Universidad Zaragoza
1542

Trabajo de Fin de Grado

Algoritmos de aprendizaje profundo en sistemas empotrados

Deep learning on embedded systems

Autor/es

Enrique Phan Razquin

Director/es

Bonifacio Martín del Brío

Grado en ingeniería electrónica y automática

Departamento de Ingeniería electrónica y comunicaciones
Escuela de ingeniería y arquitectura de Zaragoza

Junio 2021

Resumen

Este documento explora el desarrollo de aplicaciones basadas en redes neuronales y aprendizaje profundo en sistemas empotrados implementadas en microcontroladores de prestaciones reducidas. Este área recientemente se ha comenzado a denominar *TinyML*. Estos dispositivos presentan costes y consumos bajos, por lo que los hace ideales para aplicaciones portables y a gran escala.

En un primer lugar se explican las ventajas y desafíos a los que se debe enfrentar un desarrollador a la hora de realizar un proyecto en esta área. Más tarde, se procede al desarrollo de principio a fin de dos aplicaciones de distintos ámbitos, ambas procesando datos de sensores en tiempo real, con el fin de analizar las potentes herramientas hardware y software actualmente disponibles y proponer una metodología de trabajo

La primera aplicación, orientada a visión por computador, consiste en clasificación multiclase de imágenes de personas y/o coches para aplicarlo en sistemas de control de tráfico, además reduciendo consumos de gasolina y emisiones de CO2. El desarrollo se enfoca en el flujo de adquisición de datos y la selección de modelo para conseguir rendimientos profesionales y, al mismo tiempo, ajustados a las modestas especificaciones de un microcontrolador. Se lleva a cabo también un prototipo funcional que sirve como prueba de concepto y se analizan sus prestaciones, pudiéndose destacar que el microcontrolador procesa y clasifica la imagen en un segundo aproximadamente.

La segunda aplicación consiste en fijar un pequeño acelerómetro de tres ejes a la culata de un bolígrafo. La red neuronal analiza los movimientos (aceleraciones) del bolígrafo en tiempo real y transcribe las letras que se vayan escribiendo. La aplicación se enfoca en la confección del conjunto de datos (manual), así como de la implementación optimizada en un sistema operativo de tiempo real que envía los resultados por *bluetooth* de bajo consumo (*BLE*) a un servidor. También se lleva a cabo un prototipo funcional, observándose que un microcontrolador típico tarda unos 65 ms en clasificar y transcribir el trazo a la letra correspondiente.

En definitiva, hemos comprobado que la implementación de algoritmos avanzados basados en redes neuronales y aprendizaje profundo en sistemas empotrados basados en microcontroladores de prestaciones reducidas es factible y que aporta numerosas ventajas frente a sistemas tradicionales, demostrándolo con la elaboración de dos prototipos funcionales. Como conclusión fundamental podemos decir que en la actualidad las herramientas disponibles analizadas han alcanzado ya cierto grado de madurez, lo que permite automatizar en parte el desarrollo de aplicaciones de *machine* y *deep learning* en dispositivos empotrados, facilitando el desarrollo de aplicaciones reales.

Índice

1. Introducción

1.1 Motivación y objetivos del proyecto

En el ámbito de la inteligencia artificial, las técnicas de aprendizaje automático (*machine learning)* y, en concreto, las de aprendizaje profundo (*deep learning)* han tenido un gran auge en popularidad y uso. Las aplicaciones se extienden a lo largo de multitud de campos y problemas, como clasificación de imágenes, conducción autónoma, sistemas de recomendación, detectores de fraude, asistentes personales, restauración de videos, segmentación de escáneres médicos y un largo etc.

Muchas de estas aplicaciones hacen uso de redes neuronales, un modelo matemático parametrizable compuesto de pequeñas unidades lógicas conectadas entre sí. Dichos parámetros se ajustan estadísticamente mediante algoritmos de optimización para conseguir que el modelo se ajuste a la función deseada. Es decir, a diferencia de los algoritmos clásicos, donde el desarrollador crea todas las reglas mediante secuencias lógicas de forma manual, los algoritmos de aprendizaje automático utilizan ejemplos para ajustar sus parámetros y no requieren de una programación manual. Se puede decir que *aprenden* de la *experiencia* y, según su cantidad de unidades lógicas, serán capaces de resolver problemas más simples o más complejos. Esta última característica se denomina *capacidad* del modelo.

Estos algoritmos han cobrado especial relevancia a partir del año 2009 [1], pues la optimización de estos modelos se basa en métodos iterativos con una gran demanda de computación. Este proceso es denominado *entrenamiento* de la red neuronal. Sin embargo, una vez entrenados, estos modelos se pueden ejecutar (*realizar inferencia*) normalmente de forma rápida en ordenadores de gama media, smartphones o incluso pequeños sistemas electrónicos como microcontroladores.

Además, el desarrollo de técnicas que permiten realizar inferencia con menor coste computacional y memoria [2] han permitido realizar aplicaciones que usan aprendizaje profundo en tiempo real en pequeños dispositivos como microcontroladores, acuñando el término *TinyML* (*machine learning* "diminuto") [27]. Estos dispositivos tienen unas prestaciones generalmente muy modestas, con frecuencias de reloj alrededor de los 100 MHz y memoria y almacenamiento en la escala de unos cientos de *kilobytes* o unos pocos *megabytes.* Su coste suele ir entre un par de euros los de gama media y hasta diez euros los de gama alta.

La motivación de este proyecto se puede dividir en dos partes. Por un lado se va a mostrar cómo los algoritmos de aprendizaje profundo se pueden utilizar para resolver problemas reales y en aplicaciones muy diversas y, por otro lado, comprobar que el despliegue de este tipo de algoritmos en microcontroladores y su ejecución en tiempo real es posible y adecuado para aplicaciones industriales.

Partiendo de estas ideas, los objetivos principales de este proyecto son:

- Entender qué problemas se pueden resolver con aprendizaje automático y por qué se deberían resolver así.
- Ilustrar la metodología de trabajo desarrollando dos aplicaciones diferentes de principio a fin: recolección y análisis de datos, selección de modelo, entrenamiento de la red neuronal y optimización.
- Despliegue de dichas aplicaciones en microcontroladores para ser ejecutadas en tiempo real y con bajo consumo energético.
- Estudio de las últimas herramientas actualmente disponibles para implementación de *machine y deep learning* en microcontroladores.
- Experimentación con microcontroladores potentes, pero baratos, susceptibles de uso en *machine learning*.

1.2 Estructura del documento y cronograma

En este documento se presenta de manera formal el desarrollo de dos aplicaciones de aprendizaje profundo en sistemas empotrados. El código desarrollado es abierto y se encuentra en un repositorio en Github [3]. Además, se ha llevado a cabo la elaboración de una aplicación web [4] en donde se encuentra de manera visual y resumida información sobre la investigación, incluyendo demos en tiempo real y diagramas de flujo y funcionamiento.

La memoria se estructura de la siguiente manera. En el Capítulo 2 se explican brevemente los algoritmos basados en redes neuronales y se compara con el paradigma tradicional de la inteligencia artificial. En el Capítulo 3 se detallan las ventajas y desafíos asociados al denominado *TinyML* y se introducen algunas herramientas. Los dos siguientes capítulos se enfocan en el desarrollo de dos aplicaciones: clasificación multiclase de imágenes de personas y coches (Capítulo 4) y reconocimiento de la escritura en tiempo real (Capítulo 5), ambos utilizando microcontroladores. Por último, en el Capítulo 6 se presentan las conclusiones del trabajo realizado y posibles pasos futuros.

La Fig. 1 muestra el diagrama de Gantt de las actividades principales que se han llevado a cabo, incluyendo, entre otros, el estudio del área del aprendizaje profundo y *TinyML*, desarrollo de aplicaciones y documentación del trabajo en los distintos formatos [3, 4].

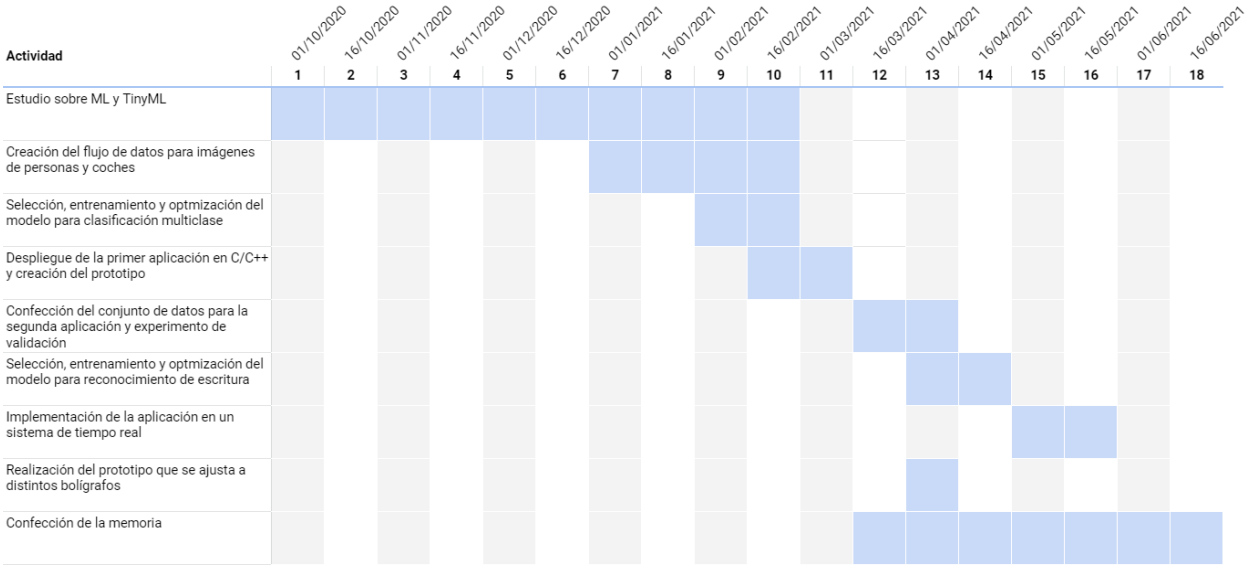

Fig. 1 Diagrama de Gantt

2. Aprendizaje profundo y redes neuronales

2.1 Redes neuronales artificiales

Antes de continuar, es importante entender de forma breve cual es la estructura de una red neuronal y cómo se entrena.

A pesar de ser su motivación inicial, el modelado de las neuronas biológicas del cerebro [5], las redes neuronales se han ido transformando con el tiempo en modelos matemáticos que no necesariamente modelan los fenómenos neurológicos. Estos algoritmos se basan en un conjunto de unidades lógico-matemáticas muy sencillas (neuronas artificiales) que se interconectan entre sí, tal y como se ve en la Fig. 2.

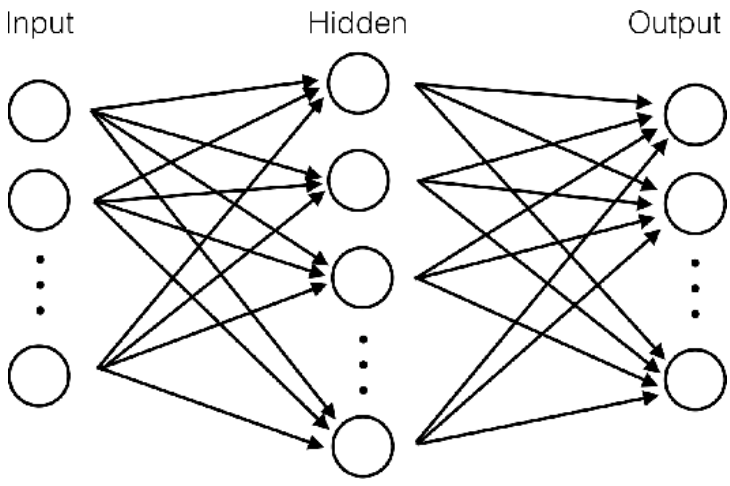

Fig. 2 Arquitectura típica de una red neuronal
Fuente: https://www.semanticscholar.org/

Una red neuronal se compone de un conjunto de capas: las de entrada y salida permiten introducir los datos y obtener los resultados, respectivamente; las capas intermedias u ocultas son las que generan las representaciones y realizan la computación efectiva. Cuando una red neuronal tiene muchas capas ocultas se habla de aprendizaje profundo (*deep learning*), donde la profundidad del modelo permite modelar relaciones entre la entrada y la salida más complejas.

En su forma más genérica, la relación entre una capa y su anterior viene dada por la siguiente expresión [6]:

$$h_{\mathbf{W},\mathbf{b}} = \phi\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right) \tag{2.1}$$

En esta ecuación:

- El vector **x** representa las salidas de las neuronas de la capa anterior.
- La matriz **W** representa la matriz de pesos, modelando la relación de cada neurona de la capa anterior con cada una de la capa actual.
- El vector **b** modela los sesgos (*bias*) de cada neurona, es simplemente un parámetro adicional.
- La función Φ se denomina función de activación de la neurona, generalmente se usa una función no lineal, para permitir modelar relaciones más complejas.

Como se puede observar, la operación de la neurona consiste en una simple transformación lineal seguida de una función de activación, generalmente no lineal.

Los parámetros de la red neuronal, conformados por los pesos (*weights*) y sesgos (*bias*), se obtienen entrenando la red mediante algoritmos de optimización. Para cada ejemplo de entrenamiento se computa una función de pérdida *J* (*loss*, también llamada función coste o función error) que mide la diferencia entre la salida proporcionada por la red y la salida deseada (*target*). Tras obtener el gradiente de dicha función con respecto a cada parámetro, se modifican dichos parámetros para reducir paulatinamente el valor de esta función de pérdida. Este es el fundamento del bien conocido algoritmo de *descenso por el gradiente (gradient descent)*. La actualización de cada parámetro *w* de la red se puede escribir matemáticamente como

$$w \leftarrow w - \eta \frac{\partial J(\mathbf{y}, \hat{\mathbf{y}})}{\partial w} \tag{2.2}$$

Donde la función de pérdida *J* depende del resultado esperado (*y*, *target*) y del producido $\hat{y}$, donde se introduce un parámetro η llamado ritmo de entrenamiento (*learning rate*) que sirve como paso del algoritmo de optimización y se ha de escoger manualmente al entrenar la red.

