ULPGC
Universidad de
Las Palmas de
Gran Canaria

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Desarrollo de una plataforma para reconocimiento de gestos basada en Tensor Flow Lite sobre el dispositivo IoT Argon de Particle

Titulación: Grado en Ingeniería en Tecnologías de la Telecomunicación
Mención: Sistemas Electrónicos
Autor: Tomás Real Rueda
Tutores: D. Valentín De Armas Sosa
D. Félix B. Tobajas Guerrero

Fecha: Septiembre 2021

AGRADECIMIENTOS

"A mi familia por apoyarme sin condiciones durante este largo trayecto, porque siempre han estado y estarán a mi lado. A todos y cada uno de los amigos que he hecho durante esta etapa de mi vida, porque sin ellos el camino hubiera sido muy diferente. A mis amigos de siempre, los de toda la vida, porque han sido el refugio donde acudir cuando lo he necesitado. A mis tutores Valentín y Félix, porque han sabido comprender la etapa tan complicada que estamos viviendo y no han dejado de motivarme cuando lo necesitaba, porque ser un ejemplo a seguir. Y, por último, agradecer especialmente a mi compañero y amigo Jorge Fernández, porque cada paso que hemos dado lo hemos dado juntos y siempre hemos estado el uno para el otro, porque hubiera sido imposible, o por lo menos mucho más complicado, terminar esta etapa sin él al lado."

RESUMEN

En los últimos años, el concepto de *Internet of Things* (IoT) ha adquirido una gran relevancia en la vida de miles de personas, ya sea de manera directa o indirecta. La razón principal es el trascendental cambio que propicia en la calidad de vida de las personas, gracias a los distintos métodos de acceso a datos o los novedosos servicios que ofrece. Además, el concepto de Inteligencia Artificial (IA) ha ido convergiendo en los últimos años con IoT, desarrollándose aplicaciones orientadas al uso de sensores, actuadores, interfaces o sistemas empotrados utilizando algoritmos o técnicas de IA con procesamiento *"on the edge"*.

La base teórica del presente TFG viene dada, principalmente, por estos dos conceptos y su convergencia, lo que actualmente se conoce como *Artificial Internet of Things* (AIoT).

Su desarrollo consiste en la implementación de una aplicación orientada a la detección de gestos, concretamente de letras del alfabeto dactilológico de la lengua española de signos, basada en el algoritmo de IA *Tensor Flow Lite*. Para ello se ha implementado un modelo inicial, así como su aplicación, para el reconocimiento de tres gestos y, a partir de éste, se ha desarrollado el modelo final que cumple con el objetivo principal de reconocer un conjunto de letras del alfabeto dactilológico.

ABSTRACT

In recent years, Internet of Things (IoT) concept has acquired great relevance in the lives of thousands of people, either directly or indirectly. The main reason is the transcendental change it brings about in people's quality of life, thanks to the different methods of accessing data or the innovative services it offers. In addition, the concept of Artificial Intelligence (AI) has been converging in recent years with IoT, developing applications oriented to the use of sensors, actuators, interfaces or embedded systems using AI algorithms or techniques with "on the edge" processing.

The theoretical basis of this TFG is mainly given by these two concepts and their convergence, which is currently known as Artificial Internet of Things (AIoT).

Its development consists of the implementation of an application oriented to the detection of gestures, specifically letters of the Spanish sign language dactylological alphabet, based on the AI algorithm Tensor Flow Lite. For this purpose, an initial model has been implemented, as well as its application, for the recognition of three gestures and, from this, the final model has been developed, which fulfills the main objective of recognizing a set of letters of the dactylological alphabet.

