redimensionadas con dimensiones de 24x24 píxeles y 48x48 píxeles para alimentar R-Net y O-Net, respectivamente.'

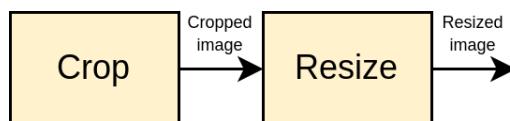


FIGURA 3.3. Bloque de preprocesamiento.

P-Net, R-Net y O-Net fueron creados con ayuda de la biblioteca para redes neuronales Keras, que es parte del *core* de TensorFlow, de acuerdo con lo expuesto en [mtcnn_info]. Se generaron 3 modelos de P-Net con diferentes dimensiones para la imagen de entrada, de tal forma que pudieran detectarse rostros a una distancia de entre 1 y 3 metros. Con las arquitecturas definidas de los modelos, el siguiente paso natural en el desarrollo debería haber sido su entrenamiento con uno o varios *datasets*, pero al ser MTCNN tan popular en el ámbito de detección facial se pudieron encontrar archivos de extensión .h5 que contenían los *weights* resultantes de un proceso de entrenamiento anterior. En el código 3.1 se pueden observar las líneas de código empleadas para crear los modelos de P-Net para TensorFlow.

```

1 # Image parameters
2 w, h, ch = (96, 96, 3) # 96x96 RGB888
3 scales = [0.3333333, 0.25, 0.125]
4
5 # List to store all P-Net models generated for all scales
6 pnets = []
7
8 # Create the models for all the scales
  
```

```

9 for scale in scales:
10    # Get the scaled width and height values
11    ws, hs = int(w * scale), int(h * scale)
12
13    # Create the TF model for the current scale and use the pre-trained
14    # weights
15    pnet_scale = mtcnn.create_pnet(ws, hs, "mtcnn_esp32s3/models_weights"
16                                    "/12net.h5")
17
18    # Append the scaled model to pnets list
19    pnets.append(pnet_scale)

```

CÓDIGO 3.1. Código para crear los modelos de P-Net con TensorFlow.

Para que los modelos obtenidos pudieran ser ejecutados en el hardware objetivo de este trabajo tuvieron que ser convertidos a un formato más liviano y eficiente llamado TensorFlow Lite. El conversor de TensorFlow Lite toma un modelo de TensorFlow y genera un modelo de TensorFlow Lite cuya extensión de archivo es .flite. La conversión puede seguir 2 caminos según como sean evaluados los modelos de TensorFlow, en la figura 3.4 se observa el flujo de trabajo del conversor.

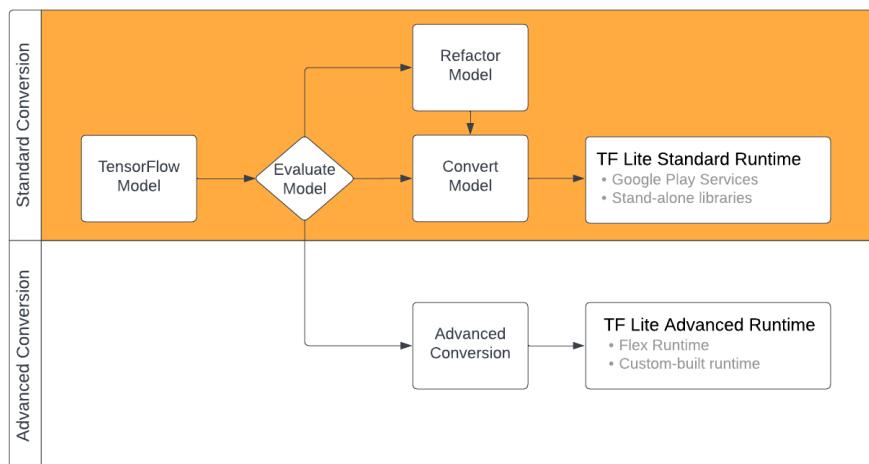


FIGURA 3.4. Diagrama de flujo de trabajo para la conversión¹.

Gracias a que todos los operadores utilizados en los modelos de TensorFlow eran compatibles con los operadores de TensorFlow Lite se realizó una conversión estándar, lo que posteriormente facilitó su implementación en el hardware destino.

Durante el proceso de conversión se aplicaron optimizaciones que responden a una necesidad de reducir aún más el tamaño y la latencia de los modelos. Se hizo una optimización por cuantización, que se refiere a la reducción de la precisión de los números usados para representar los parámetros de los modelos, los cuales por defecto son flotantes de 32 bits. Las opciones de cuantización para los modelos se tomaron del diagrama de la figura 3.5.

¹Imagen tomada de: <https://www.tensorflow.org/lite/models/convert/>

²Imagen tomada de: https://www.tensorflow.org/lite/performance/model_optimization

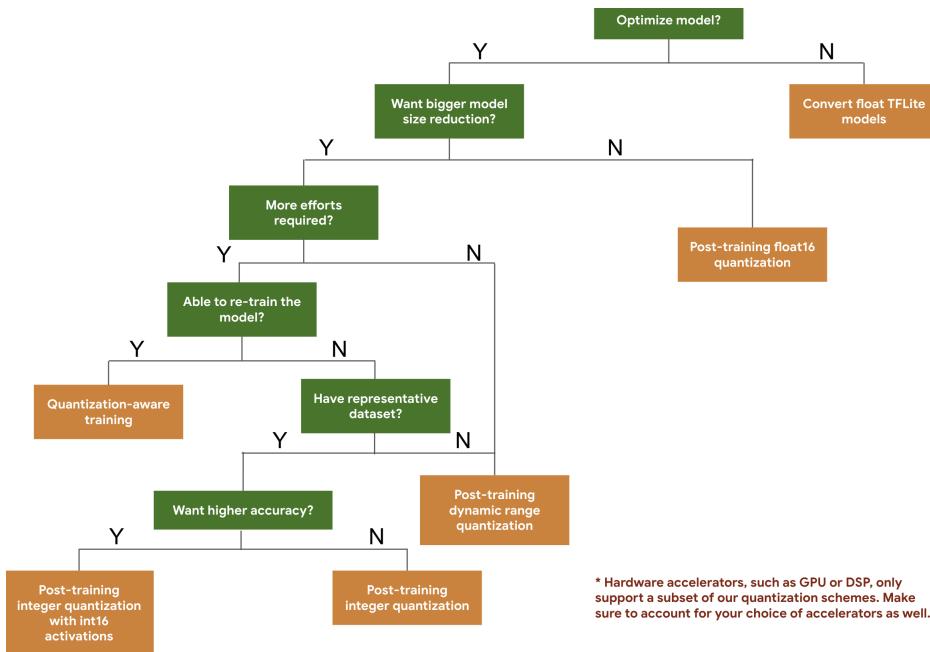


FIGURA 3.5. Diagrama de árbol de decisiones para el proceso de cuantización².

La cuantización utilizada para los modelos fue *full integer quantization*, que reduce los picos de memoria utilizados y asegura la compatibilidad con dispositivos de hardware que no pueden utilizar punto flotante. Para este tipo de cuantización se necesitó crear un *dataset* representativo, compuesto por un pequeño subconjunto (entre 100 a 500 muestras) del *dataset* de entrenamiento. En el código 3.2 se expone el código empleado para la conversión de los modelos de P-Net en TensorFlow al formato TensorFlow Lite con cuantización a 8 bits.

```

1 # List to store all P-Net int8 quantized models generated for all scales
2 pnets_quant_int8 = []
3
4 # Create the models for all the scales
5 for pnet_scale in pnets:
6     # Function to generate a representative dataset to convert to TF Lite
7     # quantized
8     def representative_dataset():
9         for i, image_file in enumerate(os.listdir("mtcnn_esp32s3/
10         representative_dataset/")):
11             image_input = cv2.imread("mtcnn_esp32s3/representative_dataset/" +
12             image_file)
13             image_input = np.expand_dims(utils.preprocess_image(image_input,
14             hs, ws, np.float32), axis=0)
15             yield [image_input.astype(np.float32)]
16
17     # Convert the TF model to the TF Lite quantized format
18     converter = tf.lite.TFLiteConverter.from_keras_model(pnet_scale)
19     converter.optimizations = [tf.lite.Optimize.DEFAULT] # Set the
      optimization flag
20     converter.target_spec.supported_ops = [tf.lite.OpsSet.
21       TFLITE_BUILTINS_INT8] # Enforce integer only quantization
22     converter.inference_input_type = tf.int8 # Define the quantization
      data type
23     converter.representative_dataset = representative_dataset # Provide a
      representative dataset to ensure we quantize correctly.
24     pnet_scale_quant_int8 = converter.convert()
  
```

```

20
21 # Append the scaled model to pnets_quant_int8 list
22 pnets_quant_int8.append(pnet_scale)

```

CÓDIGO 3.2. Código para crear los modelos de P-Net con TensorFlow Lite con cuantización de 8 bits.

La tabla 3.1 muestra los resultados de una prueba de todos los modelos obtenidos con una imagen RGB888, 96x96 píxeles y 3 rostros contenidos.

TABLA 3.1. Tabla comparativa de MTCNN

Parámetro	TensorFlow	TensorFlow Lite	TensorFlow Lite int8
Tamaño (bytes)	-	2053460	556720
Tiempo de ejecución (ms)	1.258	0.059	0.0144
Rostros	3	3	3

Todo el código para la obtención de los modelos hasta aquí expuesto, las funciones del *pipeline*, las pruebas realizadas a los modelos y el despliegue de estos en el SoC ESP32-S3, se encuentra disponible en el repositorio del trabajo [[mtcnn_repo](#)].