Estos modelos matemáticos sencillos se han ido mejorando y completando con el objetivo de producir mejores resultados y adaptarse a la aplicación de uso. Por ejemplo, las redes

neuronales convolucionales popularmente usadas en aplicaciones de visión por computador utilizan la convolución matemática para modelar las relaciones entre capas. Por su parte, las redes neuronales recurrentes añaden relaciones temporales.

Del mismo modo, las funciones de activación, algoritmos de optimización y el desarrollo de otras técnicas [6] han ayudado a favorecer el entrenamiento y funcionamiento de las redes neuronales. A lo largo del documento se describirán y justificarán el uso de estas técnicas en las aplicaciones desarrolladas.

2.2 Paradigma tradicional

Antes de la popularización de los algoritmos de aprendizaje automático (*machine learning*), la inteligencia artificial se desarrollaba de forma manual, donde todas las secuencias lógicas se debían escribir explícitamente. Por ejemplo, un programa de detección de spam se fijaría en la presencia de palabras u oraciones presentes en el e-mail concretas. Modelar a mano relaciones más complejas o la infinidad de maneras de formular oraciones resulta tedioso y generalmente sin mucho éxito. Se puede generalizar también a otras tareas como sistemas de recomendación o, especialmente, visión por computador, donde algoritmos de clasificación de imágenes era una tarea compleja y muy difícil de producir buenos resultados [7].

Con la aparición del aprendizaje automático, se ha habilitado el desarrollo de software para multitud aplicaciones de forma fiable y escalable que antes resultaba inviable. Los campos que más se han beneficiado de este nuevo paradigma son visión por computador, modelos del lenguaje y sistemas de recomendación. Sin embargo, multitud de otros sectores también han obtenido grandes avances gracias al aprendizaje automático, como la robótica [8], medicina [9] o incluso el sector inmobiliario [10].

2.3 Computación en la nube

Hasta hace poco tiempo, la tendencia del aprendizaje profundo se focalizaba en crear modelos cada vez más profundos y complejos para obtener mejores métricas en *datasets* populares como ImageNet [11]. La magnitud de estos modelos y *datasets* conlleva un entrenamiento de la red neuronal muy costoso computacionalmente, requiriendo de multitud de tarjetas gráficas (*GPUs*), que son el instrumento más común para esta tarea. Además, la complejidad añadida de entrenar un modelo con diferentes tarjetas gráficas supone un esfuerzo adicional.

Una vez entrenado el modelo, es cierto que el coste computacional de realizar inferencia es mucho menor, pero para obtener una latencia aceptable se tenían que ejecutar necesariamente en un servidor con GPUs especializadas.

Para muchas aplicaciones, la necesidad de disponer de una conexión a internet y la latencia que puede producir el envío y recepción de datos no resulta impedimento para desarrollarla. De hecho, existen multitud de servicios que utilizan *Cloud AI* para desplegar sus aplicaciones de aprendizaje profundo. Un ejemplo muy claro es el sistema de detección de

spam de los correos o la comprensión del lenguaje ofrecida por los asistentes virtuales más conocidos.

Sin embargo, muchas otras aplicaciones, típicamente aquellas que requieren una ejecución en tiempo real, no encajan dentro de este paradigma, por lo que hace necesario el procesamiento en local, mediante microcontroladores u otros dispositivos electrónicos.

2.4 Embedded AI

Con el transcurso del tiempo la necesidad de desplegar aplicaciones de aprendizaje profundo *on-device* ha crecido notablemente y se han elaborado técnicas [2, 27] que hacen posible su desarrollo. En concreto, los smartphones fueron los primeros dispositivos en adoptar este tipo de aplicaciones, por ejemplo para mejorar las fotos realizadas con la cámara. Actualmente resulta complicado diferenciar cuando una foto ha sido tomada con una cámara profesional o con un smartphone gracias a la inteligencia artificial.

En los últimos años también se han comenzado a desarrollar técnicas aún más sofisticadas y librerías en C/C++ que permiten que sistemas de prestaciones limitadas, como ordenadores monoplaca o incluso microcontroladores, tengan capacidad de realizar inferencia con un modelo pre-entrenado.

Las ventajas de realizar inferencia *on-device*, es decir, de procesar en local en vez de en la nube, son muy amplias:

- No se necesita conexión con un servidor online.
- Aplicaciones de tiempo real son ahora posibles sin sobrecargar un servidor.
- Desaparece la latencia de red.
- Los datos no salen del dispositivo, respetando la privacidad del usuario y aumentando la seguridad.

Y en concreto para microcontroladores:

- Desaparece una enorme parte del desarrollo al no necesitar conexión a internet.
- Generaliza la posibilidad de despliegue a cualquier parte del mundo por remota que sea.
- Coste muy ajustado. Los microcontroladores son dispositivos muy baratos en comparación a ordenadores monoplaca u ordenadores/smartphones en general.
- El consumo de potencia es drásticamente menor que en otros dispositivos, abriendo el abanico de aplicaciones a aquellas que requieren de baterías y gran autonomía.

No es sorprendente entonces que el desarrollo de estas técnicas y librerías esté en constante crecimiento.

Gracias a esta nueva tecnología han surgido más áreas de aplicación como la preservación de animales salvajes [12] o reconocimiento de la escritura en tiempo real. También ha impulsado el desarrollo en escala y bajo coste de aplicaciones de visión por computador como monitorización del crecimiento de plantas [13] o detección de personas. Otro sector

en constante investigación y desarrollo es el mantenimiento predictivo [26], que aporta ventajas logísticas, y por lo tanto económicas, en la industria.

En este documento se van a desarrollar y desplegar dos aplicaciones que demuestran que este tipo de aplicaciones en microcontroladores es conveniente, escalable y con un precio muy reducido. Los modelos en computador se desarrollarán en Keras bajo las librerías *Tensorflow* de Google, entornos Python bien conocidos que permiten el trabajo en computador y en la nube. Pero para la implementación de estos modelos en microcontrolador, en ambas aplicaciones utilizaremos nuevas herramientas, hasta ahora menos extendidas, que permiten trabajar con los recursos reducidos de un microcontrolador.

3. Desarrollo de aplicaciones de aprendizaje profundo en sistemas empotrados

A la hora de plantearse la decisión de desarrollar una aplicación de aprendizaje profundo es crucial tener en cuenta las condiciones del sistema en el que se va a desplegar. En concreto, los microcontroladores tienen unas características mucho más modestas que ordenadores o servidores tradicionales, es por ello que es importante tener en cuenta que:

- El **tamaño** del modelo viene determinado mayoritariamente por el número de parámetros de la red, y este tiene que ser lo suficientemente pequeño como para caber dentro de la memoria flash, normalmente del orden de los kB. Para lidiar con este problema los modelos se discretizan o *cuantizan*. Es decir, los parámetros pasan de ser números reales codificados como *float* de 32 bits, a enteros de 8 bits, resultando en un modelo aproximadamente de tamaño cuatro veces menor.
- La **latencia** del modelo viene determinada por el tipo (*float* o *integers*) y número de operaciones que se tienen que llevar a cabo; normalmente es proporcional al número de parámetros. Al trabajar con sistemas cuya frecuencia de reloj no sobrepasan los 100 MHz es muy importante considerar su tiempo de ejecución para poder ser planificados dentro del sistema operativo de tiempo real. Herramientas como TFLite utilizan técnicas denominadas "*pruning*" para "podar" el modelo simplificando o eliminando operaciones poco relevantes, reduciendo de forma efectiva su tamaño y número de operaciones.
- En cuanto a la memoria **RAM**, es difícil obtener una estimación concreta, pero de forma orientativa se tiene que poder asegurar que las capas más anchas, como la entrada (por ejemplo si es una imagen), o una capa convolucional con muchos filtros y sus resultados, caben en la memoria RAM.
- La **naturaleza** de nuestro problema puede influir de forma notable en la decisión del sistema a desplegar: ¿con qué periodicidad se tiene que ejecutar inferencia en nuestro modelo?, ¿con qué frecuencia se tiene que muestrear la entrada?, ¿a donde hay que enviar la información de salida? Por ejemplo, un sistema que requiera

reconocer palabras (*speech recognition*) deberá muestrear a gran frecuencia y de forma continuada, dejando menos tiempo para el microcontrolador a realizar otras tareas, como la propia ejecución del modelo, o para pasar a modo de bajo consumo.

Los tres primeros desafíos se ven muy aliviados gracias a herramientas muy recientes como *TFLite* (https://www.tensorflow.org/lite) o *Edge Impulse* (https://www.edgeimpulse.com/), que permiten, por una parte, realizar optimizaciones al modelo y, por otra, estimar sus condiciones de ejecución en diversos microcontroladores.

Precisamente, en este sentido el objetivo fundamental del trabajo es estudiar este incipiente campo del denominado TinyML, o aprendizaje automático en sistemas empotrados de prestaciones, coste y consumo reducidos, analizando las herramientas (como las dos citadas) y metodologías actualmente disponibles mediante el desarrollo de dos aplicaciones, que vamos a presentar en los dos capítulos siguientes: clasificación de imágenes de personas y/o coches para control de tráfico (Capítulo 4) y el reconocimiento *online* de escritura a mano mediante el procesamiento de los datos de un acelerómetro incorporado en un bolígrafo (Capítulo 5). Con ambas aplicaciones, orientadas al procesamiento de datos generados por dos tipos de sensores típicos, pero muy diferentes (cámara y acelerómetro), queremos valorar dichas herramientas y metodologías.

Finalmente, el desarrollador debe ser lo suficientemente audaz como para entender el problema y enfrentarse al último desafío. Deberá planear de antemano sus decisiones de diseño en cuanto a frecuencia de muestreo y ejecución de la red neuronal, así como la posible necesidad de que los datos de salida se envíen a un servidor, por ejemplo, a través de tecnologías LPWAN como LoRa. Todo esto condiciona la selección de modelo.

4. Detección de personas y coches en cruces

4.1 Justificación

En la actualidad una amplia mayoría de semáforos son controlados con sistemas muy básicos, como son los temporizadores, con tiempos en verde y rojo predeterminados, algunos en función de la hora y estación. Esta implementación resulta en muchas ocasiones ineficiente, donde peatones o, sobre todo, vehículos, han de detenerse y esperar de forma innecesaria.

Para estimar la demanda y poder favorecer el paso para unos u otros, se propone un sistema dotado de una cámara capaz de detectar personas y coches. Posteriormente, estos datos se pueden procesar en el propio dispositivo o ser enviados a una centralita con el objetivo de reducir los tiempos de espera.

Los beneficios inmediatos son una circulación más fluida y menores tiempos de espera, pero también reducen el consumo de combustible y, por tanto, de emisiones de CO2. Por ejemplo, por la noche es probable que un vehículo circule sin apenas peatones en la calle, el algoritmo sería capaz de entender esta demanda y actuar poniendo el semáforo en ámbar intermitente.

La aplicación propuesta se enfoca en desarrollar una prueba de concepto de la primera fase: la detección de personas y/o vehículos a través de una cámara instalada en un sistema empotrado. Tras desarrollar la aplicación en entorno computador, se portará a un microcontrolador de ST al que se conectará una cámara OV7670 de resolución QVGA 320x240, pequeña y muy económica. El hecho de poder desarrollar esta aplicación de forma empotrada permite que su despliegue futuro tenga un coste ajustado y por lo tanto sea escalable. Dicha información puede ser más tarde procesada en el propio dispositivo o enviada a través de algún protocolo de comunicación de bajo consumo, como LoRaWAN.

4.2 Obtención de datos

En el desarrollo final de la aplicación es vital que el modelo se entrene en imágenes tomadas desde los semáforos o intersecciones, con el fin de que la distribución de los datos con los que se entrena sea lo más representativa de la realidad a la que se van a enfrentar una vez desplegada la aplicación. Sin embargo, con el fin de realizar una prueba de concepto fiable y un mínimo producto viable, se propone el entrenamiento con las imágenes obtenidas de los siguientes *datasets* y con las siguientes consideraciones:

- **COCO** [14] Es típicamente usado para segmentación de imágenes, pero se puede extrapolar a clasificación multiclase. Tiene tanto imágenes de coches como de personas en distintos contextos, aportando variedad a la naturaleza de los datos y con el objetivo de aportar robustez al modelo.
- Algunas imágenes de COCO contienen las clases en las que estamos interesados pero en tamaños muy pequeños o incluso despreciables, por lo que se ha implementado un sistema que filtre aquellas en las que el área que ocupa la clase relevante tiene un área menor que una constante fijada por el usuario.
- Solo con las imágenes descargadas de COCO, la clase “persona” tiene seis veces más ejemplos que la clase “coche”. Para lidiar con este dataset no balanceado, en un principio se intentó aplicar la técnica de *downsampling*, en la que se establece un máximo de imágenes por clase igual al número de ejemplos de la clase subrepresentada. Es por ello que se propone la adición del ***Cars Dataset*** [15].
- Un inconveniente del *Cars Dataset* es que en todas las imágenes los coches están centrados, y con perspectivas e iluminación parecidas. Esto hace que el modelo no generalice tan bien la clase “coche”.
- Para esta aplicación la mayoría de las pruebas se van a realizar en casa, en la universidad o en espacios cerrados en general. Además, una curiosa observación del dataset COCO es que la mayoría de las imágenes que contienen la clase persona es en espacios cerrados. Esto llevó en las primeras iteraciones de la aplicación a que el

modelo sobreestimase la clase “persona” durante las pruebas. Como solución a este problema se añaden imágenes que contienen imágenes de interiores pero sin personas a una clase “negativa”. Se obtienen del ***MIT Indoor Dataset*** [16].