ÍNDICE DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE FÓRMULAS

ACRÓNIMOS

ADC *Analogic to Digital Converter*

AIOT *Artificial Internet of Things*

ASCII *American Standard Code for Information Interchange*

API *Aplication Programming Interfaces*

CNN *Convolutional Neural Network*

COITT *Colegio Oficial de Ingenieros Técnicos de Telecomunicación*

EITE *Escuela de Ingeniería de Telecomunicación y Electrónica*

GPIO *General Purpose Input/Output*

HW *Hardware*

IA *Inteligencia Artificial*

I2C *Inter-Integrated Circuit*

IGIC *Impuesto General Indirecto Canario*

IoT *Internet of Things*

LED *Light Emitting Diode*

MCU *MicroController Unit*

ML *Machine Learning*

PC *Personal Computer*

PWM *Pulse-Width Modulation*

RAM *Random Access Memory*

ReLU *Rectified Linear Unit*

RGB *Red Green Blue*

SPI *Serial Peripheral Interface*

SW *Software*

TF *Tensor Flow*

TFG *Trabajo Fin de Grado*

UART *Universal Asynchronous Receiver-Transmitter*

ULPGC *Universidad de Las Palmas de Gran Canaria*

USB *Universal Serial Bus*

MEMORIA

CAPÍTULO 1. Introducción

1.1 Antecedentes

El concepto de IoT (*Internet of Things*) o Internet de las Cosas es de sobra conocido por prácticamente cualquier persona que tenga relación con la Tecnología, ya sea de manera directa o indirecta. Por lo general, IoT se basa en una combinación de sensores y actuadores, dispositivos con conectividad a Internet, capaces de transmitir y recibir información para ser utilizada por distintos servicios o usuarios, si bien lo realmente importante de este concepto, relativamente nuevo, es el transcendental cambio que puede propiciar en la calidad de vida de las personas, gracias a los distintos métodos de acceso a datos o a los novedosos servicios que ofrece; desde la amplia distribución de la red, hasta redes locales inteligentes o servicios personalizados. Algunas de las principales aplicaciones de IoT son las ciudades inteligentes, aplicaciones enfocadas a la educación, automoción, agricultura, etc... [1]

Asimismo, otro concepto a tratar en este Trabajo Fin de Grado (TFG) es el de Inteligencia Artificial (IA). J. Pascual en [2]**,** lo define de forma simple como *"el intento de imitar la inteligencia humana usando un robot o software",* aunque bien es cierto que no existe una definición exacta, ya que es una ciencia muy cambiante y experimental, siendo necesario además, definir en primer lugar con exactitud lo que es la inteligencia humana. No se puede hablar de inteligencia artificial sin antes mencionar a una de las grandes figuras del mundo de la tecnología, Alan Turing, quien marcó el inicio de este concepto con el Test de Turing, que se basaba en identificar, mediante un interrogador, si el que responde a la pregunta es un humano o una máquina. Años más tarde, otro hito marcaría un antes y un después en el desarrollo de IA, el ordenador *Deep Blue* de IBM fue capaz de vencer en una partida de ajedrez al mejor jugador de la historia, Gary Kaspárov.

El funcionamiento de IA, de manera simplificada, se basa en tres pasos:

- **Aprendizaje**, aprender a realizar una tarea.
- **Entrenamiento**, donde se corregirán los fallos hasta que se consiga un rendimiento óptimo.
- **Resultados**, cuando la IA será capaz de operar de manera autónoma, recibiendo los datos de entrada y generando los resultados necesarios [2].

Actualmente, es fácil vislumbrar que las diferentes técnicas de IA se han ido desarrollando con un enfoque claro hacia la convergencia con Internet de las Cosas, lo cual implica que se han ido desarrollando aplicaciones orientadas al uso de sensores, actuadores, interfaces o sistemas empotrados utilizando algoritmos o técnicas de Inteligencia Artificial, y es lo que conforma el nuevo concepto de Inteligencia Artificial de las Cosas (AIoT), base de este TFG [3].

El algoritmo utilizado en este caso será *Tensor Flow*, desarrollado por Google y orientado a aplicaciones de *Deep Learning*, área específica dentro de *Machine Learning*. Siendo más específico, se utilizará *Tensor Flow Lite*, que permite a los desarrolladores ejecutar el algoritmo *Tensor Flow* en dispositivos móviles, empotrados o IoT. Además, cabe destacar que está específicamente diseñado para procesamiento "on *edge*" o en el borde de la nube, es decir, procesamiento local, evitando así hacer uso de servidores remotos a los que enviar y recibir información. En este sentido, los beneficios fundamentales con respecto a otros algoritmos de procesamiento en nube son [4]:

- **Latencia**, no hay que enviar o recibir información de un servidor, con el tiempo que esto conlleva.
- **Privacidad**, no se envían datos a la nube, reduciendo así las posibilidades de *hackeo*.
- **Conectividad**, no se necesita conexión a Internet.
- **Consumo de potencia**, considerablemente inferior frente al del procesamiento en la nube.

El presente TFG consiste en implementar una aplicación orientada a la detección de gestos basada en el algoritmo de Inteligencia Artificial anteriormente mencionado, *Tensor Flow Lite*, demostrando así las ventajas del procesamiento local *"on edge"* frente al procesamiento en la nube. El *hardware* a utilizar será el dispositivo IoT *Argon* de *Particle* [5], decisión que se ha tomado al tener, el grupo de investigación, experiencia trabajando con plataformas de desarrollo de esta empresa, y haber llevado a cabo proyectos con un propósito similar con otros dispositivos de *Particle*.