3.2. Desarrollo del firmware

El primer paso para el desarrollo del firmware del dispositivo fue la elección de un conjunto de herramientas de software (SDK, *Software Development Kit*). Estas herramientas permitieron implementar código para utilizar de manera eficiente todos los periféricos disponibles en el ESP32-S3. Para este proyecto el SDK utilizado fue ESP-IDF [[idf_repo](#)], las razones de su elección fueron:

- Compatibilidad: casi la totalidad de funciones en el *framework* son compatibles con todos los SoCs de la serie ESP32, salvo por algunas que solo son útiles para algunos periféricos que no se encuentran disponibles en todos los SoCs.
- Herramientas: además del código para manejar los periféricos de los SoCs ESP32, posee herramientas que son muy útiles para crear particiones, grabar código en memoria, aprovisionamiento de credenciales Wi-Fi, entre otras.
- Soporte: existen foros y sitios web especializados en el soporte de firmware y hardware para los SoCs de Espressif.
- Documentación: ESP-IDF está muy bien documentado y se dispone de muchos ejemplos que implementan las funciones de su *framework*

Con el conjunto de herramientas definido, otro aspecto de importancia fue la elección de un entorno de desarrollo para optimizar la escritura y depuración del código. El entorno de desarrollo integrado (IDE, *Integrated Development Environment*) escogido fue Eclipse IDE C/C++, los aspectos más importantes para su elección fueron:

- Herramientas: posee muchas herramientas interesantes como integración con Git, Valgrind y análisis de código, entre otros.
- Complementos: se pueden instalar diferentes tipos de *plugins* para aumentar la utilidad del IDE, para este trabajo se instaló el *plugin* ESP-IDF.

Otra herramienta importante para el proceso de desarrollo del firmware fue la utilización de software para control de versiones, que permite realizar un seguimiento de los cambios realizados en el código a lo largo del tiempo. Git fue elegido como software de control de versiones, mientras que GitHub como plataforma para alojar el repositorio de Git. Las razones para la elección de ambos son:

- Reutilización de código: se puede reutilizar código en forma de submódulos y tener los componentes de firmware descentralizados de la aplicación principal.
- Soporte: el soporte para el uso de Git y GitHub es extenso, se encuentra fácilmente en sus páginas web oficiales y en foros especializados.

Con todas las herramientas de software correctamente seleccionadas, el siguiente paso fue el diseño de la arquitectura del firmware. El firmware desarrollado siguió una arquitectura en capas, donde las capas de niveles más bajos tienen una mayor interacción con el hardware, mientras que las de niveles más altos con la aplicación del usuario. En la figura 3.6 se presenta el diagrama en capas del firmware.

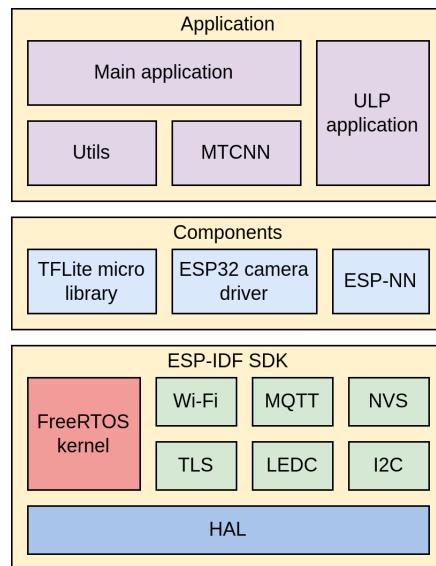


FIGURA 3.6. Diagrama de capas del firmware.

Las capas expuestas en el diagrama de la figura 3.6 son:

- **ESP-IDF SDK:** esta capa se encuentra compuesta por el *framework* ESP-IDF, que contiene todo el código necesario para controlar los periféricos del ESP32-S3.
- **Components:** esta capa está conformada por submódulos de código de Git que se encargan de controlar los componentes adicionales de hardware y las bibliotecas de código de terceros. En el repositorio del trabajo [[mtcnn_repo](#)] está representada por la carpeta `components/`.
 - TFLite micro library: biblioteca de código de TensorFlow Lite para microcontroladores [[tflm_repo](#)].
 - ESP32 camera driver: biblioteca de código para controlar diversos modelos cámaras con los SoCs ESP32 [[esp32cam_repo](#)].

- ESP-NN: biblioteca de código que implementa funciones que se utilizan para ejecutar algoritmos de ML y DL en los SoCs ESP32 [[espnn_repo](#)].
- Application: esta capa contiene el código de aplicación del dispositivo. La carpeta main/ del repositorio del trabajo [[mtcnn_repo](#)] contiene todos sus archivos.
 - Main application: código de la aplicación que corre en los procesadores principales, donde se ejecutan los procesos de detección facial y conexión con los servicios en la nube. El código se encuentra en el archivo main/main.cc
 - ULP application: código de la aplicación que se ejecuta en el coprocesador ULP, se emplea principalmente para ejecutar la aplicación principal con el menor gasto energético posible. El código de este bloque se encuentra en el archivo main/ulp/main.c
 - MTCNN: biblioteca de código que implementa las funciones del pipeline de MTCNN. Los archivos que contiene su código son main/mtcnn.cc y main/mtcnn.h
 - Image utils: código que implementa funciones para el procesamiento de imágenes. Está compuesto por los archivos main/image_utils.c e main/image_utils.h

El firmware desarrollado cumple principalmente con las siguientes tareas: detección facial con TensorFlow Lite Micro, comunicación con los servicios en la nube y gestión del consumo energético.

3.2.1. Detección facial con TensorFlow Lite Micro

El objetivo de esta tarea es obtener imágenes con la cámara del sistema para procesarlas con los modelos de MTCNN para TensorFlow Lite para microcontroladores y determinar la cantidad de rostros humanos existentes en cada imagen. El diagrama de flujo de la figura 3.7 detalla el proceso de esta tarea.

El proceso de inicialización de TFLite micro requiere de varios pasos que deben ser seguidos en orden para poder ejecutar los modelos de MTCNN correctamente. En el código 3.3 se presentan las líneas de código necesarias para la inicialización de O-Net con TFLite micro.

```

1 /* Map the model into a usable data structure */
2 const tflite::Model *onet_model = tflite::GetModel(onet_model_data);
3
4 /* Reserve memory for the tensors */
5 uint8_t *tensor_arena = (uint8_t *)heap_caps_malloc(TENSOR_ARENA_SIZE,
6   MALLOC_CAP_SPIRAM | MALLOC_CAP_8BIT);
7
8 /* Pull in only the operation implementations needed */
9 static tflite::MicroMutableOpResolver<10> micro_op_resolver;
10 micro_op_resolver.AddAveragePool2D();
11 micro_op_resolver.AddConv2D();
12 micro_op_resolver.AddPrelu();
13 micro_op_resolver.AddMaxPool2D();
14 micro_op_resolver.AddTranspose();
15 micro_op_resolver.AddFullyConnected();
16 micro_op_resolver.AddDequantize();
17 micro_op_resolver.AddDepthwiseConv2D();

```

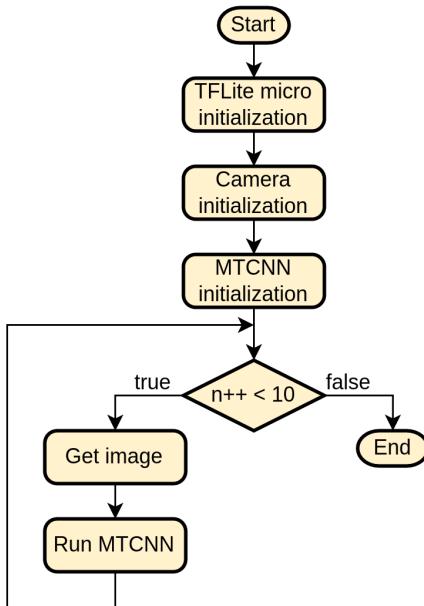


FIGURA 3.7. Diagrama de flujo de la tarea de detección facial.

```

17 micro_op_resolver.AddReshape();
18 micro_op_resolver.AddSoftmax();
19
20 /* Build an interpreter to run the model with */
21 static tflite::MicroInterpreter static_onet_interpreter(onet_model,
22   micro_op_resolver, tensor_arena, TENSOR_ARENA_SIZE);
23 onet_interpreter = &static_onet_interpreter;
24
25 /* Allocate memory from the tensor_arena for the model's tensors */
26 onet_interpreter->AllocateTensors();
  
```

CÓDIGO 3.3. Código para inicializar O-Net con TFLite micro.

Los modelos de P-Net y R-Net se inicializan de la misma forma que O-Net y una cosa a notar es la declaración de los operadores estrictamente necesarios para ejecutar estos modelos en las líneas 9 a 18. Otra aproximación más sencilla era utilizar `OpsResolver` para utilizar todos los operadores, pero esto hubiera supuesto una penalización en la cantidad de memoria RAM utilizada. En la figura 3.8 se puede observar una captura de pantalla de una página web generada con la herramienta de visualización de modelos de TensorFlow Lite que se encuentra en el repositorio oficial de TensorFlow Lite para microcontroladores [[tfm_repo](#)], donde se muestran los operadores empleados por O-Net.

Otro aspecto a destacar es la utilización de la memoria RAM pseudoestática (PS-RAM, *Pseudo static RAM*) para almacenar los tensores empleados durante la ejecución de los modelos de MTCNN. En la línea 5 del fragmento de código 3.3 se puede observar como se reserva memoria en la PSRAM de un tamaño determinado de manera experimental y denominado `TENSOR_ARENA_SIZE`.

Los métodos de MTCNN para inicializarlo y ejecutar sus modelos se encuentran en los archivos `main/mtcnn.h` y `mtcnn.cc` del repositorio [[mtcnn_repo](#)]. En el código 3.4 se pueden observar las estructuras de datos usados por los métodos de MTCNN.

```

1 typedef struct {
  
```

Operator Codes			
index	builtin_code	custom_code	version
0	CONV_2D		3
1	PRELU		1
2	MAX_POOL_2D		2
3	TRANSPOSE		2
4	RESHAPE		1
5	FULLY_CONNECTED		4
6	SOFTMAX		2
7	DEQUANTIZE		2

FIGURA 3.8. Captura de pantalla del detalle del modelo O-Net para TensorFlow Lite.

```

2   tflite :: MicroInterpreter *interpreter;
3   candidate_windows_t candidate_windows;
4   bboxes_t bboxes;
5 } model_data_t;
6
7 typedef struct {
8   model_data_t pnet[3];
9   model_data_t rnet;
10  model_data_t onet;
11 } mtcnn_t;
```

CÓDIGO 3.4. Estructura de datos de MTCNN.