- Se ha añadido una clase “negativa” (clase de rechazo) con el objetivo de ayudar al modelo a aprender que en muchas ocasiones en la imagen no aparecen ni personas ni coches.

La Fig.3 muestra algunos ejemplos de las imágenes del set de datos final.

Cabe destacar que hablamos de clasificación **multiclase**, es decir, el modelo será capaz de detectar si hay un coche, una persona, o ambas al mismo tiempo en la imagen. Este tipo de problema no es tan habitual como la clasificación monoclase y se ha tenido que recurrir a técnicas manuales para obtener el set de datos correctamente.

En cualquier caso, se ha desarrollado la obtención de datos de forma íntegra en Python, bien utilizando una API pública (COCO) o accediendo al servidor directamente para descargar los datos.

Además, para optimizar el uso en RAM y asegurar un *pipeline* (flujo) de datos eficiente se ha usado `tensorflow.data`, dotando al sistema de gran flexibilidad.

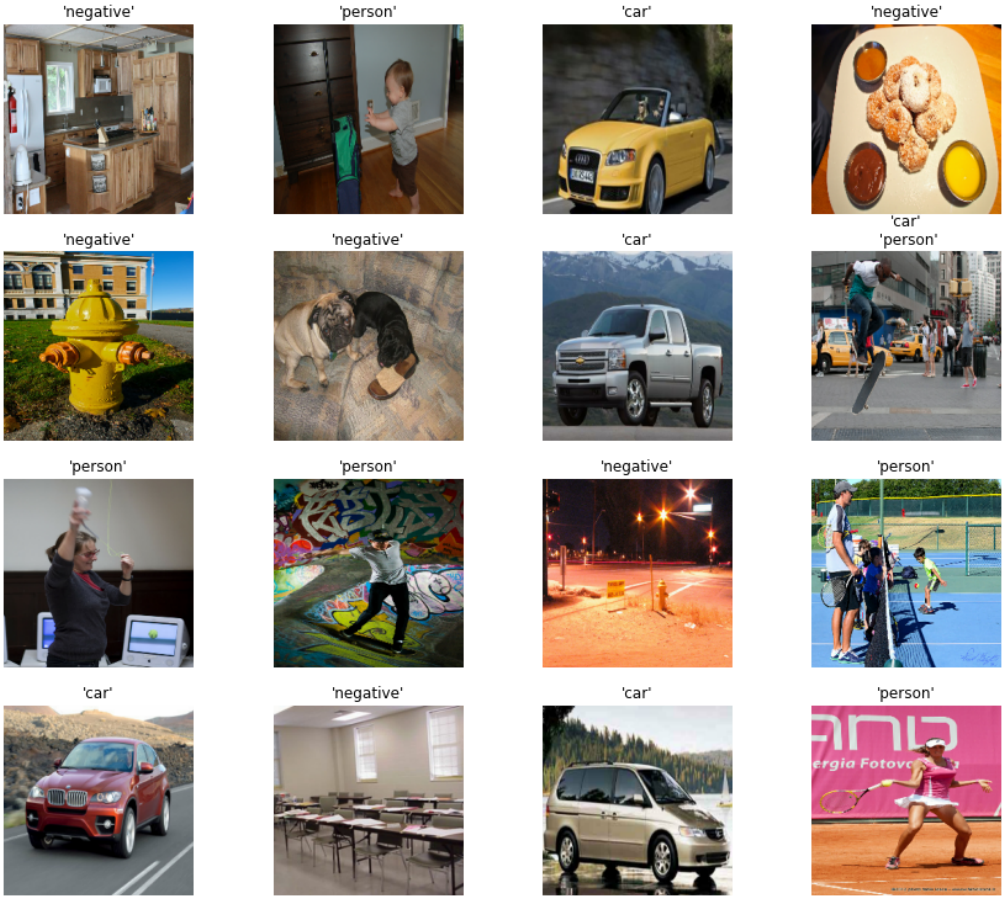

Fig. 3
Ejemplos del set de datos final

4.3 Justificación del modelo

Los modelos más populares (*state-of-the-art)* de aprendizaje profundo aplicado a visión por computador tienden a ser muy grandes, ocupando gran espacio en disco y memoria, así como una latencia considerable. La creciente preocupación para mover el cómputo de la ejecución de redes neuronales del servidor (*server-side)* al cliente (*client-side)* han dado lugar a arquitecturas mucho más ligeras, pero manteniendo métricas competitivas. La siguiente tabla compara alguna de las más populares.

Model	**Número de parametros**	**FLOPS (millones)**	**Top-1 Error**	**Top-5 Error**	**Año**
EfficientNet-B0	5,288,548	414.31	24.77	7.52	2019
NASNet-A 4@1056	5,289,978	584.90	25.68	8.16	2017
MobileNet	4,231,976	579.80	26.61	8.95	2017
MobileNetV2	3,504,960	329.36	26.97	8.87	2018
ShuffleNetV2	2,278,604	149.72	31.44	11.63	2018

Tabla 1. Comparación de modelos de visión por computador con *deep learning*

Las métricas se refieren al modelo entrenado y evaluado en *ImageNet*, un famoso set de datos de imágenes con mil clases utilizado en multitud de artículos científicos para comparar arquitecturas y técnicas.

Antes de continuar es esencial tener en cuenta el sistema al que se va a desplegar el modelo: un **microcontrolador STM32H7**, con 2 MB de memoria flash, 1 MB de memoria RAM y una frecuencia de reloj de 480 MHz. Se corresponde a las gamas más altas de la familia, con el objetivo de desarrollar una aplicación muy fiable y profesional pero que sigue manteniendo un coste muy ajustado y por lo tanto es escalable.

Desafortunadamente, ninguno de esos modelos puede caber en el microcontrolador, al menos en su forma original. *EfficientNet* [17] y *NASNet* [18] tienen muy buen rendimiento, pero *MobileNet* [19] y *MobileNetV2* [20] tienen hiperparámetros que permiten ajustar el tamaño de la arquitectura para hacerla aún más ligera y tienen un rendimiento muy parecido.

Además, como se comentará posteriormente, se va a utilizar la técnica de *transfer learning* [6] a la hora de entrenar el modelo, por lo que es estrictamente necesario que exista un modelo pre-entrenado. La librería *Keras* ofrece estos modelos y con distintos hiperparámetros para *MobileNet* y *MobileNetV2*.

Antes de proceder a entrenar el modelo con estas arquitecturas y verificar que su tamaño es suficientemente pequeño vale la pena entender estas arquitecturas.

MobileNets

Propuesta por primera vez en 2017 por Google [19], demostró que modelos mucho más ligeros y rápidos podían alcanzar métricas muy competitivas. La idea principal se basa en el uso de *depthwise separable convolution* en lugar de la convolución tradicional. En primer lugar, un filtro es aplicado a cada uno de los canales **por separado** (solo en la dimensión espacial). En segundo lugar, un filtro 1x1 es aplicado a lo largo de los canales del resultado anterior. Se puede demostrar que el ratio de coste computacional entre una capa convolucional separable y una estándar es:

$$r = \frac{1}{N} + \frac{1}{D_k^2} \tag{4.1}$$

Donde *N* es el número de filtros de salida y *Dk* es el ancho y alto del filtro espacial. Con *N* en el rango de [32, 1024] y tamaños de kernel para las capas convolucionales separables de 3x3, éstas requieren entre 7 y 9 veces menos computaciones que su contrapartida tradicional. Además, dado que la arquitectura entra dentro del grupo de *fully convolutional networks*, el tamaño de la imagen de entrada con la que se puede entrenar se puede elegir de antemano y hacerlo más pequeño. Este hiperparámetro es denominado *resolution multiplier* y reduce notablemente el uso de memoria, así como su tamaño y coste computacional.

El otro hiperparámetro, denominado *width multiplier*, con la letra griega "α", es representativo de la anchura de las capas convolucionales, refiriéndose al número de canales, que quedarán dados por α*N* o α*M*, según hablemos de canales de entrada o salida. Se puede demostrar que el coste computacional es reducido cuadráticamente en un factor α^2.

La Fig.4 detalla la arquitectura de *MobileNet*, donde se aprecia claramente el uso de convolución separable, así como de prácticas comunes, como el uso de *stride* mayor que uno para reducir la dimensión espacial, así como capas con más filtros cuanto más profundas se encuentran en la red.

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

Fig. 4 Fuente: MobileNet - Google Inc.
Arquitectura completa de MobileNet

MobileNetV2 se propuso en 2019 por Google [20] e itera sobre su predecesor, añadiendo conceptos como residuales invertidos o cuellos de botella lineales, mejorando su rendimiento.

En conclusión, en esta aplicación vamos a explorar distintas configuraciones para las arquitecturas de *MobileNet* y *MobileNetV2,* probando distintos hiperparámetros y comparando su rendimiento y huella (tamaño y coste computacional).

4.4 Entrenamiento del modelo

4.4.1 Función de pérdidas y métricas

Como se ha mencionado anteriormente, el problema a tratar es la clasificación multiclase, en el que una o varias de las clases pueden estar presentes en la imagen.

Como métricas, además de la exactitud (*accuracy*) resulta interesante conocer la precisión (*precision)* y la exhaustividad (*recall)* de cada clase [6]. En especial y sin realizar ninguna asunción previa se propone usar el Valor-F (*F1-Score)*, que es la media armónica de de las dos últimas [6]:

$$F_1 = \frac{2}{\text{recall}^{-1} + \text{precision}^{-1}} \tag{4.2}$$

La media armónica es equivalente a la media aritmética para los inversos de las cantidades que deberían ser mediados aritméticamente. Intuitivamente, se puede observar que tanto la precisión como la exhaustividad comparten el mismo denominador (verdaderos positivos). Cuando realizamos la inversa de estas cantidades podemos entonces realizar la media aritmética de los falsos positivos y los falsos negativos sobre los positivos verdaderos positivos. Por último, se hace la inversa de este resultado para obtener la representación original.

A modo de visualización, la siguiente imagen (Fig. 5) representa el valor F1 en función de la precisión y exhaustividad, demostrando que ambas contribuciones son importantes para un buen resultado:

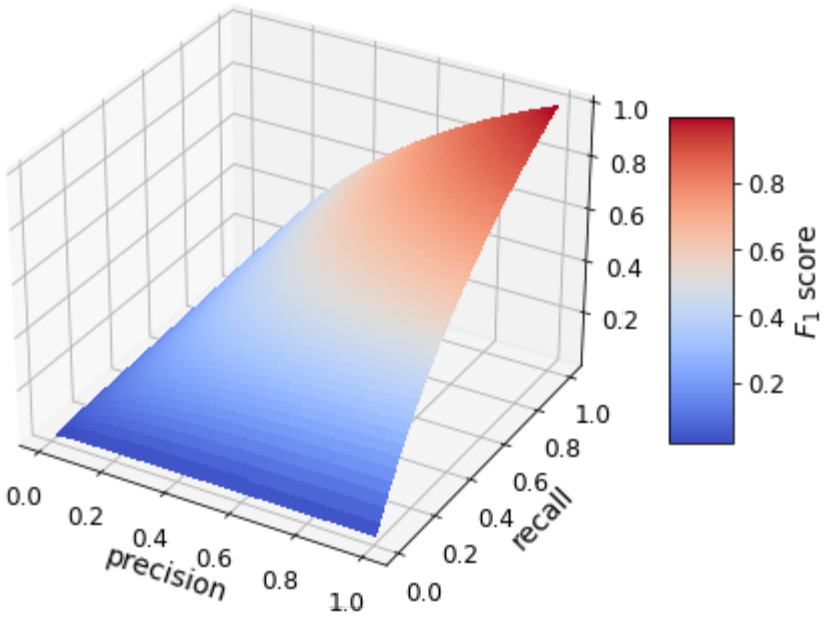

Fig. 5
Valor-F1 en función de la precisión y la exhaustividad

Para el caso de la clasificación multiclase podemos realizar la media del valor F1 de cada clase, que llamaremos *macro F1-score*.

Yendo un paso más lejos, existen artículos [21] que extienden la definición de esta métrica en una función diferenciable para poder usarla como función de pérdida. Esto tiene la ventaja de optimizar el modelo a la métrica que realmente nos interesa, entre otros. Se ha realizado la implementación usando *Tensorflow* con el objetivo de verificar sus resultados.

De todos modos, se comprobará esta asunción contrastando los resultados con la función común de pérdida *binary cross-entropy,* que optimiza la *máxima verosimilitud logarítmica* (*log-likelihood*) [1, 6].

4.4.2 Estrategia de entrenamiento

El entrenamiento de las redes neuronales es un área que sigue en desarrollo y está en constante estudio. El entrenamiento de una red convolucional partiendo de cero es un proceso largo y tedioso, por lo que habitualmente se recurre al denominado aprendizaje por transferencia o *transfer learning* [6], consistente en el uso de modelos ya pre-entrenados, que se van a adaptar a una tarea diferente, pero similar. La práctica común en tareas de visión por computador es reutilizar las capas convolucionales, que han aprendido a extraer rasgos de un dataset de enorme tamaño, tras horas y horas de entrenamiento (por ejemplo, han aprendido a detectar patrones como esquinas o bordes, o más especializados como caras u hojas). De esta forma, se procede a entrenar el modelo en dos fases:

- **Entrenamiento de las últimas capas densas** (*classification head*): se retiran las últimas capas del modelo pre-entrenado y se añaden unas nuevas, compuestas por una capa *average pooling*, que realiza la media de los últimos filtros del modelo, una capa densa y la capa de salida. También se incluye *dropout* y regularización de las activaciones de la capa densa. Durante esta fase el modelo se entrena ya con el nuevo dataset (que no requiere ya de tantos ejemplos) con un *learning rate* por defecto y optimizador *Adam,* y sólo aprenden estas nuevas capas.
- **Ajuste fino** (*fine tuning*): se procede a descongelar algunas capas del modelo pre-entrenado (*feature extractor*) y se realiza otro bucle de entrenamiento, esta vez con un *learning rate* mucho más pequeño y optimizador RMSProp.