1.2 Objetivos

El objetivo principal de este TFG consiste en la validación experimental del desarrollo de aplicaciones basadas en *Tensor Flow Lite* y procesamiento *"on edge"* sobre la plataforma IoT *Argon* de *Particle*. Con esta finalidad, se implementará una aplicación para la detección de un conjunto de gestos de la mano, definidos en el desarrollo del proyecto, a partir de una serie de elementos como, por ejemplo, un acelerómetro. Así, se han de seguir una serie de pasos que conforman, a grandes rasgos, diferentes objetivos parciales del presente Trabajo Fin de Grado.

De esta manera, los objetivos parciales propuestos para cumplir con el propósito general del Trabajo Fin de Grado son los expuestos en la Tabla 1.

Tabla 1. Objetivos del presente Trabajo Fin de Grado

Objetivos	Descripción
OBJ1	Comprensión del entorno de programación y funcionalidades de la plataforma *Argon* de *Particle*.
OBJ2	Comprensión del algoritmo de IA *Tensor Flow Lite* y sus funcionalidades, así como las aplicaciones de referencia.
OBJ3	Desarrollo de nuevos modelos para aplicaciones específicas.
OBJ4	Implementación y validación de los modelos desarrollados sobre el dispositivo IoT *Argon* de *Particle*.

1.3 Peticionario

Actúa como peticionario del presente Trabajo Fin de Grado (TFG) la Escuela de Ingeniería de Telecomunicación y Electrónica (EITE) de la Universidad de Las Palmas de Gran Canaria (ULPGC) como requisito indispensable para la obtención del título de Graduado en Ingeniería en Tecnologías de la Telecomunicación, tras haber superado con éxito las asignaturas especificadas en el Plan de Estudios.

1.4 Estructura del documento

El presente documento está dividido en tres partes diferenciadas: Memoria, Pliego de Condiciones y Presupuesto. A su vez, la Memoria se ha estructurado en 6 capítulos, además de las referencias bibliográficas empleadas, tal como se describe a continuación:

Capítulo 1. Introducción. Se presenta el planteamiento del problema que ha motivado la realización de este TFG, se plantean las soluciones que se van a desarrollar, y se detallan los objetivos propuestos para ello. A continuación, se expone el peticionario y la estructura general del documento.

Capítulo 2. Componentes HW/SW. Se presentan los recursos que se utilizarán en el desarrollo del presente TFG, proporcionando una descripción detallada de los dispositivos *hardware* y los recursos *software* utilizados.

Capítulo 3. Tensor Flow. En este capítulo se introduce brevemente el concepto de redes neuronales convolucionales (CNN). Además, se detalla el proceso de construcción y entrenamiento de un modelo basado en *Tensor Flow Lite*, así como el proceso de desarrollo de una aplicación para la implementación y ejecución de dicho modelo.

Capítulo 4. Desarrollo de un modelo inicial Tensor Flow. En este capítulo se presente el desarrollo completo de un modelo capaz de reconocer tres gestos predeterminados a partir de la información registrada a partir de un acelerómetro.

Capítulo 5. Desarrollo de un modelo Tensor Flow para el reconocimiento de letras del alfabeto dactilológico. En este capítulo se expone el proceso de desarrollo de un modelo capaz de reconocer tres letras del alfabeto dactilológico a partir de la información capturada por un acelerómetro, así como una serie de sensores flexibles integrados en un guante.

Capítulo 6. Conclusiones y líneas futuras. Se recogen las conclusiones obtenidas tras haber completado los objetivos propuestos para este TFG.

La segunda parte del documento consiste en el Pliego de Condiciones, mientras que la tercera parte se corresponde con el Presupuesto.

Pliego de condiciones. Se exponen las condiciones bajo las que se ha desarrollado el presente TFG.

Presupuesto. En este apartado se recogen los gastos generados en la realización del presente TFG.