Como el tipo de dato `mtcnn_t` contiene todos los datos de entrada y salida de los modelos de MTCNN, fue utilizado como parámetro de todas las funciones encargadas de ejecutar los modelos. La función que ejecuta O-Net es `mtcnn_run_onet` y en el código 3.5 se puede observar su implementación.

```

1 void mtcnn_run_onet(mtcnn_t *mtcnn, uint8_t *img, uint16_t img_w,
2   uint16_t img_h) {
3   /* Pre-process R-Net ouputs */
4
5   /* Feed the model and run it */
6   TfLiteTensor *input = mtcnn->interpreter->input(0);
7
8   for(int i = 0; i < ONET_SIZE * ONET_SIZE * 3; i++) {
9     input->data.int8[i] = ((uint8_t *) onet_image)[i] ^ 0x80;
10 }
11
12 mtcnn->interpreter->Invoke();
13
14 /* Store the scores and offsets output */
15 TfLiteTensor *scores = interpreter->output(0);
16
17 for(uint8_t j = 0; j < 2; j++) {
18   probs_buf[j + (i * 2)] = probs->data.f[j];
19 }
20
21 TfLiteTensor *offsets = interpreter->output(1);
22
23 for(uint8_t j = 0; j < 4; j++) {
24   offsets_buf[j + (i * 2)] = offsets->data.f[j];
25 }
26
27 /* Add the candidate windows to the candidate windows array */
28 add_candidate_windows();
```

```

29     /* Post-process O-Net outputs */
30 }

```

CÓDIGO 3.5. Función mtcnn_run_onet.

3.2.2. Comunicación con los servicios en la nube

Esta tarea fue diseñada para establecer conectividad con los servicios en la nube, más precisamente con el servicio IoT Core de AWS, para transmitir y recibir datos mediante el protocolo MQTT. El diagrama de flujo de la figura 3.9 muestra el proceso que sigue esta tarea.

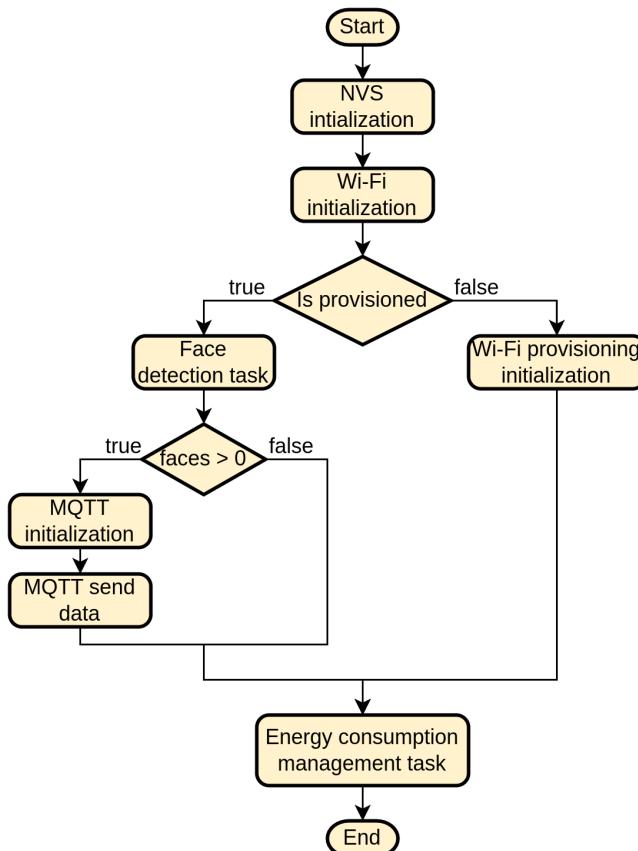


FIGURA 3.9. Diagrama de flujo de la tarea de comunicación con los servicios en la nube.

El diagrama de la figura 3.9 empieza con la inicialización de la memoria NVS y el periférico Wi-Fi en modo punto de acceso y estación a la vez, para después determinar si el dispositivo debe ejecutar el proceso de aprovisionamiento de credenciales Wi-Fi o si debe ejecutar la tarea de detección facial y transmitir los datos generados. Los mensajes se publican en el tópico `faceCounter` y contienen los datos de identificación del dispositivo, cantidad de rostros detectados, estado de la batería y temperatura del dispositivo. El formato de los mensajes a publicar se muestra en el código 3.6.

```

1 {
2     "id": "abddeab5-3428-4129-bbaf-dab22e15d978",
3     "payload": {
4         "faces": 3,
5         "battery": 89,

```

```

6     "temp":24
7 }
8 }
```

CÓDIGO 3.6. Formato de los mensajes a publicar.

Cuando uno más rostros son detectados por la tarea de detección facial, esta información se transmite por MQTT. AWS IoT Core dispone de un *broker* MQTT que implementa TLS [tls_doc] para brindar seguridad en el intercambio de mensajes, por tanto, la autenticación y cifrado de datos necesita de certificados y llaves para llevarse a cabo. La llave privada y el certificado del dispositivo son generados en la plataforma AWS IoT Core y deben quedar grabadas en la memoria del ESP32-S3 para que puedan ser utilizadas en el código. La forma más simple de usar la llave y el certificado es añadirlas al binario de la aplicación durante el proceso de compilación, en la figura 3.10 se observa un diagrama representativo de la distribución de la memoria del ESP32-S3, donde los valores encima de los bloques son las posiciones de memoria donde se encuentran las particiones y los valores que se encuentran por debajo son la cantidad de memoria empleada.

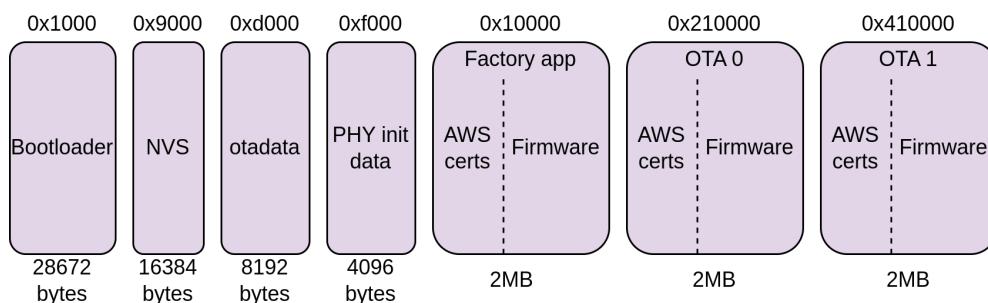


FIGURA 3.10. Diagrama representativo de la memoria del ESP32-S3.

En el diagrama de la figura 3.10 se puede observar un problema no menor con respecto a al proceso de actualizaciones OTA, que las actualizaciones sobreescribirán tanto el *firmware* como el certificado y llave privada del dispositivo. Esto no sería problemático para un solo dispositivo, pero si existieran más dispositivos establecerían conexión con AWS IoT Core con el mismo certificado y llave privada, lo que supondría una grave falla de seguridad de la información. Para corregir esta falla de seguridad se optó por generar llaves y certificados únicos para cada dispositivo, y grabarlos en la memoria NVS en un *namespace* llamado "certs", de esta forma el proceso de actualización OTA solo sobreescribirá el *firmware* más no el certificado ni la llave. En la figura 3.11 se muestra el diagrama representativo de la memoria del ESP32-S3 utilizado en este trabajo.

La inicialización de MQTT en el código utiliza una estructura de datos donde algunos campos deben ser asignados al certificado del dispositivo y la llave privada en formato de cadena de caracteres. En los códigos 3.7 y 3.8 se exhiben la función para obtener una cadena de caracteres de la NVS y la inicialización de MQTT en modo cliente, respectivamente.

```

1 esp_err_t load_string_from_nvs(const char * namespace_name, const char *
2   key, char ** value) {
3   esp_err_t ret = ESP_OK;
4   nvs_handle_t nvs_handle;
5   size_t value_size;
```

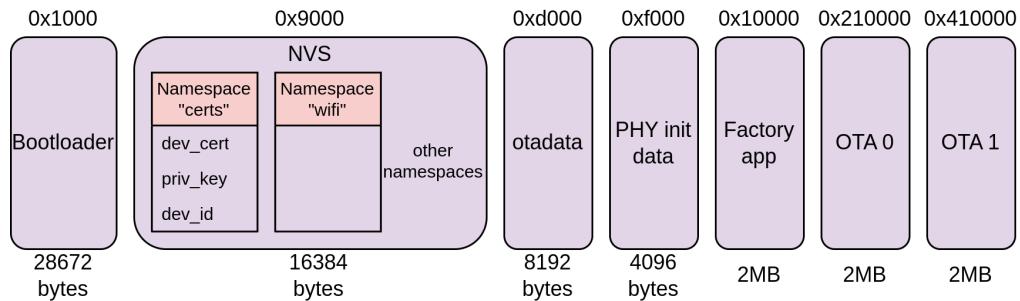


FIGURA 3.11. Diagrama representativo de la memoria del ESP32-S3.

```

6  /* Open the name space to read and write */
7  ret = nvs_open(namespace_name, NVS_READONLY, &nvs_handle);
8
9  if (ret != ESP_OK) {
10    ESP_LOGE(TAG, "Error opening namespace %s", namespace_name);
11    return ret;
12 }
13
14 /* Get value size */
15 ret = nvs_get_str(nvs_handle, key, NULL, &value_size);
16
17 if (ret != ESP_OK){
18    ESP_LOGE(TAG, "Failed to get size of key: %s", key);
19    return ret;
20 }
21
22 /* Allocate memory and get value */
23 *value = (char *)malloc(value_size);
24 ret = nvs_get_str(nvs_handle, key, *value, &value_size);
25
26 if (ret != ESP_OK){
27    ESP_LOGE(TAG, "Failed to load key: %s", key);
28    return ret;
29 }
30
31 /* Close NVS and return */
32 nvs_close(nvs_handle);
33 return ret;
34 }
```