La motivación de realizar aprendizaje por transferencia en dos bucles de entrenamiento tiene su origen en el hecho que es muy probable que por culpa de las nuevas capas añadidas las anteriores pierdan calidad, dado que los pesos iniciales de las nuevas capas son aleatorios.

La siguiente tabla resume los modelos entrenados y sus resultados, incluyendo también aquellos que se entrenaron con menos imágenes y de forma más sencilla con el objetivo de tener un punto de partida (*baseline*):

Model number	Version	alpha	Img size	loss	Dense neurons	Size INT8 (kB)	Train acc	Val acc	Val macroF1
1	V1	0.5	224x224	BCE	256	935	0.841	0.817	-
2	V2	0.5	224x224	BCE	256	1015	0.859	0.830	-
3	V2	0.35	224x224	BCE	256	732	0.844	0.829	-
4	V1	0.25	224x224	SoftF1	256	732	0.915	0.935	0.907
5	V1	0.25	224x224	BCE	256	732	0.912	0.933	0.904
6	V1	0.25	224x224	SoftF1	128	572	0.914	0.935	0.908
7	V1	0.25	224x224	SoftF1	64	345	0.915	0.935	0.907
8	V1	0.25	160x160	SoftF1	64	345	0.909	0.928	0.897

Tabla 2. Comparación de modelos entrenados

Antes de analizar las métricas, conviene señalar que, desafortunadamente y como se explicará más tarde, algunas operaciones de *MobileNetV2* no están soportadas por *TFLite Micro*, el paquete de software (SDK) que interpreta el modelo en C/C++. Esto se ha comprobado tras entrenar estos modelos e intentar desplegarlos al microcontrolador. Por lo tanto, será necesario trabajar exclusivamente con *MobileNet*.

Como aclaración, *dense neurons* se refiere al número de neuronas de la capa densa (todas las neuronas de una capa conectadas con las de la siguiente) que se ha añadido después del modelo pre-entrenado. *BCE* es una forma abreviada de *binary cross-entropy.* El tamaño de los modelos se mide una vez comprimido y discretizado para que los parámetros sean enteros de 8 bits, en vez de float de 32 bits (más detalles sobre esto en la siguiente sección).

Así pues, en la Tabla 2 podemos observar que el modelo 4 y 5 son iguales excepto en la función de pérdida usada. *BCE* parece tener un peor rendimiento y no optimiza la métrica que estamos buscando, por lo que podemos validar que la extensión diferenciable del Valor-F1 es una buena función de pérdida para clasificación multiclase.

Por su parte, los modelos 6 y 7 estudian el rendimiento reduciendo el número de neuronas en la capa densa, reduciendo de forma considerable el tamaño del modelo. Se observan métricas similares, por lo que se concluye que 64 neuronas es más que suficiente para un buen rendimiento.

Hasta el momento el tamaño de la imagen de entrada utilizada ha sido de 224x224x3, pero tras realizar una prueba con la resolución más pequeña para la que existe modelo pre-entrenado, 160x160x3, se obtienen métricas ligeramente peores, pero muy similares.

Para estudiar su viabilidad según el artículo de *MobileNet* el coste computacional se verá reducido aproximadamente por un factor

$$\rho^2 = \frac{160 \times 160}{224 \times 224}^2 = 0.26 \quad (4.3)$$

Es decir, el modelo tendrá una latencia aproximadamente cuatro veces menor. Este último modelo será el final, con métricas similares a los demás y con un rendimiento y tamaño mucho más favorable. En la Tabla 3 puede verse en detalle las métricas para cada clase.

	Precisión	Exhaustividad	Valor-F1
persona	0.93	0.97	0.95
coche	0.89	0.86	0.88
negativo	0.92	0.83	0.88

Tabla 3. Métricas del modelo final

Como se puede observar, el modelo reconoce mejora la clase “persona”, lo que no resulta del todo sorprendente, pues es la clase con más variedad de ejemplos y de la misma naturaleza. En anteriores iteraciones y con menos datos, la clase “coche” estaba muy subrepresentada. Afortunadamente, la adición de los diferentes set de datos consiguen un modelo más robusto.

4.5 Optimización y compresión

Los modelos anteriores se han desarrollado en Keras bajo las librerías TensorFlow de Google. Este entorno Python permite el trabajo en computador y en la nube. Para el desarrollo de sistemas empotrados inteligentes Google hace pocos años introdujo TensorFlow Lite, *TFLite*, que vamos a utilizar a continuación.

Para la compresión del modelo y su adaptación a microcontroladores se usará el optimizador de *TFLite* para microcontroladores, en donde se ha especificado la discretización (cuantización) completa del modelo. Esto quiere decir que a través de un set de datos representativo se hallan los pesos y sesgos para que el modelo esté formado íntegramente por enteros de 8 bits. Este modelo final es cuatro veces más pequeño que su contrapartida en *float* de 32 bits. Además, en la mayor parte de los microcontroladores, las operaciones de *float* no están directamente soportadas por hardware. Por su parte, también reduce su ocupación de RAM, que es el factor limitante en esta aplicación, como se va a explicar a continuación.

Además, hay que tener en cuenta el formato en el que se van a capturar las imágenes suministradas por la cámara, que son RGB565 de 240x160, claramente distinto al de la entrada de la red (RGB888). Para simplificar la tarea de implementación se propone añadir en forma de capas adicionales el preprocesado de la imagen. Sin embargo, a fecha de este escrito la librería de *TFLite Micro* tiene un set de operaciones soportadas muy limitado, por lo que hay que asegurarse de incluir sólo éstas. En más detalle, el preprocesado se hará de la siguiente forma:

- La conversión de RGB565 a RGB888 se hará en el microcontrolador, ya que son operaciones lógicas que se ven favorecidas por ejecutarse directamente (sin intérprete de *TFLite)*. Además, el tipo *uint16* no está soportado, que es el ideal para trabajar con RGB565.
- La normalización de la imagen (aunque luego *TFLite* la optimice*)* se realiza mediante la multiplicación de una constante. La operación de división no está soportada, así que se deberá hallar la constante equivalente aritmética.
- La redimensión de la imagen podría ser algo costosa de implementar en C. Afortunadamente, si se selecciona el método de interpolación "vecino más cercano" en la operación *resize*, está soportada por la librería.

Es pertinente también señalar que en realidad no se han añadido capas densas al modelo, ya que la operación *reshape* daba problemas de compatibilidad (si bien está soportada). Como solución al problema se han aplicado técnicas para convertir la red en una *fully convolutional network*. Matemáticamente, si la capa convolucional tiene filtros de tamaño 1x1 y tantos como neuronas deseadas de salida deseadas, es equivalente a una capa densa.

Como se puede observar, si bien el optimizador de *TFLite* es sencillo de utilizar, hay que tener en cuenta sus limitaciones y encontrar soluciones creativas que permitan desplegar el modelo deseado.

4.6 Despliegue de la aplicación en STM32H7

En el prototipo desarrollado(Fig. 6), las imágenes se capturan mediante una **cámara OV7670** de resolución QVGA 320x240, pequeña y muy económica, procesándose en tiempo real en un **microcontrolador STM32H7**. Con un procesador Cortex-M7, 2 MB de memoria flash, 1 MB de memoria RAM y 480 MHz, el **STM32H7** se sitúa entre las gamas más altas de microcontroladores, con el objetivo de poder integrar arquitecturas profesionales y obtener una latencia suficiente para la aplicación en mano. Además sigue manteniendo un coste ajustado, alrededor de los ocho euros tan solo. En concreto, se utilizará la placa de desarrollo **NUCLEO-H743ZI2** para el desarrollo del prototipo.

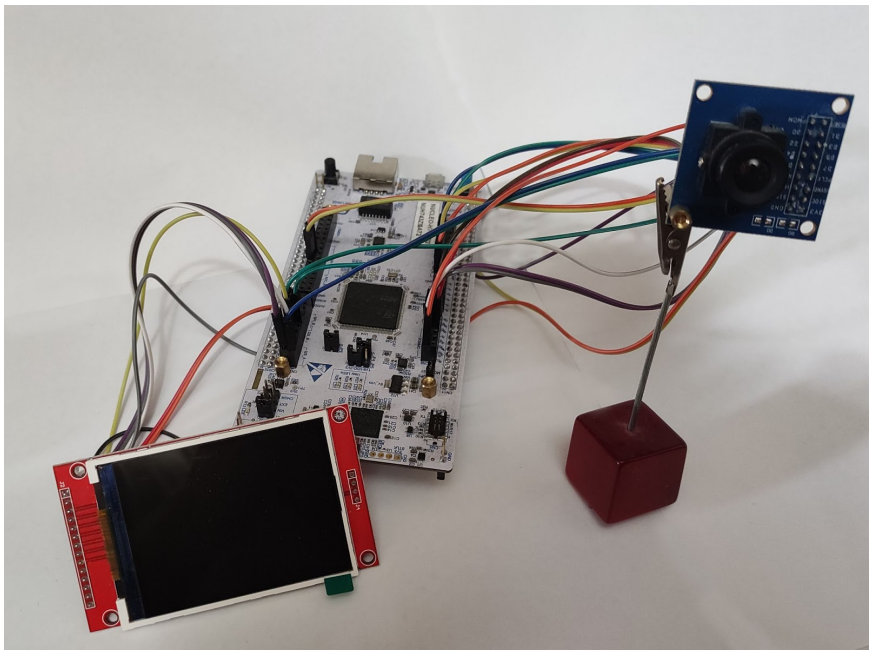

Fig. 6
Prototipo funcional de la aplicación

Para la implementación en C/C++ de la aplicación, se ha decidido usar las librerías de abstracción de hardware del **STM32H7** y apoyarse en el uso del entorno de desarrollo **STM32CubeIDE**, derivado del entorno Eclipse (Fig. 7) y equipado con multitud de funcionalidades para facilitar el trabajo con microcontroladores del fabricante ST, además de extensiones o librerías del ecosistema listas para ser importadas.

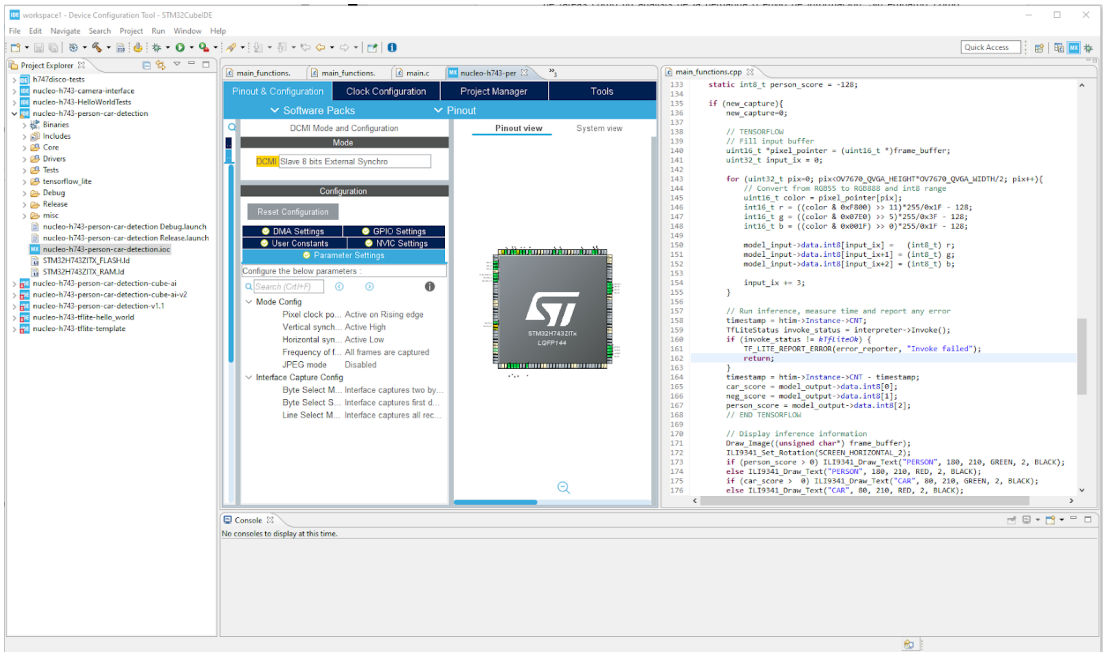

Fig. 7
Entorno de desarrollo STM32CubeIDE y herramienta CubeMX
para configurar periféricos

En concreto cabe destacar el paquete **X-CUBE-AI**, propietario de ST y utilizado también para *deep learning.* Se ha decidido usar ***TFLite Micro*** por su portabilidad y por ser código abierto.

El enfoque básico de esta primera aplicación ha sido la selección del modelo y su entrenamiento. Por su parte, la implementación es sencilla pero óptima, tal y como se muestra en la Fig.8.