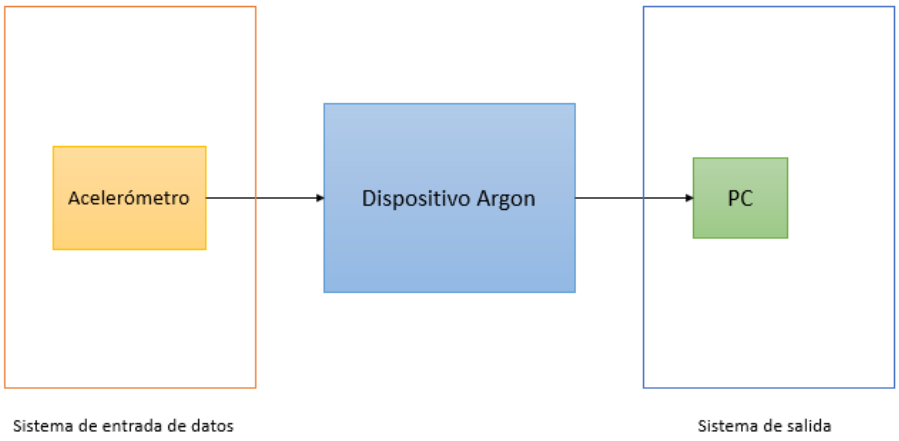
CAPÍTULO 2. Componentes HW/SW

2.1 Componentes hardware

A partir del planteamiento inicial del objetivo del presente TFG, se establecen los recursos HW necesarios para el diseño de la plataforma inicialmente planteada. De esta manera, se establece la necesidad de disponer de un acelerómetro que permita contribuir a la identificación del movimiento de la mano, y un dispositivo capaz de procesar localmente los datos recogidos a partir de la ejecución del algoritmo TF, que en el caso concreto de este TFG es el dispositivo Argon de la empresa *Particle*.

En la Figura 1 se muestra el diagrama de bloques que refleja la estructura del diseño inicial, así como los componentes necesarios.

Figura 1. Diagrama de bloques de la estructura inicial



2.1.1 Dispositivo Argon [6]

Se trata de un potente dispositivo de desarrollo que puede actuar como punto independiente de Wi-Fi, concebido para la creación de proyectos y productos basados en IoT.

Cuenta con una memoria *flash* de 1 MB de capacidad y 256 KB de RAM además de varios LED integrados. Asimismo, dispone de 20 entradas/salidas de propósito general (GPIO) y periféricos avanzados entre otras características.

Además, está equipado con los procesadores **Nordic nRF52840**, que cuenta con un ARM Cortex-M4F de 32 bits y 64Mhz, y **Espressif ESP32**, coprocesador para la red Wi-Fi. Cuenta también con circuitería que permite la carga de baterías, facilitando la conexión de una batería Li-Po y 20 señales GPIOs a una interfaz con sensores, actuadores y otros componentes electrónicos.

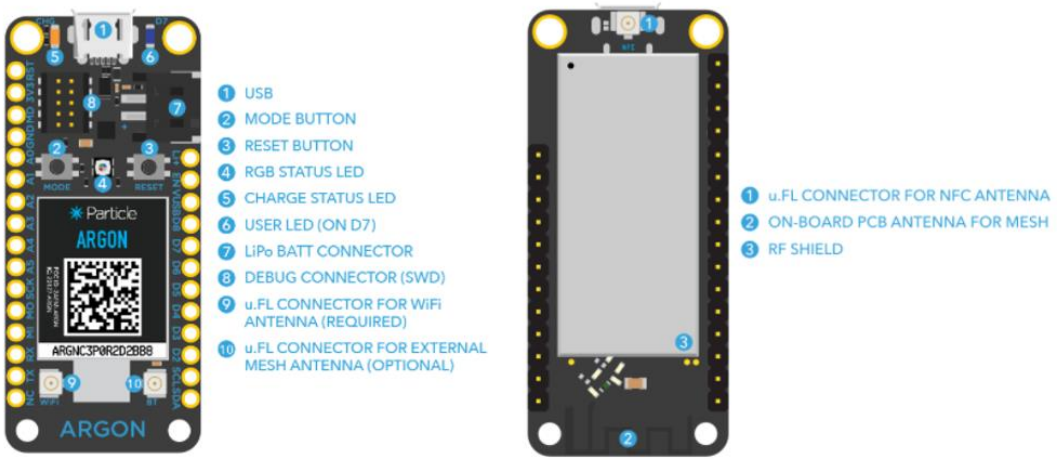
Además, cuenta con diversas interfaces analógicas, digitales y de comunicación, tal y como se puede ver en la Tabla 2.