CÓDIGO 3.7. Función para cargar una cadena de caracteres de la NVS.

```

1 /* CA certificate is embedded in the binary application */
2 extern const char amazon_root_ca1_pem_start[] asm(
3     "_binary_amazon_root_ca1_pem_start");
4
5 /* Get device certificate and private key from NVS */
6 char *device_cert = NULL;
7 char *priv_key = NULL;
8
9 load_string_from_nvs("certs", "dev_cert", &dev_cert);
10 load_string_from_nvs("certs", "priv_key", &priv_key);
11
12 /* Fill MQTT client configuration */
13 const esp_mqtt_client_config_t mqtt_config = {
14     .broker = {
15         .address = {
```

```

15     .uri = BROKER_URL, /* Broker address in port 8883 */
16 },
17 .verification = {
18     .certificate = (const char *)amazon_root_ca1_pem_start,
19 },
20 },
21 .credentials = {
22     .authentication = {
23         .certificate = (const char *)dev_cert,
24         .key = (const char *)priv_key
25     },
26 },
27 };
28
29 /* Initialize MQTT client */
30 mqtt_client = esp_mqtt_client_init(&mqtt_config);

```

CÓDIGO 3.8. Código para inicializar MQTT en modo cliente.

3.2.3. Gestión del consumo energético

Esta tarea reduce el consumo promedio de corriente del dispositivo para lograr que pueda ser operado por 2 baterías AA de litio durante un tiempo prolongado. Hace uso del coprocesador ULP del ESP32-S3 para correr un programa que monitorea el estado del sensor de movimiento en un determinado intervalo de tiempo, para determinar si el procesador principal debe ser despertado para ejecutar las tareas de detección facial y comunicación con los servicios en la nube. Para utilizar el coprocesador ULP se debe configurar e inicializar como se muestra en el código 3.9.

```

1 /* Initialize selected GPIO as RTC IO, enable input, disable pullup and
2 pulldown */
3 rtc_gpio_init(PIR_GPIO);
4 rtc_gpio_set_direction(PIR_GPIO, RTC_GPIO_MODE_INPUT_ONLY);
5 rtc_gpio_pulldown_dis(PIR_GPIO);
6 rtc_gpio_pullup_dis(PIR_GPIO);
7 rtc_gpio_hold_en(PIR_GPIO);
8
9 /* Declare the embedded ULP app */
10 extern const uint8_t ulp_main_bin_start[] asm(
11     "_binary_ulp_main_bin_start");
12 extern const uint8_t ulp_main_bin_end[]    asm(" _binary_ulp_main_bin_end ");
13
14 /* Load the ULP application */
15 ulp_riscv_load_binary(ulp_main_bin_start, (ulp_main_bin_end -
16 ulp_main_bin_start));
17
18 /* Configure the timer to wake up de ULP co-processor */
19 ulp_set_wakeup_period(0, 300000); /* 300 ms */
20
21 /* Start the ULP program */
22 ulp_riscv_config_and_run(&cfg);
23
24 /* Enable the wakeup source */
25 esp_sleep_enable_ulp_wakeup();

```

CÓDIGO 3.9. Código para inicializar el coprocesador ULP.

El funcionamiento del coprocesador ULP sigue una secuencia que puede ser explicada con ayuda del diagrama de la figura 3.12.

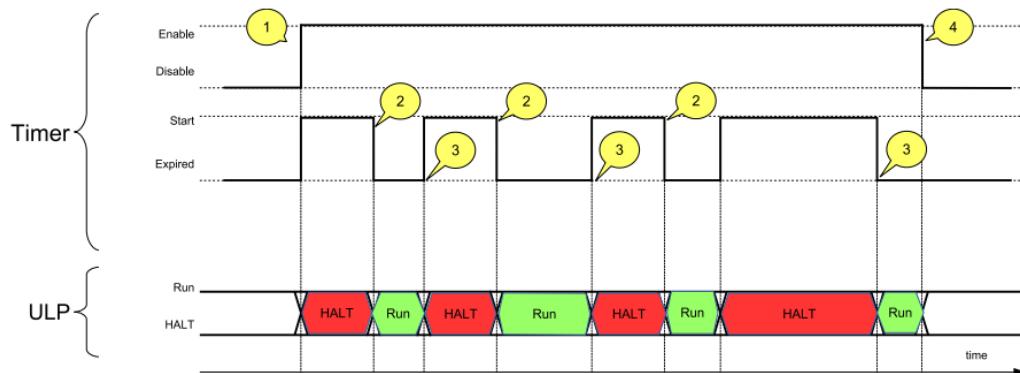


FIGURA 3.12. Diagrama de la secuencia de operación del coprocesador ULP del ESP32-S3³.

1. Se habilita el *timer* y empieza su conteo.
2. El *timer* expira y despierta al coprocesador ULP. El coprocesador ULP empieza la ejecución de su programa.
3. El coprocesador ULP se pone en estado *halt*, es decir, detiene su funcionamiento, y el *timer* comienza su conteo de nuevo.
4. Se deshabilita el *timer* mediante el programa del coprocesador ULP o el programa del procesador principal.

El sensor de movimiento está basado en un diseño de Texas Instruments [pir_ti] que utiliza el TLV8544 para lograr una aplicación de ultra bajo consumo de detección de movimiento humano. El sensor genera un nivel lógico alto en su salida cuando detecta movimiento en un rango de 4 metros y consume aproximadamente $2 \mu\text{A}$. Pero al trabajar con corrientes de polarización tan pequeñas, la salida del sensor tiende a generar señales de muy corta duración o falsos positivos como se muestra en la figura 3.13

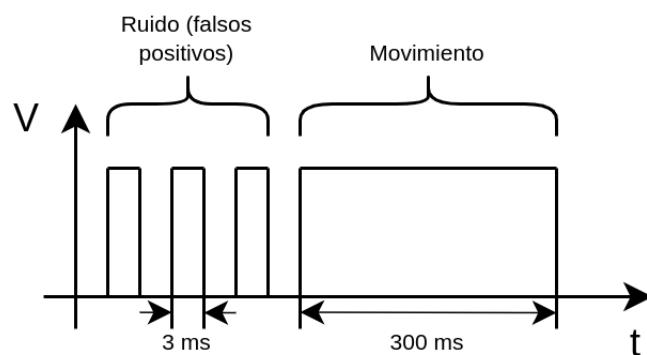


FIGURA 3.13. Diagrama de la secuencia de operación del coprocesador ULP del ESP32-S3.

³Imagen tomada de: https://www.espressif.com/sites/default/files/documentation/esp32-s3_technical_reference_manual_en.pdf

Para evitar que el procesador principal se active por la aparición de falsos positivos generados por el sensor de movimiento, se diseñó una máquina de estados que corre en el coprocesador ULP. Esta máquina de estados tiene la función de filtrar todas las señales de nivel lógico alto con una duración menor a 300 ms y está basada en el diagrama de la figura 3.14.

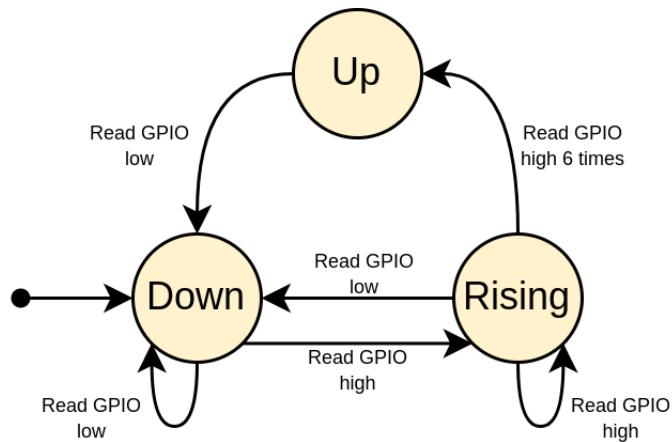


FIGURA 3.14. Máquina de estados para filtrar falsos positivos del sensor de movimiento.

Con todas las técnicas de bajo consumo aplicadas hasta ahora, el funcionamiento del dispositivo se puede describir como la transición entre todos sus modos de consumo, tal como se muestra en la figura 3.15.

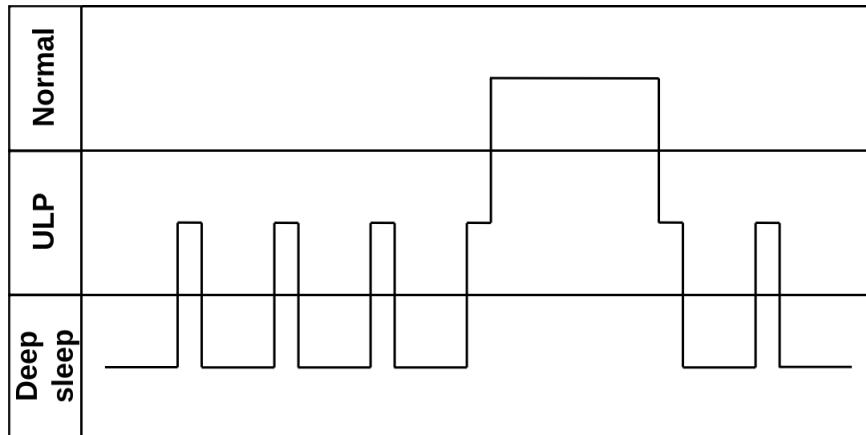


FIGURA 3.15. Diagrama de transición de modos de consumo del dispositivo.