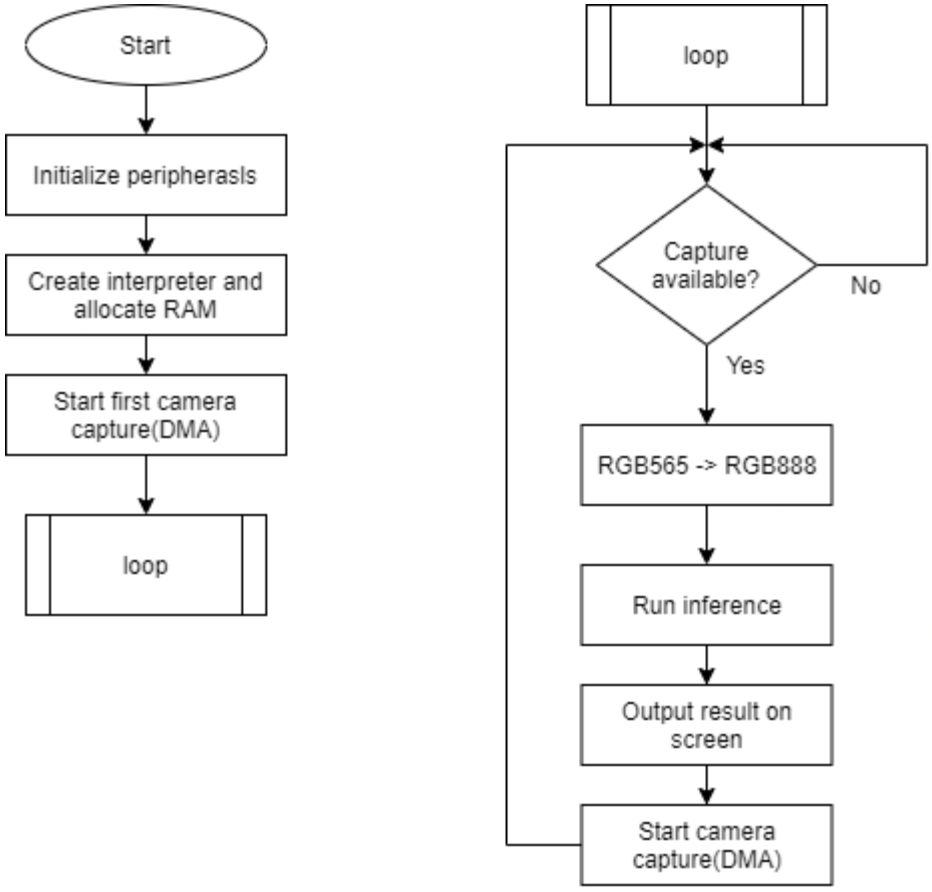

Fig. 8
Diagrama de flujo de la aplicación

En primer lugar, se inician todos los periféricos (internos y externos), incluyendo:

- El driver para la cámara, que utiliza comunicación I2C para los comandos más un bus de datos paralelo para transferir la imágen y las señales de sincronización. Gracias al periférico DCMI del microcontrolador se configura para que el proceso de recepción no bloquee el sistema usando acceso directo a memoria (DMA) con interrupciones.
- El driver para la pantalla (a modo de demostración) se configura mediante SPI.
- También se usa UART para obtener los resultados y depurarlos.
- Temporizadores HTIM para analizar la latencia.

Para crear el intérprete del modelo y que sea lo más ligero posible, se añaden las operaciones que requiere de forma manual. Esto quiere decir que solo se compilarán

aquellos módulos del paquete de *TFLite Micro* que sean realmente necesarios para el modelo en cuestión.

Una vez lanzada la captura de la imágen (no bloqueante), el sistema podría realizar otra serie de tareas como un análisis de la demanda o envío de información. Sin embargo, como prueba de concepto la aplicación desarrollada espera a recibirla, transforma el formato de la imágen y realiza inferencia.

A modo de visualización la imagen capturada y dos indicadores superpuestos indican las clases que se han detectado en ese instante, rojo en caso de que no y verde en caso afirmativo.

Se han realizado numerosas pruebas en casa y, dado que no se dispone de coches en interiores, se ha presentado una imagen ante la cámara para la clase “coche”. Algunos de los resultados se muestran en la Fig.9.

Fig. 9.
Pruebas de rendimiento en el prototipo

El rendimiento en general es aceptable, si bien es cierto que tiende a sobreestimar la clase persona o subestimar la clase coche. Este funcionamiento, quizá no tan ideal, era de esperar pues estamos exponiendo el modelo a una naturaleza considerablemente distinta a la que está entrenada, por ejemplo al poner la imágen de un coche en vez de un coche real en la cámara.

Tras el éxito de la ejecución del modelo y su correcto funcionamiento, queda demostrado que una aplicación de visión por computador con arquitectura de aprendizaje profundo con un modelo que utiliza una arquitectura profesional (*MobileNet) es* posible en microcontroladores de alta gama.

Cabe destacar que la cámara utilizada OV7670 es muy económica y tiene unas especificaciones más que suficientes. Sin embargo, pese a que la captura es realmente en resolución QVGA (320x240), el consumo de RAM de la captura y la memoria dedicada al modelo superan el 1MB. Por lo tanto se propone reducir la resolución de captura a la mitad, resultando en una imagen de 240 píxeles de alto y 160 píxeles de ancho. Se ha comprobado que esta transformación sólo afecta ligeramente (~0.1% en Valor-F1) en el set de datos de validación.

La ejecución de la red neuronal en el microcontrolador STM32H7, a una frecuencia de 480 MHz y un procesador Cortex-M7 tarda aproximadamente **1195 ms** (poco más de un

segundo) y el tamaño del modelo es de solo **345 kB** en memoria **flash** y de **364 kB** en memoria **RAM**, que es el tamaño máximo de RAM contigua en el STM32H7. Para la aplicación con la que se justifica, que es es la estimación de tráfico, es más que suficiente. De hecho, se propone que el sistema realice inferencia de forma menos frecuente, con el objetivo de consumir menos potencia y, en el caso de que se envíen los resultados, ancho de banda.

En la aplicación mostrada, se envía la imagen a una pantalla LCD-TFT a través de un controlador ILI9341. Obviamente esto no es necesario en la aplicación final, sino una conveniencia para mostrar su funcionamiento. La Fig.10 muestra el hardware empleado en el prototipo y la Fig. 6 el prototipo funcional.

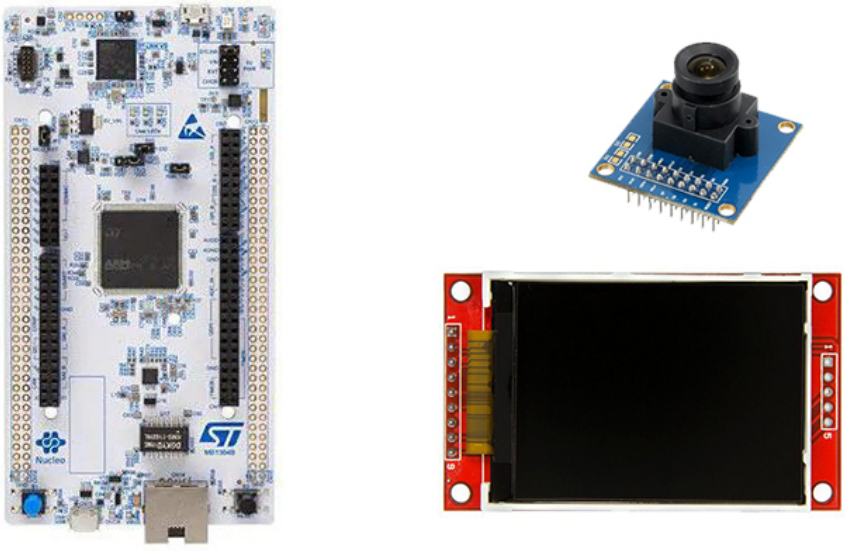

Fig. 10
Hardware utilizado en el prototipo: placa de desarrollo NUCLEO-H743ZI2, Cámara OV7670 y pantalla LCD-TFT con ILI9341

4.7 Conclusiones

En las secciones anteriores se ha descrito el desarrollo de principio a final de una aplicación de aprendizaje profundo que sirve como prueba de concepto para estimar la presencia de coches y/o personas en una imagen y con el objetivo de mejorar el tráfico y reducir el consumo de combustible y de emisiones de CO_2.

Se ha observado que no es nada trivial la obtención de un set de datos que represente la verdadera distribución a la que se va a enfrentar el modelo. Del mismo modo, la selección de un modelo que sea capaz de encajar en el microcontrolador y que al mismo tiempo esté soportado por *TFLite Micro,* a la vez que sea profesional, puede ser un verdadero desafío y requiere de soluciones creativas.

A la hora de desplegar el modelo, el mayor peso de la aplicación cae realmente en configurar los periféricos y desarrollar los controladores de la cámara y de la pantalla (en el caso que se utilice). Con aproximadamente **un segundo de latencia** y tan solo **345 kB** la aplicación se ejecuta con éxito en el microcontrolador con un rendimiento bueno, si bien nunca se ha entrenado en los escenarios probados (casa, universidad).

Por lo tanto, el desarrollo de aplicaciones de aprendizaje profundo y, en concreto, de visión por computador en sistemas empotrados es factible, siendo un área que está en auge gracias al coste ajustado, flexibilidad y escalabilidad de estos dispositivos.

5. Reconocimiento de escritura (on-line)

5.1 Justificación

El reconocimiento de caracteres es una práctica común en distintas aplicaciones [22, 23] y es típicamente realizado mediante técnicas de reconocimiento de imágen. A veces el uso de una cámara o de un sensor óptico puede resultar inconveniente, así que se propone un método alternativo.

Un acelerómetro LSM9DS1 de tres ejes del fabricante de ST se coloca en la culata de un bolígrafo (donde no resulta incómodo) y se mide la aceleración en los tres ejes mientras se escribe un caracter. La red neuronal procesa los datos de forma continua y genera predicciones si se detecta un patrón de movimiento que se corresponde con alguno de los caracteres para los que se ha entrenado.

Este método proporciona una manera más barata y menos intrusiva para el reconocimiento de caracteres escritos. Además, las predicciones se realizan *online*, mientras se dibujan los trazos, por eso se le denomina reconocimiento en directo (i.e. *online handwriting recognition)*.

Para ilustrar el concepto, la Fig.11 muestra el prototipo realizado mediante una estructura impresa en 3D y material flexible, que se adapta a distintos anchos de bolígrafo.

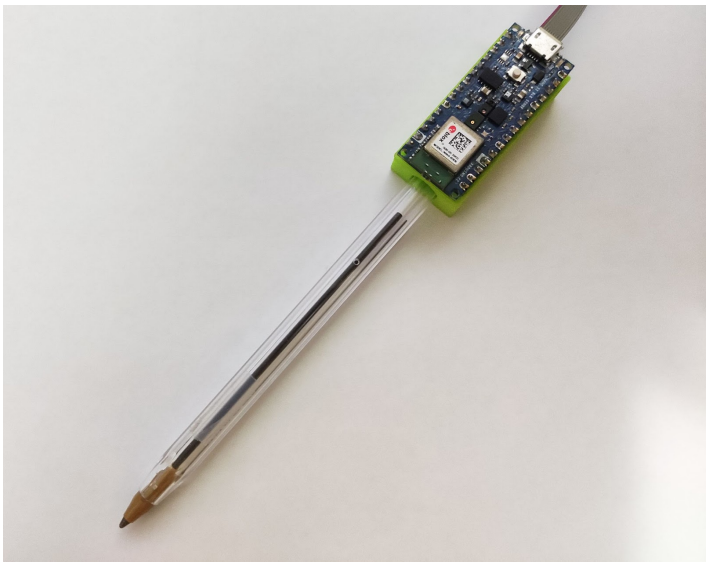

Fig. 11
Ilustración del sistema desarrollado para reconocimiento de escritura

5.2 Obtención de datos

Desafortunadamente no existen set de datos públicos para esta tarea, por lo que se ha confeccionado de forma manual.

En la primera iteración del modelo se pretendía verificar que, efectivamente, el método que se propone es realmente factible y que los datos de la aceleración realmente contienen la información necesaria para realizar las predicciones. Para ello, se procedió a tomar 211 muestras para las primeras seis letras del abecedario en mayúsculas (A, B, C, D, E, F). Es decir un total de 2166 ejemplos. Para estudiar los datos y comprobar si un modelo supervisado convenientemente entrenado podrá distinguirlos, suelen utilizarse técnicas no supervisadas. En este caso se ha realizado una visualización UMAP en tres dimensiones para intentar comprender la distribución de los datos, tal y como muestra la Fig. 12. UMAP [24] es una técnica moderna de reducción de dimensiones para visualización de datos que asume que los datos se organizan en nubes de puntos que forman un recorrido claro (*manifolds*).

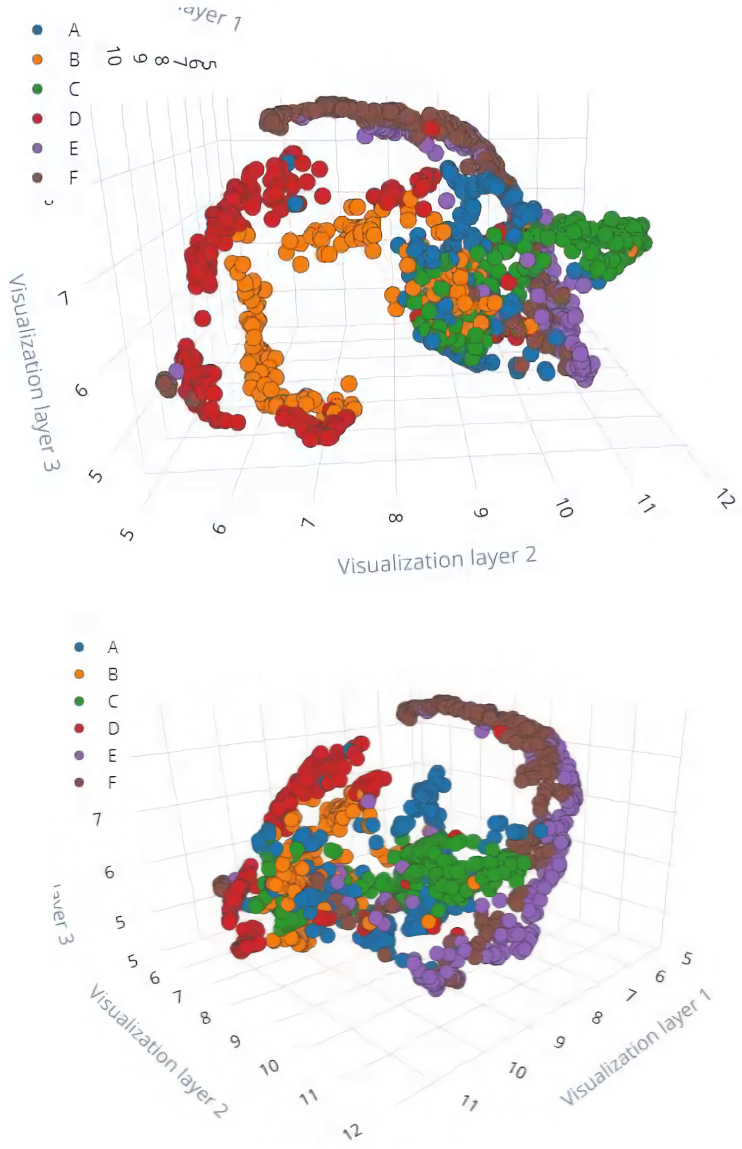

Fig. 12
Visualización UMAP del set de datos del experimento inicial

Se puede observar claramente que cada clase parece formar su propia nube de puntos (*manifolds)*. Por ejemplo, las clases E (morado) y F (marrón) se dibujan de forma similar y su *manifolds* lo demuestran. Si el modelo es capaz de entender esta relación y desdoblar estos recorridos, no debería tener problema en distinguir entre ellas.