Tabla 2. Periféricos de la placa Argon de Particle

Tipo de periférico	Cantidad	Entrada(I)/Salida(O)
Digital	20	I/O
Analógico (ADC)	6	I
UART	1	I/O
SPI	1	I/O
I2C	2	I/O
USB	1	I/O
PWM	8	O

En la Figura 2 se muestra la estructura del dispositivo Argon, así como los distintos botones y conectores que lo componen.

Figura 2. Estructura del dispositivo Argon de Particle [6]



Los pines son otro componente fundamental, permitiendo las interacciones con el microcontrolador. Los pines GPIO pueden conectarse a sensores o botones entre otros elementos, funcionando como entradas, o a LED funcionando como salidas. Además, cuenta con una serie de pines de *reset* y alimentación, o pines para la comunicación serie, tal y como se expone de forma detallada en la Tabla 3.

Tabla 3. Pines del dispositivo Argon de Particle [6]

Pin	Description
Li+	This pin is internally connected to the positive terminal of the LiPo battery connector.
VUSB	This pin is internally connected to the USB (+ve) supply.
3V3	This pin is the output of the on-board 3.3V regulator.
GND	System ground pin.
EN	Device enable pin is internally pulled-up. To disable the device, connect this pin to GND.
RST	Active-low system reset input. This pin is internally pulled-up.
MD	This pin is internally connected to the MODE button. The MODE function is active-low.
RX	Primarily used as UART RX, but can also be used as a digital GPIO.
TX	Primarily used as UART TX, but can also be used as a digital GPIO.
SDA	Primarily used as data pin for I2C, but can also be used as a digital GPIO.
SCL	Primarily used as clock pin for I2C, but can also be used as a digital GPIO.
MO,MI,SCK	These are the SPI interface pins, but can also be used as a digital GPIO.
D2-D8	These are generic GPIO pins. D2-D8 are PWM-able.
A0-A5	These are analog input pins that can also act as standard digital GPIO. A0-A5 are PWM-able.

2.1.2 Dispositivo LSM9DS1 [7]

Se trata de un circuito integrado, desarrollado por la empresa *Sparkfun*, capaz de medir tres propiedades clave del movimiento, como son la velocidad angular, la aceleración y la dirección. Para ello, cuenta con un acelerómetro, un giroscopio y un magnetómetro de tres ejes cada uno. De esta manera se consigue proporcionar una gran cantidad de información sobre el movimiento y la orientación de un objeto. Dicho dispositivo se puede observar en la Figura 3.

Figura 3. Acelerómetro LSM9DS1 [7]

El giroscopio se encarga de medir la velocidad angular, o lo que es lo mismo, la velocidad de rotación alrededor de un eje. Este dispositivo es capaz de medir hasta ± 2000 DPS (grados por segundo).

Por otro lado, el acelerómetro mide, como su propio nombre indica, la aceleración asociada al cambio de velocidad. El dispositivo mide su aceleración en g (gravedades – 9,8 m/s^2), y su escala se puede establecer en ± 2,4,8 o 16 g.

Finalmente se encuentra el magnetómetro, que mide la potencia y la dirección de los campos magnéticos. La unidad de medida del dispositivo es unidades de gauss (Gs), estableciendo su escala en ± 4,8,12 o 16 Gs.

El dispositivo LSM9DS1 cuenta con 3 ejes para cada propiedad, lo cual significa que mide cada una de estas propiedades en tres dimensiones, produciendo nueve datos: aceleración en x,y,z, rotación angular en x,y,z, y fuerza del campo magnético en x,y,z.

Cabe destacar que es compatible con interfaces SPI y I2C, siendo esta última el método de comunicación elegido para el presente TFG. Es un protocolo diseñado para permitir múltiples circuitos integrados digitales esclavos que comuniquen con uno o más dispositivos maestros [8].

De esta manera, en la Tabla 4 se exponen los pines del dispositivo necesarios en el diseño de la plataforma HW/SW.

Tabla 4. Pines necesarios del Acelerómetro LSM9DS1

Pin	Función	Descripción
GND	Conexión a tierra	0V de tensión
VDD	Alimentación	Suministro de tensión al chip
SDA	I2C: *Serial Data*	Datos en serie I2C
SCL	*Serial Clock*	Reloj en serie I2C

Por último, se detalla a continuación el *pin-out* propuesto inicialmente para la interconexión del acelerómetro LSM9DS1 con el dispositivo Argon de Particle mediante la interfaz I2C.