La mayor parte del tiempo de funcionamiento el dispositivo se encontrará entre los modos *deep sleep* y ULP, hasta que el sensor de movimiento genere una señal de suficiente duración y se cambie al estado normal, donde se ejecutan las tareas de detección facial y comunicación con los servicios en la nube. En la tabla 3.2 se muestran los valores de corriente utilizados por el dispositivo en cada uno de los estados de funcionamiento.

TABLA 3.2. Consumo de corriente aproximado de todos los modos del dispositivo

Estado	Consumo (mA)
Normal	400
ULP	0.2
<i>Deep sleep</i>	0.008

3.3. Procesamiento y visualización en la nube

Otro aspecto de importancia en este trabajo fue la integración del sistema embedido con los servicios en la nube responsables de procesar los datos entrantes y mostrarlos en un *dashboard* adecuado para que los usuarios finales puedan visualizar la actividad de los dispositivos. En la figura 3.16 se puede observar la arquitectura de los servicios en la nube.

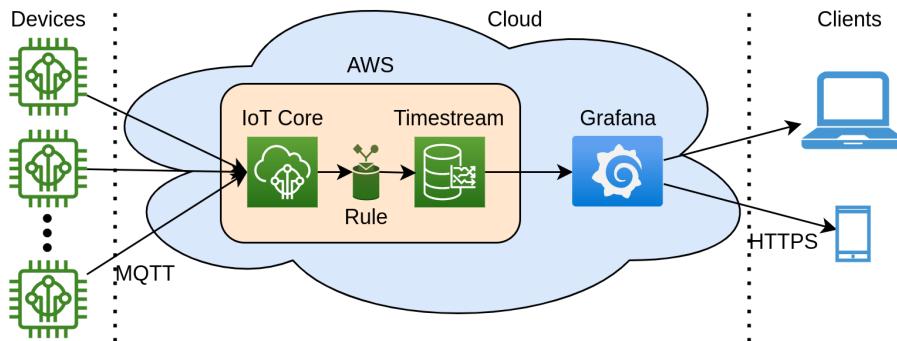


FIGURA 3.16. Arquitectura de los servicios en la nube.

De la figura 3.16 se infiere que los datos generados por los dispositivos son transmitidos por MQTT hacia IoT Core, donde mediante un *rule* se extrae la información relevante y se guarda en una base de datos de Timestream. Los datos de Timestream son utilizados para crear un *dashboard* en Grafana, que puede ser accedido por los clientes a través de un navegador web y el protocolo HTTPS.

3.3.1. Gestión de dispositivos con IoT Core

Para que los dispositivos se conecten a IoT Core para recibir y transmitir mensajes deben ser previamente registrados. El proceso de registro consistió de los siguientes pasos:

1. Crear un *thing*: un *thing* representa el dispositivo físico o virtual que se desea registrar. Se puede crear manualmente a través de la consola de administración de AWS.
2. Generar certificados y llaves: para habilitar la comunicación segura entre el dispositivo y IoT Core, se deben generar certificados y llaves de seguridad.
3. Descargar certificados y claves: una vez generados los certificados y llaves, IoT Core proporcionará los archivos necesarios para autenticar y establecer

una conexión segura entre el dispositivo y el servicio. Estos archivos incluyen un certificado X.509 [**x509_info**] del dispositivo, una llave privada y un certificado de autoridad raíz.

4. Adjuntar políticas de acceso al certificado: el certificado del dispositivo debe ser adjuntado a una determinada política de acceso para que el dispositivo pueda interactuar con los servicios de AWS requeridos.

Todos estos pasos pueden realizarse manualmente a través de la consola web de AWS, pero no resulta un proceso adecuado cuando se desea registrar muchos dispositivos. En cambio, para este trabajo se creó un *script* que utiliza la interface por línea de comandos AWS (aws-cli) [**awscli_info**] para automatizar esta tarea. En el código 3.10 se puede observar el código de este *script*.

```

1 #!/bin/bash
2
3 # Create thing, device certificate and private key
4 AWS_IOT_ENDPOINT="xxxxxxxxxxxxxx-ats.iot.us-east-1.amazonaws.com"
5 THING_NAME=$(uuidgen)
6
7 aws iot create-thing --thing-name $THING_NAME
8
9 CERT_ARN=$(aws iot create-keys-and-certificate --set-as-active --
10   certificate-pem-outfile device.pem.crt --private-key-outfile private
11   .pem.key --query 'certificateArn' --output text)
12
13 # Attach certificate to thing and policy to certificate
14 aws iot attach-thing-principal --thing-name $THING_NAME --principal
15   $CERT_ARN
16 aws iot attach-principal-policy --principal $THING_NAME --principal
17   $CERT_ARN --policy-name IOT_ACCESS
18
19 # Update the NVS settings file
20 NVS_CONFIG="nvs.csv"
21
22 awk -v new_value="$THING_NAME" -F ',' '
23 BEGIN { OFS = FS }
24 {
25   if ($1 == "device_id") {
26     $4 = new_value
27   }
28   print
29 }
30 '$NVS_CONFIG' > "${NVS_CONFIG}.tmp" && mv "${NVS_CONFIG}.tmp" "$NVS_CONFIG"
```

CÓDIGO 3.10. *Script* para automatizar el registro de un dispositivo en AWS IoT Core

La política de acceso asociada al certificado del dispositivo le da permisos para conectarse, publicar, suscribirse y recibir. En el código 3.11 se muestra la política de acceso en formato JSON.

```

1 {
2   "Version": "2012-10-17",
3   "Statement": [
4     {
5       "Effect": "Allow",
6       "Action": [
7         "iot:Connect",
8         "iot:Publish",
9         "iot:Subscribe",
```

```

10      "iot:Receive"
11  ],
12  "Resource": [
13    "*"
14  ]
15 ]
16 }
17 }
```

CÓDIGO 3.11. Política de acceso asociada a los certificados de los dispositivos.

Para grabar en la memoria del dispositivo los certificados y llaves generados anteriormente, además de otros datos de importancia, se utiliza el generador de particiones NVS incluido en ESP-IDF. Este requiere un archivo de tipo .csv de donde obtiene toda la información que servirá para generar un archivo binario que deberá ser grabado en la región de memoria correspondiente a la partición NVS. En los códigos 3.12 y 3.13 se muestran el archivo nvs.csv y el comando para generar el archivo binario de la partición, respectivamente.

```

1 key , type , encoding , value
2 certs , namespace ,
3 dev_id , data , string , abddeab5-3428-4129-bbaf-dab22e15d978
4 dev_cert , file , string , certificate .pem. crt
5 priv_key , file , string , private .pem. key
```

CÓDIGO 3.12. Archivo nvs.csv.

```
1 nvs_partition_gen.py generate "nvs.csv" nvs.bin 16384
```

CÓDIGO 3.13. Comando para crear una particion con el generador de particiones NVS

3.3.2. Bases de datos de series temporales con Timestream

Para almacenar los datos de relevancia que los dispositivos conectados a IoT Core pueden generar, se utilizó Timestream, de esta forma pueden ser accedidos por otros servicios para su posterior procesamiento y visualización. Los pasos que se siguieron para implementar Timestream fueron:

1. Crear una base de datos de Timestream: se creó una base de datos con el nombre `iot` con las opciones que se muestran en la captura de pantalla de la figura 3.17
2. Crear una tabla: dentro de la base de datos `iot`, se creó una tabla de nombre `faceCounter` para organizar los datos de series temporales de los dispositivos. En la figura 3.18 se muestran las opciones para la creación de la tabla `faceCounter`.
3. Ingesta de datos: para enviar datos a Timestream se creó un *rule* en IoT Core que redirige los datos relevantes contenidos en los mensajes que llegan al broker MQTT hacia la tabla `faceCounter` de la base de datos `iot`. En el código 3.14 se muestra el código SQL necesario para extraer los datos que debe ser almacenados en la base de datos.

```
1 SELECT payload.face AS face , payload.battery AS battery , payload.
temperature AS temperature FROM 'faceCounter/data_in'
```

CÓDIGO 3.14. Código SQL del *rule* para alcenar datos en Timestream.

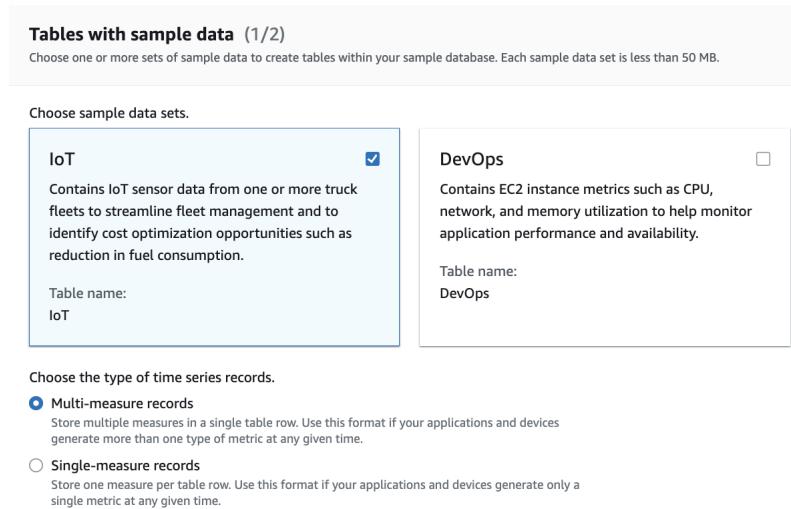


FIGURA 3.17. Captura de pantalla de la creación de una base de datos en Timestream.