Así pues, con el objetivo de validar la propuesta, se propone un simple modelo que utiliza capas convolucionales 1D para encontrar parámetros a lo largo del tiempo. Los resultados obtenidos se pueden observar en la matriz de confusión de la Fig.13.

	A	B	C	D	E	F
A	91.2%	2.9%	5.9%	0%	0%	0%
B	8.1%	89.2%	2.7%	0%	0%	0%
C	2.2%	0%	97.8%	0%	0%	0%
D	4.2%	6.3%	2.1%	87.5%	0%	0%
E	0%	0%	0%	0%	100%	0%
F	0%	0%	0%	0%	12%	88%
F1 SCORE	0.87	0.89	0.95	0.93	0.93	0.94

Fig. 13
Resultados del experimento de validación

No cabe ninguna duda que el modelo está aprendiendo a reconocer las distintas clases. Con estos experimentos iniciales se valida la aplicación propuesta y se procede a la confección de un dataset más grande.

En concreto, se coleccionan 1266 ejemplos para cada clase de las letras de la “A” a la “O” en mayúsculas y con la caligrafía del autor de este documento. Este set de datos es lo suficientemente variado y con clases suficientes para demostrar la viabilidad de escalar el proyecto. Es decir, desarrollamos una prueba de concepto, pues resulta inviable recolectar los suficientes datos de distintas personas a tiempo y presupuesto de este documento. También se forma un set de datos de validación de 185 imágenes por clase.

Cabe destacar que se ha utilizado la plataforma de *Edge Impulse* para la colección de datos y como herramienta a lo largo del desarrollo de la aplicación. Es una nueva herramienta en la industria que hemos comprobado que acelera y simplifica el desarrollo de aplicaciones de aprendizaje automático en sistemas empotrados, que es justamente el objetivo del presente documento. Además, gracias a sus ajustes avanzados no se pierde flexibilidad, pudiéndose escribir el código a gusto del desarrollador para cada bloque de la aplicación, que es lo que se ha realizado.

5.3 Preprocesamiento

Los datos recibidos del sensor pueden contener mucho ruido. Además de muestrear a una frecuencia relativamente baja, resulta esencial filtrar las componentes de alta frecuencia mediante el uso de filtros digitales. Además, los trazos al escribir son normalmente fluidos. Es decir, los cambios bruscos no ayudarán al modelo.

Con una tasa de muestreo de 100 Hz, se propone el uso de un filtro *Butterworth* de paso bajo de segundo orden con una frecuencia de corte de 20 Hz. Es importante tener en cuenta

que es recomendable usar filtros de orden bajo y sencillos, pues estos deberán ser implementados en C/C++ y se ejecutarán dentro del microcontrolador.

Por otra parte y de forma intuitiva, el modelo está interesado en cómo evoluciona la aceleración a lo largo del tiempo, incluyendo la magnitud de estos cambios. Por lo tanto, se ha decidido no normalizar los ejemplos. Sin embargo, con el fin de facilitar el aprendizaje y hacerlo más robusto en cuanto a distintas posturas, se resta la media a lo largo del tiempo para cada eje (X, Y, Z).

Así pues, la Fig.14 muestra un ejemplo en su estado original así como del resultado del preprocesado con dos implementaciones distintas: una utilizando librerías de python que optimicen la latencia a la hora de entrenar aprovechando el cálculo vectorial y otra con operaciones clásicas y de forma manual, con el objetivo de asegurar su posible implementación en C/C++.

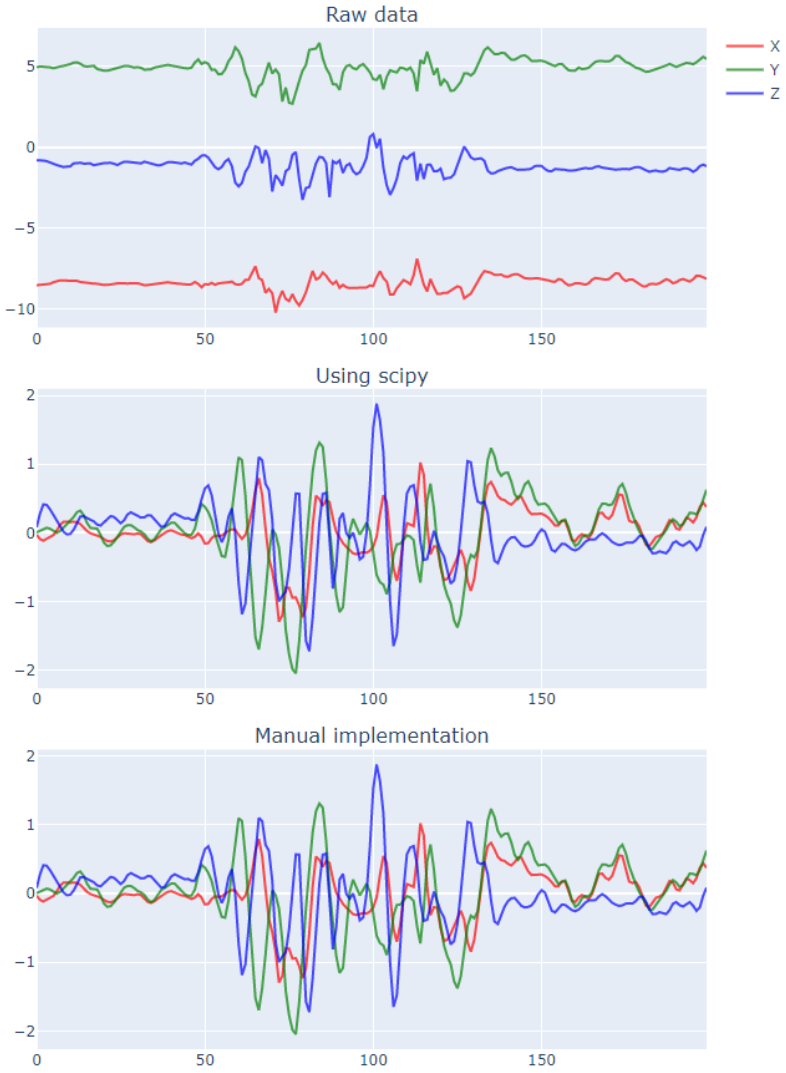

Fig. 14
Ejemplo de una muestra y su resultado tras preprocesamiento.

De la imágen podemos destacar dos cosas:

- Los datos pasan a tener mucho menos ruido de alta frecuencia y además todos los ejes tienen un rango parecido
- Tanto la implementación en *Python* con paquetes de cálculo numérico (útil durante el entrenamiento) como la implementación manual (sin librerías) dan lugar a los mismos resultados. Esto demuestra que la implementación en C/C++ de este flujo de preprocesamiento es muy factible.

La ventana de muestreo es de dos segundos, tiempo más que suficiente para dibujar cualquier caracter.

5.4 Selección de modelo

Cuando se trabaja con datos que varían con el tiempo, como los datos del acelerómetro, vienen a la mente redes neuronales recurrentes o células LSTM [6]. Sin embargo, estos modelos pueden resultar en costes computacionales muy altos y, además, no están soportados por *TFLite Micro*, que es de nuevo la librería que se utilizará para interpretar el modelo comprimido en C/C++.

Durante la fase de validación del experimento se usó un modelo con capas convolucionales que obtuvo muy buenas métricas. A modo de visualización, se pueden imaginar los datos del acelerómetro como una imagen de altura uno, ancho igual al número de muestras y tantos canales como ejes (X, Y, Z). Si elegimos el tamaño de los filtros también con altura uno, el kernel se desplazará a lo largo del eje temporal, pudiendo modelar patrones que varían con el tiempo, que es exactamente lo que se busca.

Así pues, la arquitectura que se propone es muy sencilla: capas convolucionales de forma consecutiva y *max*-pooling, con capas más profundas teniendo más filtros y menos resolución temporal, seguida de una capa densa para la clasificación. La Fig. 15 muestra el diagrama del modelo.

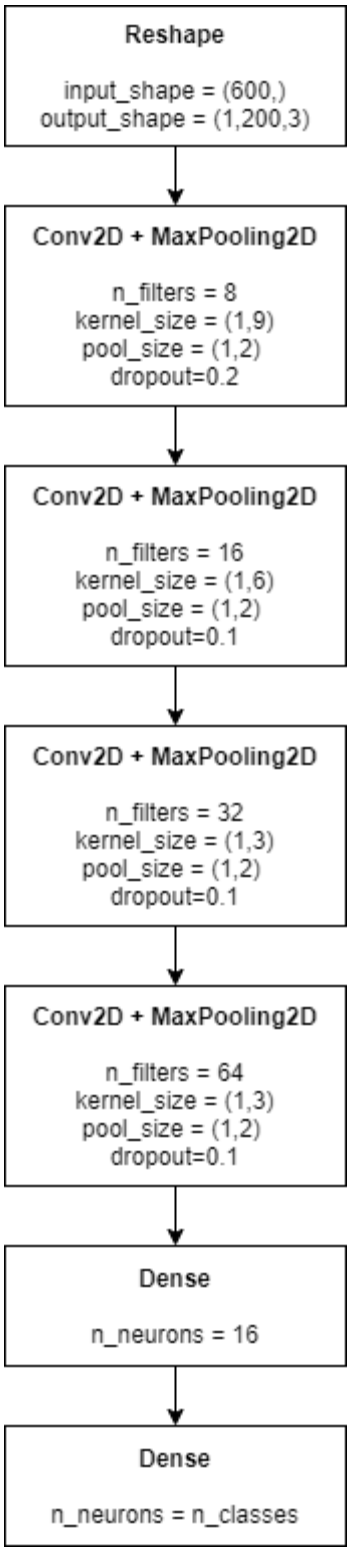

Fig. 15
Arquitectura del modelo

Se ha escogido un modelo tan sencillo principalmente por dos motivos:

- Con el set de datos tan reducido que se posee, un modelo más grande puede resultar contraproducente y sobreajustar (*overfit)* los ejemplos.

- El microcontrolador en el que se va a utilizar es muy modesto y la naturaleza de la aplicación requiere ser ejecutada de forma continua en tiempo real para reconocer los caracteres de forma fiable.

La Fig. 16 muestra la matriz de confusión obtenida en el conjunto de *test* y el Valor-F1 para cada clase.

Fig. 16
Resultados en el conjunto de test

El rendimiento es excelente y muestra qué clases le cuesta al modelo diferenciar, que además coincide con la experiencia en el prototipo. La letra “G” es la clase más subrepresentada y queda clasificada como “C”. Otra clase subrepresentada en el prototipo y que no es demasiado notable en esta matriz es la letra “H”, que queda clasificada como la letra “A” en algunas ocasiones a no ser que sus trazos sean claramente definidos.

5.5 Optimización y compresión del modelo

Como se ha mencionado anteriormente, se utiliza la plataforma de ***Edge Impulse*** para el desarrollo de la aplicación. Una de sus funcionalidades es la compresión del modelo, que goza de las optimizaciones de un compilador desarrollado por la empresa llamado *EON Compiler*, pudiendo reducir hasta en un 50% la memoria requerida.

De nuevo se selecciona la discretización de los parámetros del modelo a enteros de 8 bits para reducir el coste computacional y ocupación de memoria. Sin embargo, por comodidad la entrada y salida del modelo se han dejado en float de 32 bits, que es el formato con el que se captura la aceleración.

La plataforma también nos indica unas estimaciones relativas a la ocupación de memoria (bytes) del modelo, resumidas en la Tabla 4.

RAM	ROM	Latencia	Exactitud
6.9 kB	52.8 kB	65 ms	95.83%

Tabla 4. Estimaciones de rendimiento ofrecidas por Edge Impulse

Como comprobaremos en las siguientes secciones, se asemejan a los resultados obtenidos. Esta información es de mucha utilidad para saber de antemano si la aplicación cumple con los requerimientos.

5.6 Despliegue de la aplicación en nRF52840

5.6.1 Primeras consideraciones

El microcontrolador que se utilizará para esta aplicación es el **nRF52840**, de *Nordic Semiconductors*, con una frecuencia de reloj de 64 MHz, 1 MB de memoria flash y 256 kB de memoria RAM. Unas especificaciones muy modestas que lo convierten en un microcontrolador de gama media y a un coste por debajo de los cinco euros. Además, el nRF52840 está equipado con tecnología *bluetooth* de bajo consumo (*BLE: Bluetooth Low Energy),* que se utilizará para enviar las predicciones.

En concreto, se utilizará la placa de desarrollo **Arduino Nano 33 BLE Sense**, pero **no** se utilizarán las librerías ni el entorno de Arduino, con el objetivo de tener más flexibilidad y tener la capacidad de crear una prueba de concepto escalable para una posible aplicación comercial. Esta placa es una fantástica herramienta dada la amplia variedad de sensores incrustados, entre ellos un módulo inercial de nueve ejes LSM9DS1, de donde se obtendrán las aceleraciones lineales en los ejes X, Y y Z.