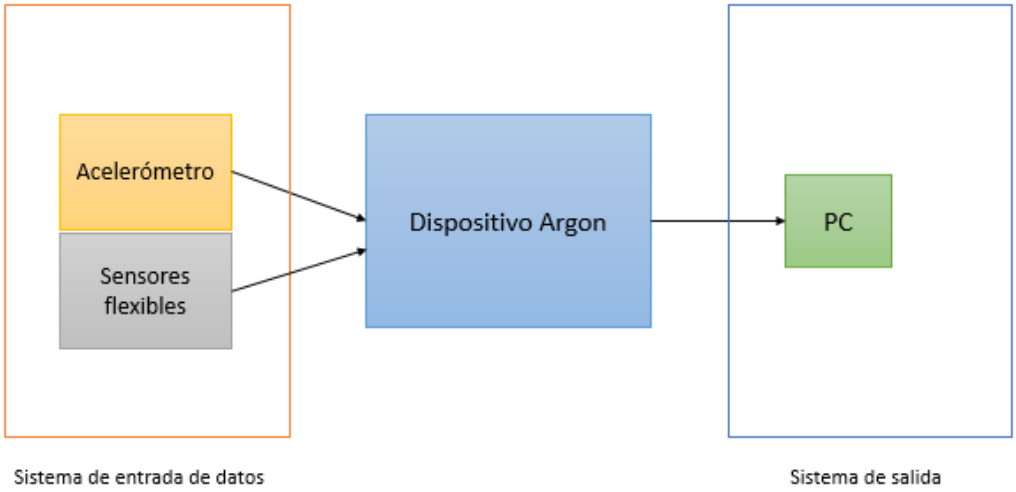
LSM9DS1 Argon

SCL----------SCL
SDA----------SDA
VDD----------3V3
GND----------GND

2.1.3 Sensores flexibles [9]

El planteamiento inicial de la plataforma se verá modificado debido a la necesidad de incluir sensores flexibles que ayuden en el objetivo final de detectar una serie de letras del alfabético dactilológico de la lengua de signos española, con lo que, para este caso, el diagrama de bloques varía y se muestra en la Figura 4.

Figura 4. Diagrama de bloques de la estructura con sensores flexibles



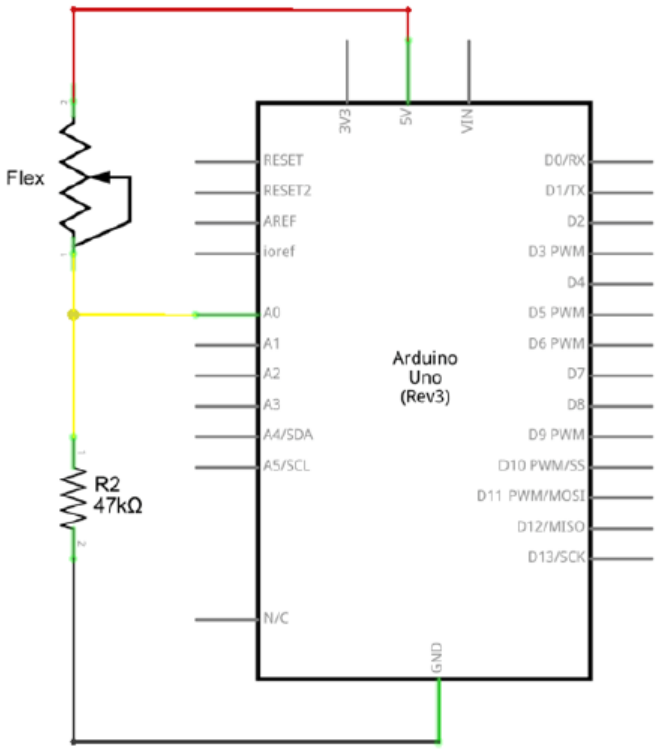
Los sensores de 2,2" utilizados en el presente TFG, fabricados por la empresa *Spectra Symbol*, son resistencias variables que aumentan su valor a medida que se doblan. Esto es posible porque uno de los lados del sensor está impreso con una tinta de polímero con partículas conductoras incrustadas en él, de tal manera que cuando dicho sensor está recto, las partículas le dan a la tinta una resistencia de, aproximadamente, 30kΩ, mientras que cuando está doblado, las partículas conductoras se separan y la resistencia aumenta. Es por ello que estos sensores resultan elementos útiles para el desarrollo de la plataforma HW/SW del presente TFG, permitiendo conocer la posición de cada dedo de la mano a partir del grado de doblez de las resistencias.

Cabe destacar que dichos sensores se flexionan en una sola dirección, tal y como se muestra en la Figura 5, de la otra manera, se obtendrían datos muy poco fiables, e incluso se podría llegar a dañar el sensor.