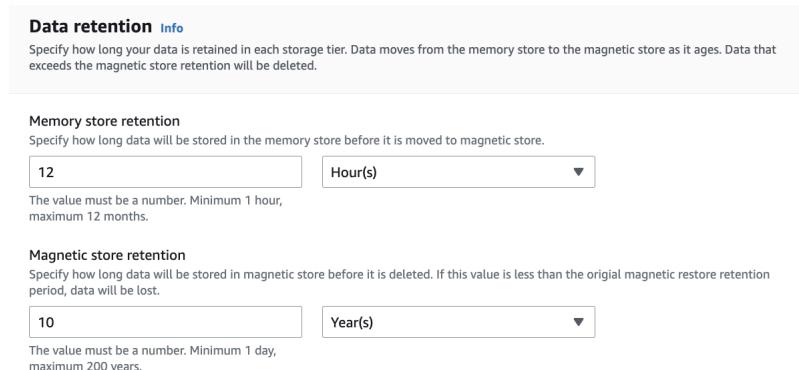


FIGURA 3.18. Captura de pantalla de la creación de una tabla en Timestream.

3.3.3. Visualización de datos con Grafana

Una vez creada la base de datos de series temporales para almacenar los datos del dispositivo, puede ser utilizada para ofrecer sus datos a otros servicios, en este caso Grafana. Grafana ofrece integración con Timestream a través del uso de *plugins* y credenciales de acceso a AWS, donde los datos almacenados pueden ser accedidos con el uso del lenguaje SQL. Para visualizar datos con Grafana se deben generar *dashboards* que, a través de elementos de visualización que resultan comprensibles para los usuarios finales. El *dashboard* creado está compuesto por 2 elementos de visualización: 1 gráfico de series de tiempo para los datos de rostros detectados, junto con 2 gráficos indicadores para los datos de estado de la batería y temperatura del dispositivo. En la figura 3.19 se muestra el *dashboard* creado para este trabajo, mientras que en el código 3.15 se encuentra el código SQL para obtener los datos sobre los rostros detectados de Timestream.

```

1 SELECT CREATE_TIME_SERIES(time, measure_value :: bigint) as Face
2 FROM "iot"."mse_test_table" WHERE $__timeFilter
3   and measure_name = 'face'
4   and device_id = '${id}'
```

CÓDIGO 3.15. Código SQL para obtener datos de Tiemstream en Grafana.



FIGURA 3.19. Captura de pantalla del *dashboard*.

Capítulo 4

Ensayos y resultados

En este capítulo se presentan las pruebas realizadas sobre el prototipo de pruebas del sistema. Se detallan los procedimientos para probar los modelos para detección facial, el sensor de movimiento, el consumo energético del sistema y los servicios en la nube empleados.

4.1. Banco de pruebas

Para llevar a cabo pruebas y mediciones precisas sobre el sistema, fue necesario montar un conjunto de herramientas e instrumentos para evaluar su funcionamiento. El banco de pruebas utilizado para este trabajo es el que se muestra en la figura 4.1.

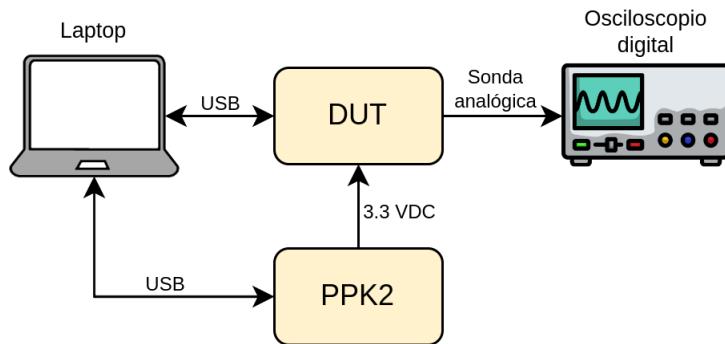


FIGURA 4.1. Diagrama del banco de pruebas.

Los componentes que conforman el banco de pruebas según la figura 4.1 son:

- Laptop: cumple la función de ejecutar el software necesario para ejecutar las pruebas, monitorear y controlar todos los periféricos conectados mediante USB.
- Osciloscopio digital: TDS200C de Tektronix, es el instrumento utilizado para visualizar y medir las señales eléctricas generadas por el sistema. Su función principal para este trabajo fue evaluar las señales generadas por el sensor de movimiento.
- PPK2: Power Profiler Kit 2 de Texas Instruments, es un *datalogger* enfocado en la medición de corrientes muy pequeñas y sirvió para evaluar el consumo de corriente consumido por el dispositivo.
- DUT: es el prototipo de pruebas en sí, sobre este se realizan todas las pruebas y mediciones con todos los herramientas del banco de pruebas.

4.2. Pruebas sobre los modelos

Estas pruebas tuvieron el objetivo de ensayar el *pipeline* donde se encuentran los modelos de MTCNN para detección facial obtenidos para TensorFlow, TensorFlow Lite sin cuantización y TensorFlow Lite con cuantización a 8 bits y TensorFlow Lite Micro con cuantización a 8 bits. La figura 4.2 presenta una imagen que contiene 3 rostros de formato RGB888 y dimensiones 96x96 píxeles, que fue utilizada como entrada para los *pipelines* a probar.

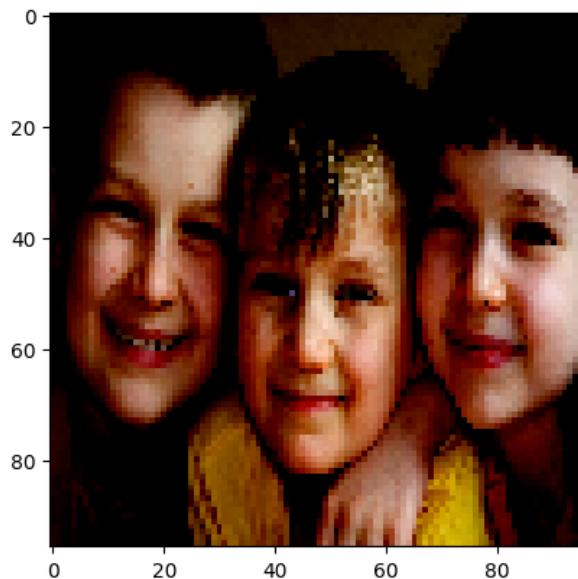


FIGURA 4.2. Imagen de prueba para los modelos.

Para probar los modelos para TensorFlow, TensorFlow Lite sin cuantización y TensorFlow Lite con cuantización a 8 bits, se utilizó Google Colab en conjunto con el *framework* TensorFlow en lenguajes Python y varias bibliotecas para visualización de imágenes. En las figuras 4.3, 4.4 y 4.5 se muestran los resultados obtenidos.

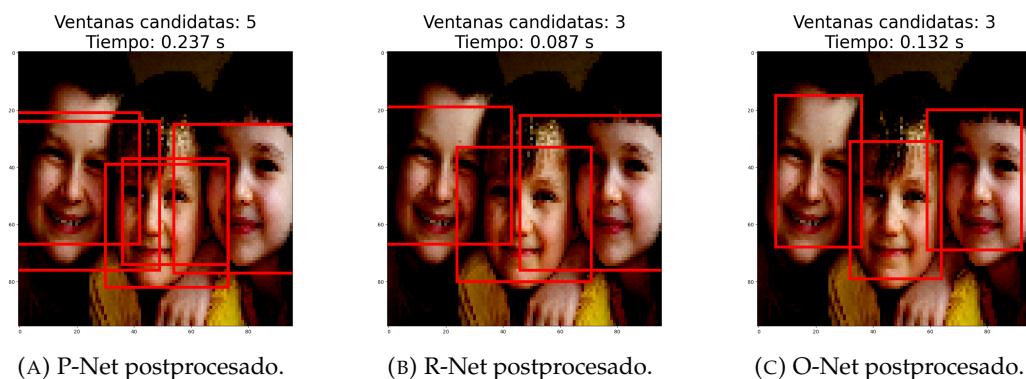


FIGURA 4.3. Resultados para TensorFlow.

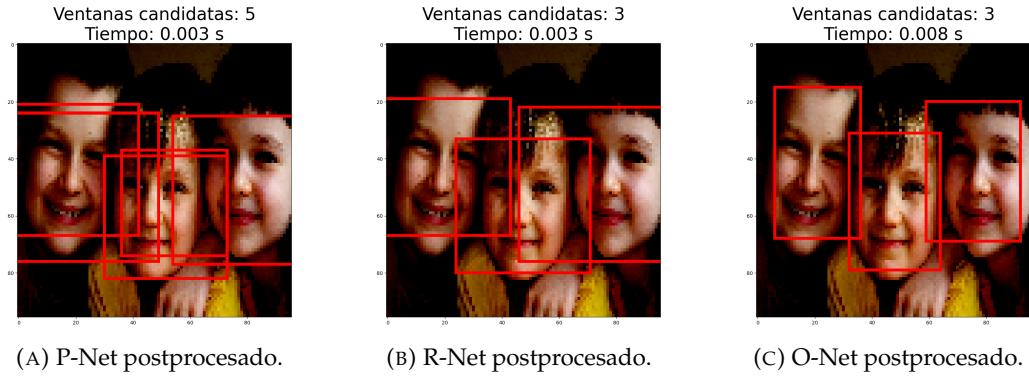


FIGURA 4.4. Resultados para TensorFlow Lite sin cuantización.

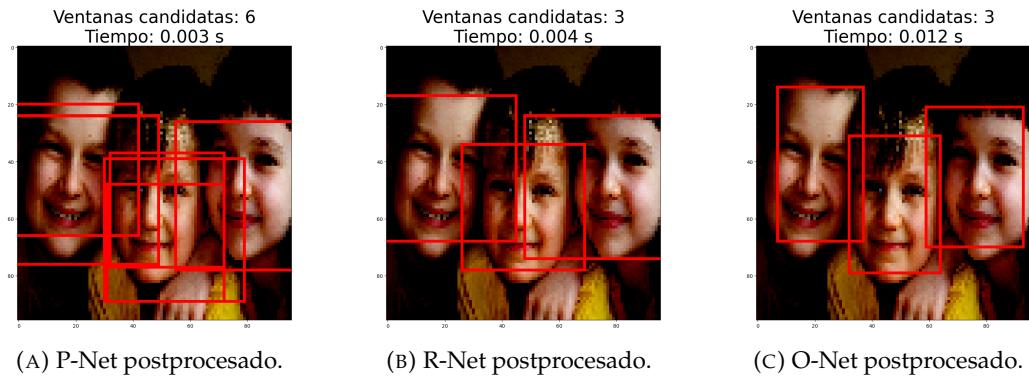


FIGURA 4.5. Resultados para TensorFlow Lite con cuantización a 8 bits.