La Fig.17 muestra una imágen de esta placa de desarrollo, que ya contiene el acelerómetro que se necesita para la aplicación.

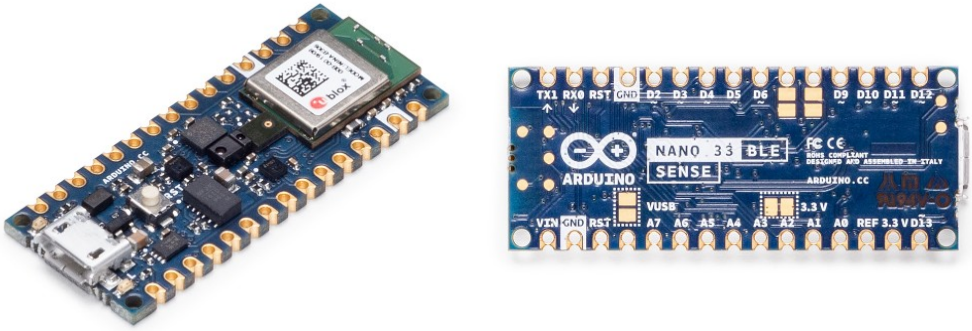

Fig. 17
Arduino Nano 33 BLE Sense

El único inconveniente es que como no se van a usar las librerías de Arduino, se deberá programar mediante un depurador SWD. Desafortunadamente, los pines SWD de la placa se encuentran en la cara inferior en forma de *pads* y se han debido soldar unos cables que incomodan ligeramente el uso del prototipo.

La implementación de la aplicación requiere de un sistema operativo de tiempo real, de los que se ha escogido *Mbed OS*, ya que es profesional, escalable, flexible y tiene un gran soporte en la familia de microcontroladores de la familia nRF, entre otras (que utilicen procesadores Cortex-M).

Como Mbed no es solo un sistema operativo de tiempo real sino un *framework* que impulsa el desarrollo de aplicaciones IoT en sistemas empotrados, deberemos usar un depurador compatible con el protocolo código abierto de CMSIS-DAP para comunicarse con el microcontrolador. Se propone el uso del *Particle Debugger* por su coste económico y su posible uso en diferentes placas(Fig.18).

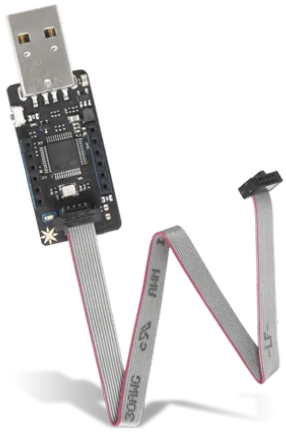

Fig. 18
Particle debugger con soporte CMSIS-DAP e interfaz SWD

Con el objetivo de simplificar y facilitar el desarrollo, se usará el entorno de desarrollo *Mbed Studio* (Fig. 19), que es similar a Visual *Studio Code*, pero con herramientas que facilitan el uso del sistema operativo de tiempo real *mbed OS* y librerías compatibles.

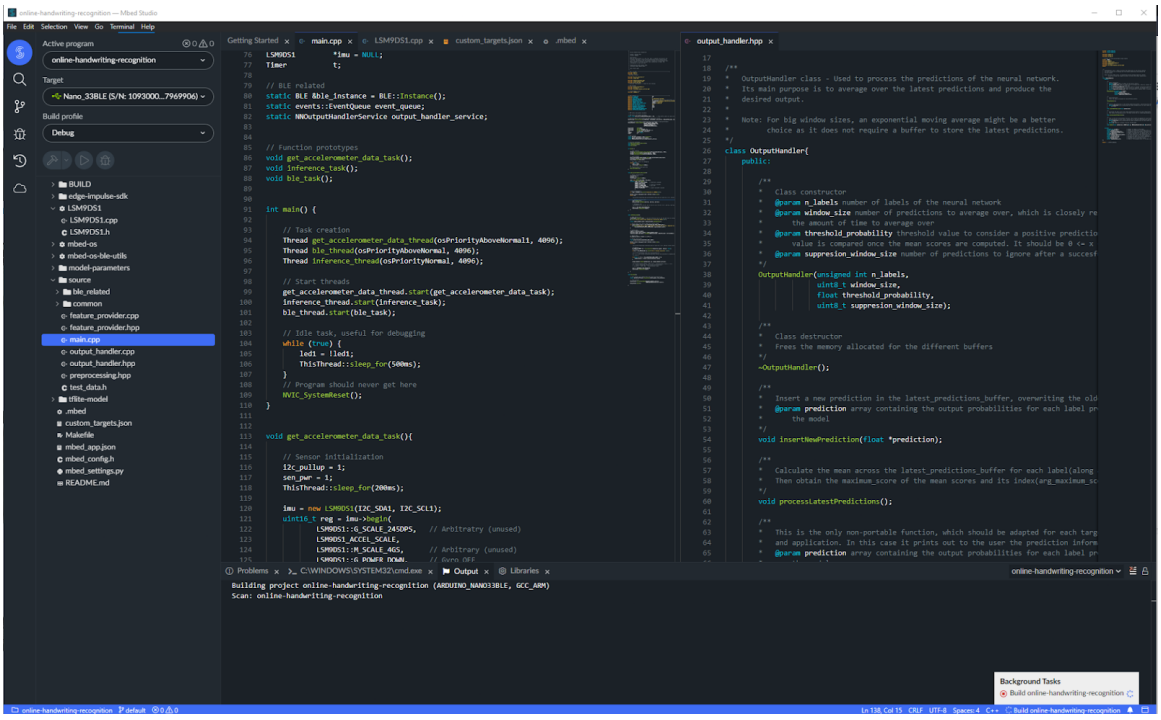

Fig. 19
Entorno de desarrollo Mbed Studio

Desafortunadamente, el proceso de flash con la herramienta de *mbed* resultaba anormalmente alto, por lo que se usó *J-Flash Lite* para cargar el programa en la memoria del microcontrolador.

5.6.2 Generación del paquete de desarrollo con Edge Impulse

Entre las diferentes herramientas que ofrece la plataforma de *Edge Impulse*, quizá la más interesante sean las opciones de despliegue. Entre ellas la creación de una librería en C++, para Arduino, para *X-CUBE-AI*, *WebAssembly* o *TensorRT*.

El caso que interesa para esta aplicación es la generación de una librería en C++. Los resultados son muy intuitivos, se genera una carpeta llamada `edge-impulse-sdk` que contiene el paquete de desarrollo que envuelve a *TFLite Micro* y proporciona una API muy sencilla. También se genera una carpeta con el modelo y un archivo principal (*main*) con un programa esqueleto muy útil.

La única parte del *pipeline* (flujo de proceso) que se ha de implementar por el desarrollador es el preprocesado en el caso de que se haya elegido "personalizado", que es el caso.

Mientras se investigaba el kit de desarrollo de *Edge Impulse* se encontró una pseudo-implementación del paquete de Python *numpy* y de *scipy*, pero en C++. El uso de estas funciones simplifica, optimiza y hace más legible el código de preprocesamiento. Por ejemplo, existen funciones como `numpy::mean_axis0` o `numpy::transpose` haciendo trivial trabajar de forma vectorizada.

Una vez implementado el bloque de preprocesamiento y verificado el funcionamiento del modelo en el microcontrolador (por ejemplo con una entrada constante), se puede proceder a construir la aplicación.

5.6.3 Implementación de la clasificación en tiempo real

Desarrollar un flujo de datos (*pipeline)* eficiente que tome muestras y clasifique los datos en tiempo real puede ser un desafío. Se recuerda que se dispone del código en Github [3] y que se ha documentado extensivamente para ser compatible con herramientas de generación de APIs de referencia como *Doxygen*.

Para entender el funcionamiento de la aplicación, se presenta el diagrama de flujo de la Fig. 20 y se explican las consideraciones y bloques:

- Se requiere por lo menos de dos tareas. Una se encarga de de muestrear el valor del acelerómetro a una frecuencia de muestreo constante, que se lleva a una cola (array circular). La otra es la tarea encargada de realizar inferencia, debiendo recoger de la cola los dos últimos segundos de datos, preprocesarlos y ejecutar la red neuronal para obtener sus predicciones.
- Para guardar los datos, la clase `DataProvider` contiene una implementación simple de una cola con carga serie y descarga en paralelo, asegurándose además de que no existen conflictos al ser accedida por dos tareas. Esto se consigue simplemente haciendo la cola más larga que lo estrictamente necesario, así cuando la tarea lee los datos para ejecutar la red neuronal, la tarea de muestreo guarda los nuevos valores en unos huecos de memoria que no están siendo usados más que por esta tarea.
- El anterior punto da a entender que entonces la red se ejecuta cada cierto periodo de tiempo (se ha elegido 150 ms) y que entonces se generan predicciones de forma continua.
- Para lidiar con el continuo flujo de predicciones, se han de filtrar estos resultados. La clase `OutputHandler` computa la media de las puntuaciones para cada clase para las últimas `window_size` predicciones. Cuando la media de un valor supera un umbral de probabilidad y es la mayor de entre todas clases, se obtiene una predicción válida.
- Cuando se realiza una predicción válida, el modelo espera `suppresion_window_size` ejecuciones antes de poder considerar otra nueva. Esto asegura que no se considere el mismo evento (misma letra) como válido más de una vez.

- En el caso de este prototipo se envían las predicciones válidas a un servidor BLE (*Bluetooth Low Energy)* con un servicio personalizado. Por lo tanto, se crea otra tarea que gestione todo el *stack* de comunicaciones

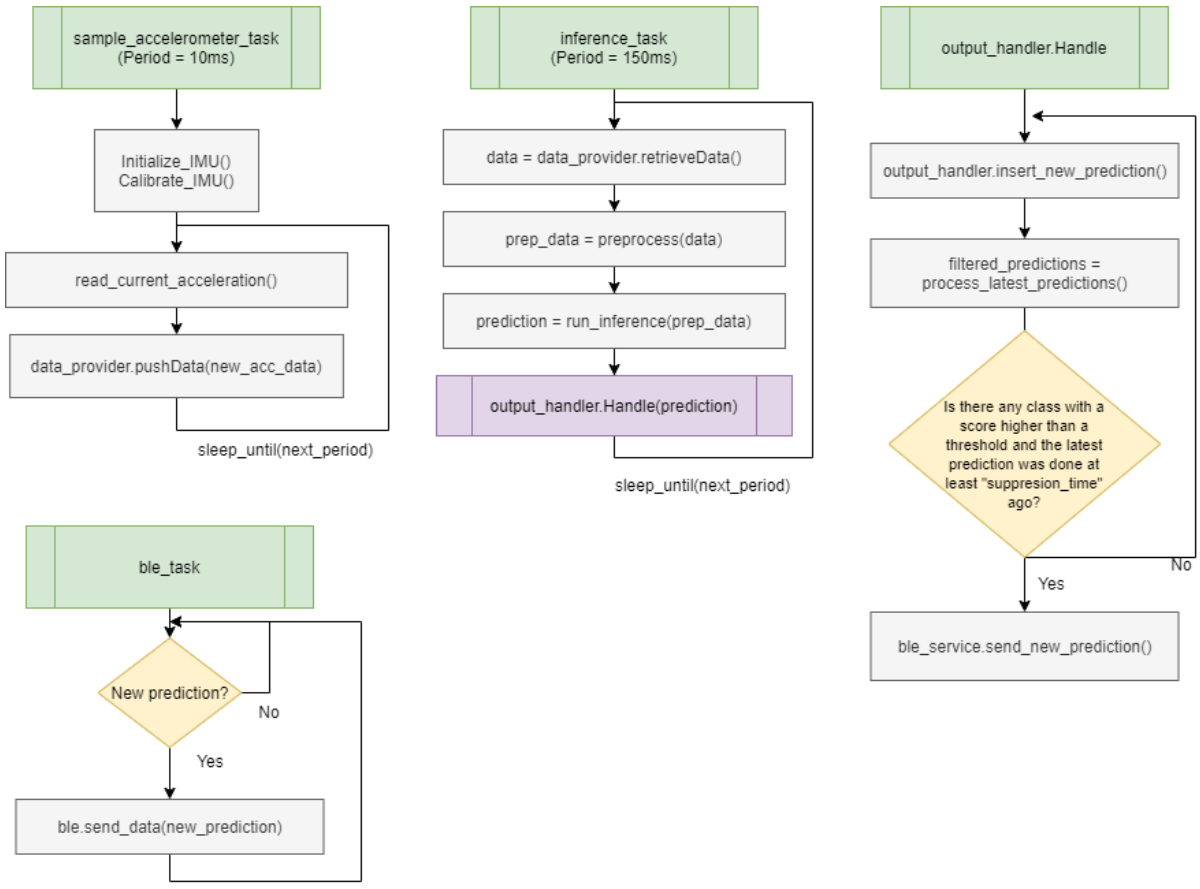

Fig. 20
Diagrama de flujo del funcionamiento de la aplicación

Las constantes como el periodo de ejecución de la red neuronal, el tamaño de ventana para mediar predicciones y otras se han obtenido de forma empírica.

El servidor *BLE* podría ser una aplicación en un dispositivo móvil, pero como prueba de concepto se ha programado en una pequeña *Raspberry Pi Zero* con una pantalla montada. El resultado es el de la Fig.19.