Figura 5. Sensor flexible de 2.2" de Spectra Symbol [9]

En la Figura 6 se representa el circuito propuesto para la conexión de los sensores utilizando una plataforma basada en MCU [10], y el cual se ha tomado como referencia para el montaje del circuito basado en el dispositivo Argon utilizado en el presente TFG. Al combinar el sensor de flexión con una resistencia estática para crear un divisor de tensión, es posible generar una tensión variable que pueda leerse mediante un convertidor analógico-digital integrado en el microcontrolador.

Figura 6. Circuito de conexión de los sensores flexibles [10]



Además, se recomienda que el valor de la resistencia usada para formar el divisor de tensión se encuentre entre 10k y 100k. Para calcular dicho valor, es decir, aquel que cubra el mayor rango de tensiones posible, se han calculado los valores de tensión mínimos y máximos teniendo en cuenta los valores mínimos y máximos de cada sensor flexible, obteniéndose los resultados mostrados en la Tabla 5.

Tabla 5. Valores del divisor de tensión

R	Vmin	Vmax	Vmax-Vmin
48000	1,23211488	2,20758621	0,97547132
49000	1,21623711	2,19246575	0,97622864
50000	1,20076336	2,17755102	0,97678766
51000	1,18567839	2,16283784	0,97715945
52000	1,17096774	2,14832215	0,97735441
53000	1,15661765	2,134	0,97738235
54000	1,14261501	2,11986755	0,97725254
55000	1,12894737	2,10592105	0,97697368
56000	1,11560284	2,09215686	0,97655403
57000	1,10257009	2,07857143	0,97600134
58000	1,08983834	2,06516129	0,97532295
59000	1,07739726	2,05192308	0,97452582
60000	1,06523702	2,0388535	0,97361648

Así, con una $V_{cc} = 3.3\ V$, siendo R la resistencia del divisor de tensión y $R_{flexmin}$ y $R_{flexmax}$, valores tomados experimentalmente mediante el uso de un multímetro, se obtienen los resultados expuestos en la Tabla 6, aplicando las fórmulas de un divisor de tensión que se exponen en la Fórmula 1, observándose que la resistencia óptima para el divisor de tensión es de 50kΩ.

Fórmula 1. Divisor de tensión

$$V_{min} = V_{cc} \cdot \frac{R_{flexmin}}{R + R_{flexmin}}$$

$$V_{max} = V_{cc} \cdot \frac{R_{flexmax}}{R + R_{flexmax}}$$

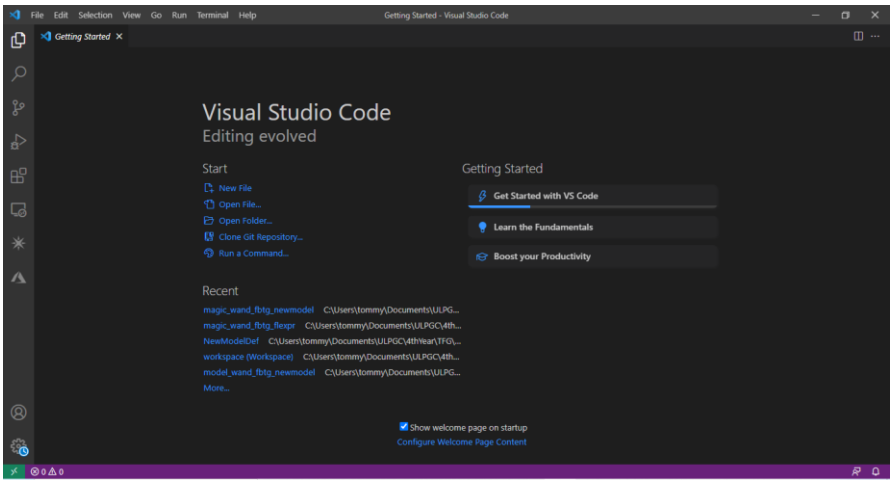
Tabla 6. Resultados de las resistencias flexibles

	Rflexmin	Rflexmax	Rdiv
Flex 18 (dedo pulgar)	32000	75000	50000
Flex 40 (dedo índice)	83000	130000	50000
Flex 26 (dedo corazón)	75000	145000	50000
Flex 22 (dedo anular)	28000	80000	50000
Flex 25 (dedo meñique)	83000	140000	50000