Para probar el *pipeline* y los modelos de MTCNN para TensorFlow Lite Micro con cuantización a 8 bits, se modificó el *firmware* del prototipo de pruebas para que la imagen de prueba esté embebida en el binario de la aplicación y pueda ser leída dentro del programa. En la figura 4.6 se observan los resultados de esta prueba.

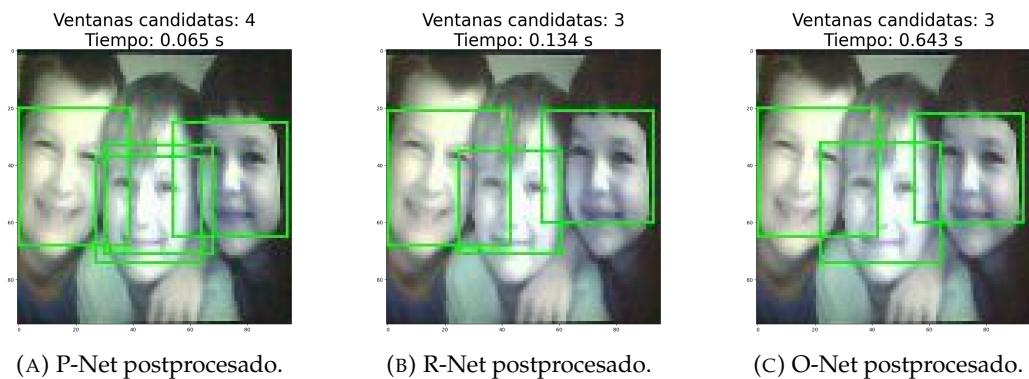


FIGURA 4.6. Resultados para TensorFlow Lite Micro con cuantización a 8 bits.

Los resultados obtenidos de las pruebas en Google Colab fueron los esperados. A medida que los modelos se iban optimizando en tamaño y tiempo de respuesta, la precisión de los resultados se reducía. Para las pruebas realizadas sobre el prototipo de pruebas, los resultados también estuvieron acordes a lo planeado, si bien el tamaño es el más pequeño posible, los tiempos de inferencia son mucho más

altos que para los modelos probados en Google Colab, esto por las limitaciones de hardware del ESP32-S3, aun así los resultados finales son muy similares. En la tabla 4.1 se exponen los resultados para todas las pruebas sobre los modelos.

TABLA 4.1. Resultados de las pruebas sobre los modelos

Modelo	Rostros encontrados	Tiempo de ejecución MTCNN (ms)
TF	3	456
TF Lite	3	14
TF Lite 8 bits	3	19
TF Lite Micro 8 bits	3	842

4.3. Pruebas sobre el sensor de movimiento

Las pruebas realizadas sobre el sensor de movimiento tuvieron el objetivo de determinar el comportamiento de las señales de salida generadas por el TLV8544 cuando se genera movimiento en el rango de acción del sensor PIR. Para este fin se conectaron las sondas del osciloscopio en la salida de la señal analógica y en las 2 salidas digitales de los comparadores. En la figura 4.7 se observan las señales capturadas por el osciloscopio para 5 movimientos detectados por el sensor PIR.

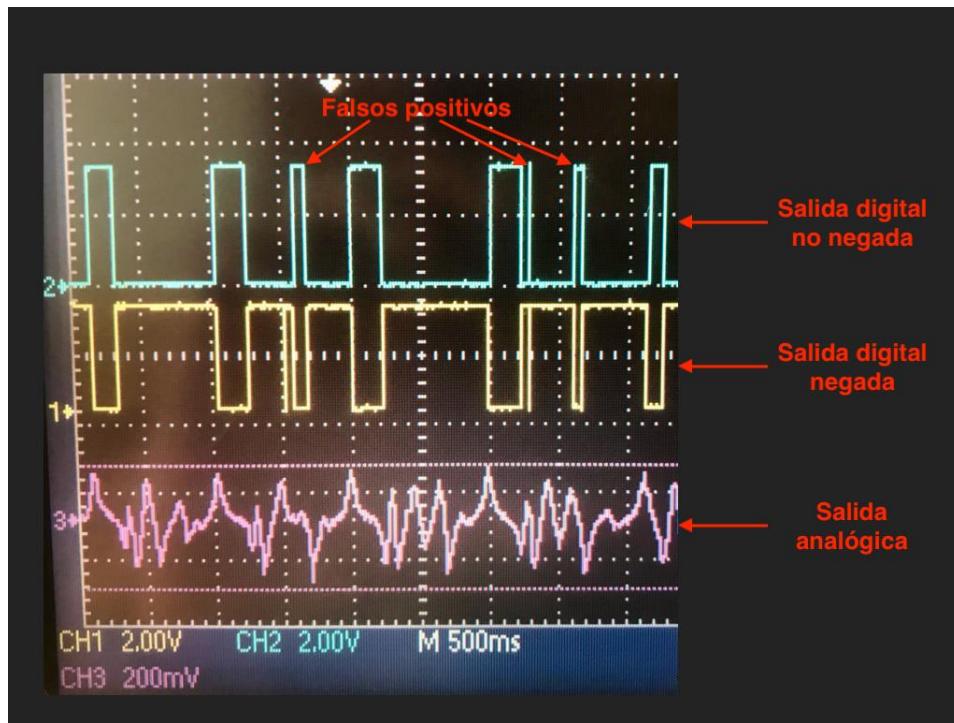


FIGURA 4.7. Señales de salida del sensor de movimiento.

Como se puede observar de la figura 4.7 se generaron señales de corta duración que causaron falsos positivos en la detección de movimiento. Este efecto fue subsanado con la ayuda de una máquina de estados implementada en el programa que corre el coprocesador ULP.

4.4. Pruebas de consumo energético sobre el sistema

Estas pruebas consistieron en poner en funcionamiento el prototipo de pruebas y medir el consumo de corriente para determinar si todas las técnicas de bajo consumo fueron aplicadas correctamente. Como se mostró en la figura 3.15 el sistema durante su funcionamiento realiza transiciones entre 3 estados distintos, donde cada estado tiene un consumo de corriente diferente. Con ayuda del PPK2 y del software nRF Connect for Desktop con su módulo Power Profile, se midió el consumo de corriente del sistema durante un tiempo de 5 minutos, donde fue activado en varias ocasiones mediante el sensor de movimiento. En la figura 4.9 se observa un gráfico donde se puede apreciar el comportamiento del sistema a través de la forma en como cambia su consumo de corriente con relación al tiempo.

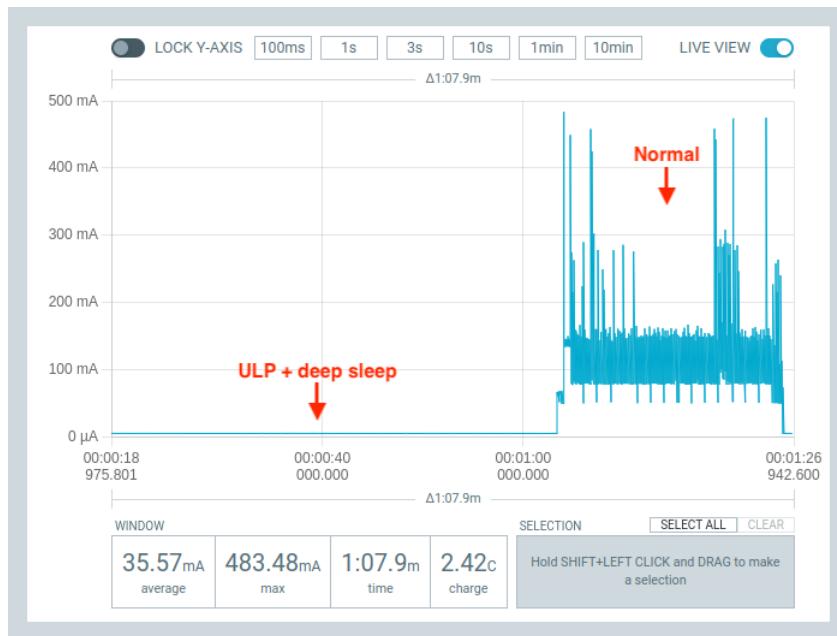


FIGURA 4.8. Gráfico de consumo de corriente para un ciclo de funcionamiento.

La figura 4.8 muestra el 1 ciclo de trabajo normal, es decir, que se mantiene en modo de bajo consumo hasta que detecta un movimiento válido mediante el sensor de movimiento y activa el procesador principal para realizar las tareas de detección facial y comunicación con los servicios en la nube. Cuando el dispositivo se encuentra en modo de bajo consumo cambia entre los estados Deep sleep y ULP cada 100 ms, donde en el estado ULP se ejecuta la máquina de estados que mitiga los errores causados por los falsos positivos generados por el sensor de movimiento. En la figura 4.9 se exhibe con mayor detalle el funcionamiento cuando el dispositivo se encuentra en modo de bajo consumo.



FIGURA 4.9. Gráfico de consumo de corriente en modo de bajo consumo.

En la tabla 4.2 se detallan los consumos de corriente y los tiempos de duración de todos los estados de consumo del dispositivo.

TABLA 4.2. Consumo de corriente aproximado de todos los modos del dispositivo

Estado	Consumo promedio (mA)	Consumo máximo (mA)	Tiempo (s)
Normal	97.31	478.21	22.8
ULP	0.247	0.310	0.003
Deep sleep	0.112	0.121	0.1

El evento que despierta al procesador principal para ejecutar las tareas de detección facial y comunicación con los servicios en la nube es asíncrono, es decir, que no es posible predecirlo y puede ocurrir muchas veces o muy pocas, por lo que no es posible medir una corriente promedio del dispositivo en general y por tanto tampoco se pudo establecer el tiempo que un par de baterías de litio AA podría alimentar el dispositivo.