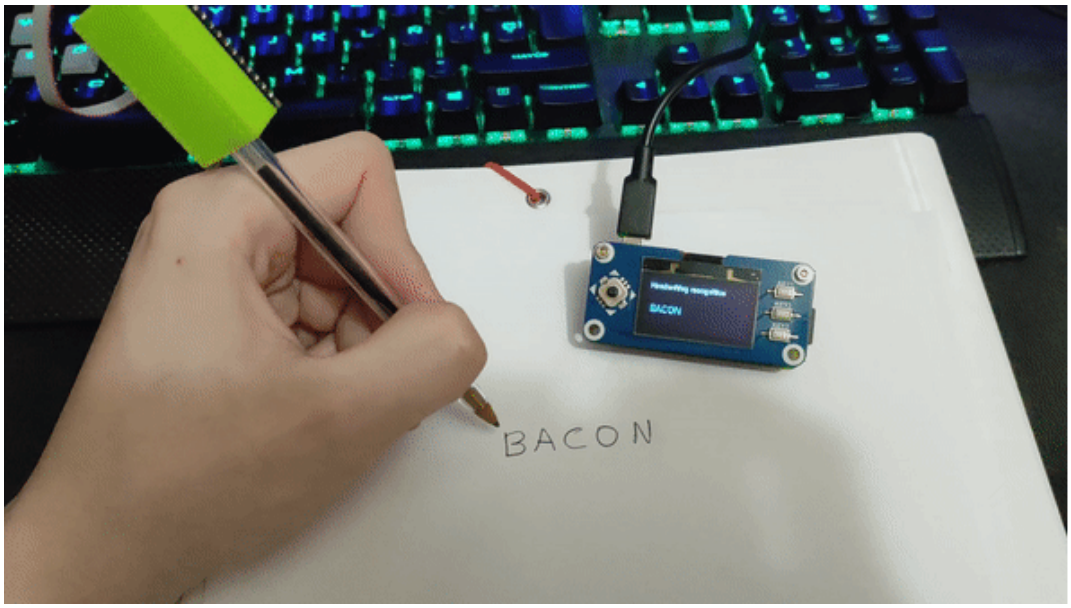

Fig. 19
Demostración de funcionamiento

En la imágen se aprecia el resultado de haber escrito la palabra “BACON”, cuyos caracteres quedan escritos en la pequeña pantalla. La aplicación web [4] contiene el vídeo completo.

Como se ha mencionado anteriormente, la letra “G” queda clasificada como “C” y la “H” como “A” en ciertas ocasiones. Por lo demás, el prototipo funciona de manera satisfactoria, clasificando en tiempo real los caracteres dibujados, con solo el uso de un acelerómetro y un microcontrolador sencillo.

El bloque de preprocesamiento tarda en ejecutarse alrededor de 10 ms. Por su parte, el modelo tiene una latencia de alrededor de 65 ms, muy parecido a las estimaciones que proporciona *Edge Impulse.* El modelo ocupa nada más que 52.8 kB.

5.7 Conclusion

Durante esta sección se ha explorado el desarrollo de una aplicación de reconocimiento de escritura mediante acelerómetro, método distinto a los tradicionales, que se suelen apoyar en procesamiento de imagen. El foco de la aplicación se ha puesto en dos partes. En primer lugar, la confección del conjunto de datos y la validación del experimento, que resultan vitales antes de poder continuar. En segundo lugar, se ha hecho hincapié en la implementación de la aplicación para que sea capaz de ejecutarse en tiempo real, teniendo en cuenta el flujo constante de predicciones y de muestras y filtrándolas acordemente.

Los resultados son en general muy satisfactorios, aunque no resulta sorprendente que el funcionamiento deja que desear cuando lo prueban otras personas con otra caligrafía. Esto

puede ser perfectamente solventado si se dispusiese de un gran dataset en el que han participado numerosas distintas personas.

De todos modos, queda comprobado que este método de reconocer la escritura es posible, donde los costes y el consumo de potencia son bajos gracias al uso de un microcontrolador, y no resulta intrusivo ni incómodo gracias al pequeño perfil de estos dispositivos. En conclusión, se ha desarrollado una prueba de concepto fiable y con una gran infraestructura en términos de código que indican potencialmente la escalabilidad para desarrollar un producto comercial o industrial.

6. Conclusiones y trabajo futuro

El desarrollo de aplicaciones basadas en redes neuronales y aprendizaje profundo en sistemas empotrados implementadas en microcontroladores o "*TinyML*" [27] es un sector que está en constante crecimiento e investigación. Sus obvias ventajas como el coste ajustado, bajo consumo o privacidad de los datos las hacen atractivas en la industria para desarrollar a gran escala.

Hasta hace poco el desarrollo de este tipo de aplicaciones era totalmente manual, proceso largo, tedioso y costoso (tiempo y dinero), que requería de grandes conocimientos de aprendizaje automático, redes neuronales y programación. Desde hace unos años han ido apareciendo herramientas que permiten cierta automatización del proceso de desarrollo. Si a esto unimos que hoy en día se dispone de microcontroladores de 32 bits relativamente potentes, pero muy baratos (como los usados en este TFG), se entiende que el abanico de posibilidades de aplicación real comercial e industrial de estos sistemas se amplíe enormemente.

Así, el objetivo fundamental de este trabajo era estudiar este incipiente campo del TinyML, aprendizaje automático en microcontroladores de prestaciones, coste y consumo reducidos, analizando las herramientas y microcontroladores actualmente disponibles. Este estudio se ha llevado a cabo mediante el desarrollo de dos aplicaciones: la clasificación de imágenes de personas y/o coches para control de tráfico y el reconocimiento *online* de escritura a mano mediante el procesamiento de los datos de un acelerómetro incorporado en un bolígrafo. Ambas aplicaciones, orientadas al procesamiento de datos generados por dos tipos de sensores típicos, pero muy diferentes (cámara y acelerómetro), nos ha permitido valorar las herramientas actualmente disponibles, comprobando que, efectivamente, facilitan el desarrollo e implantación de redes neuronales y deep learning en aplicaciones hasta ahora vetadas a personal no experto.

De esta manera, en nuestro caso, hemos podido desarrollar dos aplicaciones de cierta envergadura y con buenos resultados, que anteriormente requerirían meses de trabajo por un equipo de especialistas. Por ejemplo, en el primer caso hemos construido sobre un

microcontrolador STM32H7 de ST un clasificador de imágenes en tiempo real procedentes de una micro cámara OV7670 que requiere tan solo 345 KB y un segundo aproximado de latencia (tiempo de procesamiento total). En este caso se han estudiado herramientas como el optimizador *TFLite* de Google y *X-CUBE-AI* de ST. Una conclusión importante es que el optimizador *TFLite* es sencillo, pero aún es limitado y requiere de futuras mejoras.

En el segundo caso, reconocimiento de caracteres escritos, se ha usado un microcontrolador nRF52840 de *Nordic Semiconductors* incorporado en la placa de desarrollo Arduino Nano 33 BLE Sense y, entre otras herramientas, se ha utilizado la plataforma online de desarrollo Edge Impulse (https://www.edgeimpulse.com/), comprobando que acelera y simplifica el desarrollo de aplicaciones de aprendizaje automático en sistemas empotrados. El modelo final requiere unos 53 KB y requiere unos 65 ms tan solo.

Cabe destacar que la librería *TFLite Micro* automatiza gran parte del trabajo de optimizar la red y ofrecer las herramientas C/C++ para integrar en la aplicación. Sin embargo, es el desarrollador el encargado de asegurarse que el modelo cumple con los requisitos de rendimiento en cuanto a métricas y de que es capaz de ejecutarse en unas condiciones óptimas de latencia, así como de verificar que el código y parámetros caben en la memoria flash o RAM del microcontrolador elegido.

Por otro lado, la plataforma *Edge Impulse* facilita todo el flujo del desarrollo de aplicaciones de *TinyML,* desde la recolección de datos hasta lo que consideramos su punto más fuerte, el despliegue de la aplicación en distintos formatos (por ejemplo librería en C++) y una API muy intuitiva. Además gracias al *EON Compiler*, se puede conseguir reducir aún más la ocupación de memoria de la aplicación.

Prueba de la viabilidad y la utilidad de estos entornos son las dos aplicaciones que se han desarrollado. Dos ámbitos distintos que pueden aprovechar el uso de estas nuevas tecnologías para solucionar problemas o mejorar las soluciones actuales. Además, ambas se han desarrollado con el concepto de escalabilidad en mente, donde un gran set de datos y de calidad podría beneficiar enormemente a los modelos entrenados.

En un futuro y con el constante crecimiento del internet de las cosas (IoT) [25] se espera que esta industria siga manteniendo un auge constante y se aplique a distintos sectores y problemas. TinyML tiene grandes ventajas frente a las aplicaciones tradicionales basadas en el aprendizaje profundo y tiene grandes oportunidades de crecer, como por ejemplo en visión por computador o mantenimiento predictivo [26].

En relación al trabajo futuro, y como se ha mencionado en numerosas ocasiones en el documento, sería fundamental conseguir conjuntos de datos más grandes y especializados, lo que potenciaría el rendimiento y capacidad de generalización de las aplicaciones desarrolladas. Somos conscientes de ello, pero el limitado tiempo disponible para el desarrollo del TFG ha llevado a utilizar conjuntos de datos que, obviamente, no son todo lo completas como sería deseable para una aplicación real.

En cuanto a la primera aplicación se refiere, un trabajo muy interesante a realizar sería la estimación del tráfico y el algoritmo de control de los semáforos en función de ésta; resultaría muy interesante estudiar hasta qué punto puede suponer un ahorro en combustible y emisiones de CO2.

En cuanto al reconocimiento de la escritura, quedaría escalar la aplicación para hacerla más robusta en cuanto a distintas caligrafías de personas diversas. Además, se propone que sea otra red neuronal la que reciba las predicciones del primer modelo y genere las palabras que se están intentando escribir, haciendo el sistema más fiable y versátil; se habla de un modelo del lenguaje.

Del mismo modo, se pretende seguir realizando aplicaciones en el sector del *TinyML* para distintos sectores y microcontroladores, con el fin de desarrollar el ecosistema y mejorar las técnicas de optimización de éstas.

Referencias

1. Goodfellow, Ian, Bengio, Yoshua y Courville, Aaron. Deep Learning. MIT Press. 2016.
2. Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, Dmitry Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. 15 de Diciembre de 2017
3. Phan Razquin, Enrique. https://github.com/PHANzgz/embedded-deep-learning
4. Phan Razquin, Enrique. https://share.streamlit.io/phanzgz/embedded-deep-learning/demo_webapp/app.py
5. Warren S. McCulloch, Walter Pitts. A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics. 1943
6. Aurelien Geron. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems 2nd Edition. O'Reilly Media, Inc. Septiembre 2019. ISBN: 9781492032649
7. Niall O' Mahony, Sean Campbell, Anderson Carvalho, Suman Harapanahalli, Gustavo Velasco Hernandez, Lenka Krpalkova, Daniel Riordan, Joseph Wals. Deep Learning vs. Traditional Computer Vision. IMaR Technology Gateway. 30 de Octubre de 2019
8. Jens Kober, J. Andrew Bagnell, Jan Peters. Reinforcement Learning in Robotics: A Survey. 23 de Agosto de 2013
9. John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Kathryn Tunyasuvunakool, Olaf Ronnebergerø*, Russ Bates, Augustin Žídek, Alex Bridgland, Clemens Meyer, Simon A A Kohl, Anna Potapenko, Andrew J Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reimanø, Martin Steineggerù, Michalina Pacholska, David Silver, Oriol Vinyals, Andrew W Senior, Koray Kavukcuoglu, Pushmeet Kohliø, Demis Hassabis. AlphaFold 2. Deepmind. 2020
10. Alejandro Baldominos, Iván Blanco, Antonio José Moreno 2, Rubén Iturrarte, Óscar Bernárdez, Carlos Afonso. Identifying Real Estate Opportunities Using Machine Learning. MDPI. 17 de Octubre de 2018

11. Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, Li Fei-Fei. ImageNet: A large-scale hierarchical image database. IEEE. 25 de Junio de 2009

12. ElephantEdge competition. Hackster.io. 2020

13. Olivier Debauche, Mahmoudi Saïd, Meryem Elmoulat, Sidi Ahmed Mahmoudi. Edge AI-IoT Pivot Irrigation, Plant Diseases and Pests Identification. Procedia Computer Science. Noviembre 2020

14. Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hay,s Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, Piotr Dollar. Microsoft COCO: Common Objects in Context. Microsoft. 1 de Mayo de 2014

15. Jonathan Krause, Michael Stark, Jia Deng, Li Fei-Fei. 3D Object Representations for Fine-Grained Categorization. 4th IEEE Workshop on 3D Representation and Recognition, at ICCV 2013 (3dRR-13). Sydney, Australia. 8 de Diciembre de 2013.

16. A. Quattoni, and A.Torralba. Recognizing Indoor Scenes. IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2009.

17. Mingxing Tan, Quoc V. Le. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. 28 de Mayo de 2019

18. Barret Zoph, Vijay Vasudevan, Jonathon Shlens, Quoc V. Le. Learning Transferable Architectures for Scalable Image Recognition. 21 de Julio de 2017

19. Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. Google Inc. 17 de Abril de 2017.

20. Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. Google Inc. 13 de Junio de 2018.

21. Ashref Maiza. The Unknown Benefits of using a Soft-F1 Loss in Classification Systems. 4 de Diciembre de 2019.

22. Noman Islam, Zeeshan Islam, Nazia Noor. A Survey on Optical Character Recognition System. 3 de Octubre de 2017.

23. Ian J. Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, Vinay Shet. Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks. Google Inc. 20 de Diciembre de 2013.

24. Leland McInnes, John Healy, James Melville. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. 9 de Febrero de 2018.

25. The global Internet of Things market is projected to grow from $381.30 billion in 2021 to $1,854.76 billion in 2028 at a CAGR of 25.4% in the 2021-2028. Mayo de 2021.

26. Yongyi Ran, Xin Zhou, Pengfeng Lin, Yonggang Wen, Ruilong Deng. A Survey of Predictive Maintenance: Systems, Purposes and Approaches. 12 de Diciembre de 2019.

27. Pete Warden, Daniel Situnayake. TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers. O'Reilly Media, Inc. Diciembre 2019. ISBN: 9781492052043