4.5. Pruebas sobre los servicios en la nube

Las pruebas sobre los servicios en la nube se basaron en simular la conexión y publicación de mensajes de 3 dispositivos distintos por un lapso de 10 minutos. La simulación de los dispositivos se hizo con ayuda del *script* de Python mostrado en 4.1, que tiene la función de conectarse al broker de IoT Core y publicar 15 mensajes en un lapso 15 minutos para cada uno de los dispositivos a simular.

```

1 from random import random
2 from awscrt import mqtt
3 from awsiot import mqtt_connection_builder
4 import time as t
5 import random

```

```

6
7 ENDPOINT = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx.amazonaws.com"
8 CLIENT_ID = "dev1"
9 PATH_TO_CERTIFICATE = "dev_cert.pem.crt"
10 PATH_TO_PRIVATE_KEY = "priv_key.pem.key"
11 PATH_TO_AMAZON_ROOT_CA_1 = "AmazonRootCA1.pem"
12 TOPIC = "faceCounter"
13
14 mqtt_connection = mqtt_connection_builder.mtls_from_path(endpoint=
    ENDPOINT, cert_filepath=PATH_TO_CERTIFICATE, pri_key_filepath=
    PATH_TO_PRIVATE_KEY, ca_filepath=PATH_TO_AMAZON_ROOT_CA_1, client_id=
    =CLIENT_ID, clean_session=False, keep_alive_secs=6)
15
16 # Device IDs
17 id = ["7a5b4c79-621d-476d-83b4-861005589752", "8a5b4c79-621d-476d-83b4-
    -861005589752", "9a5b4c79-621d-476d-83b4-861005589752"]
18
19 # Connect to broker
20 connect_future = mqtt_connection.connect()
21 connect_future.result()
22
23 # Send messages for all IDs
24 for i in range(30):
25     for j in id:
26         # Set new values for the JSON message
27         message = "{\n\"id\":\"%s\",\\n\"payload\":{\"\\n\"faces\":%d,\\n\"-
            battery\":%d,\\n\"temperature\":%d\\n}\\n}" % (j, random.randint(0, 5),
            random.randint(50, 90), random.randint(19, 27))
28         mqtt_connection.publish(topic=TOPIC, payload=message, qos=mqtt.QoS.AT_LEAST_ONCE)
29     # Wait for 1 min
30     time.sleep(60)
31
32 # Disconnect from broker
33 disconnect_future = mqtt_connection.disconnect()
34 disconnect_future.result()

```

CÓDIGO 4.1. Código del *script* para probar IoT Core.

Para controlar los mensajes que son recibidos por el *broker* se utilizó el monitor MQTT de IoT Core MQTT Test, que consiste en un cliente que puede publicar y suscribirse a uno o varios tópicos. En la figura 4.10 se observa un mensaje publicado por el *script* del código 4.1 en el tópico `faceCounter`.



FIGURA 4.10. Mensaje de prueba recibido en MQTT Test Client.

Los mensajes publicados en `faceCounter` fueron procesados mediante un `rule` para grabar los datos de relevancia en la tabla `faceCounter` de la base de datos `iot` de Timestream. El código 4.2 consulta los datos de los últimos 5 minutos de la tabla `faceCounter` y en la figura 4.11 se muestran los resultados obtenidos.

```
1 SELECT * FROM "iot"."faceCounter" WHERE time between ago(5m) and now()
    ORDER BY time
```

CÓDIGO 4.2. Código SQL para obtener los datos de la tabla `faceCounter`.

<code>id</code>	<code>measure_name</code>	<code>time</code>	<code>measure_value::bigint</code>
7a5b4c79-621d-476d-83b4-861005589752	battery	2023-05-20 21:06:46.810000000	55
7a5b4c79-621d-476d-83b4-861005589752	faces	2023-05-20 21:06:46.810000000	5
7a5b4c79-621d-476d-83b4-861005589752	temperature	2023-05-20 21:06:46.810000000	22
8a5b4c79-621d-476d-83b4-861005589752	battery	2023-05-20 21:06:46.918000000	80
8a5b4c79-621d-476d-83b4-861005589752	temperature	2023-05-20 21:06:46.918000000	19
8a5b4c79-621d-476d-83b4-861005589752	faces	2023-05-20 21:06:46.918000000	2
9a5b4c79-621d-476d-83b4-861005589752	battery	2023-05-20 21:06:47.055000000	56
9a5b4c79-621d-476d-83b4-861005589752	temperature	2023-05-20 21:06:47.055000000	21
9a5b4c79-621d-476d-83b4-861005589752	faces	2023-05-20 21:06:47.055000000	4

FIGURA 4.11. Tabla `faceCounter` con los datos de prueba.

Finalmente, en la figura 4.12 se exhibe el *dashboard* creado con Grafana, donde cada columna representa mediante gráficos los datos de uno de los dispositivos simulados.

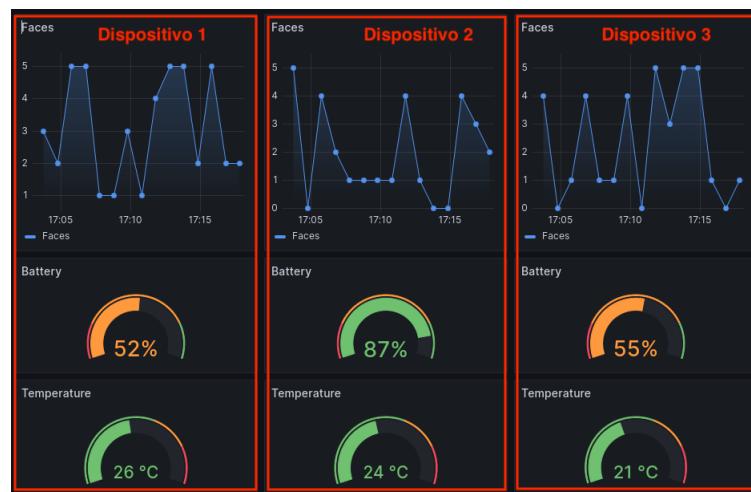


FIGURA 4.12. *Dashboard* con datos de prueba para 3 dispositivos.

Capítulo 5

Conclusiones

5.1. Conclusiones generales

En este trabajo se logró diseñar e implementar el prototipo de pruebas de un dispositivo con la capacidad de ejecutar MTCNN para cumplir con la tarea de detección facial y transmitir esta información hacia los servidores de AWS. También se desarrolló un *dashboard* para visualizar la información del dispositivo mediante gráficos que facilitaron su comprensión.

Cabe destacar que este trabajo está implementado sobre un hardware con muchas limitaciones en capacidad de memoria y poder computo, pero que cumplió de muy buena manera los objetivos planteados. Otro punto importante del hardware fue su costo, que no superó los 15 \$us americanos y queda muy por debajo de dispositivos comerciales que cumplen funciones similares.

La implementación de los algoritmos de DL fue una tarea que requirió mucho más tiempo y esfuerzo que el planificado. Fue necesario aprender conceptos sobre procesamiento de imágenes y visión artificial, para esto se realizaron varios cursos y se encararon proyectos más pequeños que sirvieron como punto de entrada para este. Uno de los cursos más útiles fue "Bootcamp: Visión Artificial para los ODS" de la organización Hackcities que tuvo una duración de 4 meses.

Otro aspecto que retrasó el trabajo, aunque en menor medida, fue el despliegue de los servicios en la nube de AWS. Para utilizarlos fue imprescindible adquirir conocimientos sobre bases de datos y lenguaje SQL, que también fueron útiles al momento de implementar el *dashboard* en Grafana.

Casi la totalidad de los requerimientos funcionales y no funcionales del trabajo fueron cumplidos exitosamente. Solamente el requerimiento para implementar los mecanismos de seguridad en hardware y firmware no pudo ser cumplido. Los mecanismos de seguridad de firmware disponibles en el ESP32-S3 son la encriptación de los datos en la memoria *flash* y la verificación de la autenticidad de las aplicaciones firmadas digitalmente grabadas en la memoria. Estos mecanismos para funcionar necesitan interactuar con los efuses, que son secciones de memoria no volátil y que solo pueden ser modificadas una sola vez, para determinar la configuración del ESP32-S3. Cuando estas características de seguridad son habilitadas, el proceso de compilación y grabado del binario en la memoria demora aproximadamente 3 veces más. Por tanto, si bien podía haberse implementado el requerimiento faltante, esto hubiera retrasado mucho el proceso de desarrollo y pruebas.

5.2. Próximos pasos

Esta memoria describe el proceso de diseño e implementación del prototipo de pruebas, que fue utilizado para comprobar la factibilidad técnica de todos los requerimientos funcionales planteados. Los siguientes pasos a nivel de hardware serían los siguientes:

1. Incorporar componentes de hardware para controlar y monitorear el suministro de energía de las baterías.
2. Diseñar el diagrama esquemático.
3. Seleccionar una carcasa adecuada al tamaño y entorno de aplicación del dispositivo.
4. Diseñar un PCB cuyas dimensiones se correspondan con las de la carcasa.

Si bien la tarea de detección facial se ejecutó de manera exitosa mediante el uso de MTCNN, esta puede ser empleada como punto de partida para lograr aplicaciones más interesante para lograr un producto comercial atractivo para el mercado. Los pasos a seguir para optimizar y aumentar las funcionalidades de los modelos de AI serían:

1. Estudiar la factibilidad de utilizar FaceNet en el dispositivo para lograr que en conjunto con MTCNN realicen reconocimiento facial, reconocimiento de edad y género, o reconocimiento de emociones.
2. Estudiar la factibilidad de usar Amazon Rekognition para ejecutar reconocimiento facial en la nube.