2.2 Componentes software

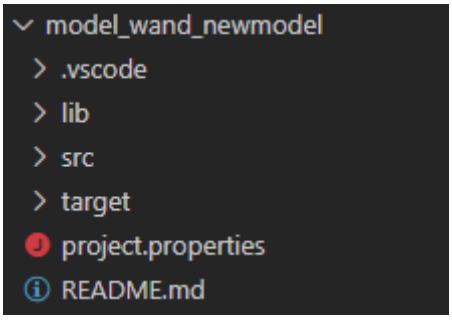
El entorno de desarrollo que se ha utilizado en el presente Trabajo Fin de Grado ha sido *Particle Workbench*, cuya pantalla inicial se expone en la Figura 7, concretamente mediante la plataforma *Visual Studio Code* [11].

Figura 7. Particle Workbench



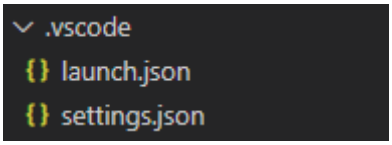
La estructura básica de un proyecto se puede ver en la Figura 8.

Figura 8. Estructura básica en Visual Studio Code



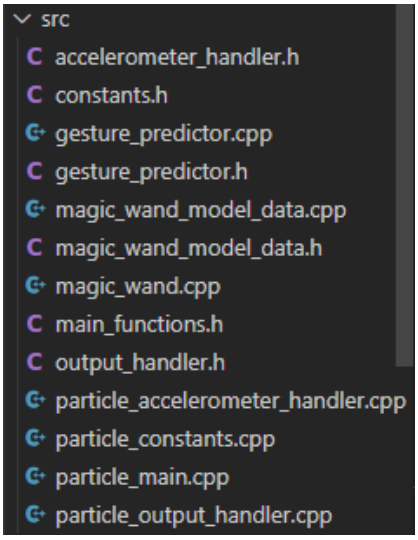
- El directorio `.vscode` contiene los ajustes específicos del proyecto, tal y como se muestra en la Figura 9.

Figura 9. Directorio .vscode



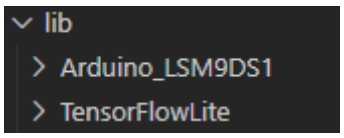
- El directorio `src` contiene los archivos fuente. Se pueden crear múltiples archivos fuente y cabeceras en este directorio, como se muestra en la Figura 10.

Figura 10. Directorio src



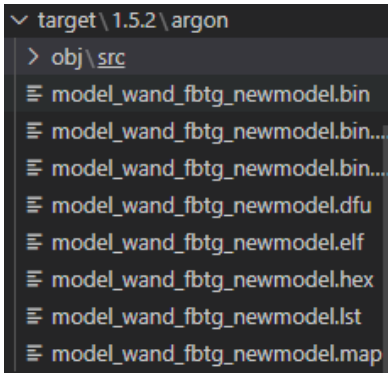
- El directorio `lib` contiene las librerías que se han incluido en el proyecto, como en el extracto que se expone en la Figura 11.

Figura 11. Directorio lib



- El directorio `target` contiene los ficheros generados a partir de la compilación del proyecto, organizados en subcarpetas basadas en la versión del dispositivo o plataforma de Particle que se utilice, como se observa en la Figura 12.

Figura 12. Directorio target



- El archivo `*.bin` es el resultado de la compilación en la nube para este proyecto.
- El archivo `project.properties` especifica todas las librerías utilizadas en el proyecto.
- `README.md` es donde se puede encontrar la documentación necesaria para el proyecto.

2.2.1 Plataforma TensorFlow 2.2.0

TensorFlow (TF) [12] es una plataforma de código abierto de extremo a extremo para el aprendizaje automático. Cuenta con un ecosistema integral y flexible de herramientas, bibliotecas y recursos de la comunidad que permite a los investigadores innovar con el aprendizaje automático y, a los desarrolladores, compilar e implementar con facilidad aplicaciones con dicha tecnología. Asimismo, ofrece varios niveles de abstracción para poder elegir el que se adecue a cada una de las necesidades. Compila y entrena modelos mediante la API de alto nivel de Keras [13].

Por su parte, **TensorFlow Lite** [4] es un conjunto de herramientas para ayudar a los desarrolladores a ejecutar modelos de TF en dispositivos empotrados, móviles o de IoT. Además, permite la inferencia de aprendizaje automático en dispositivos con una latencia baja y un tamaño de objeto binario pequeño. El término **inferencia** se refiere al proceso de ejecutar un modelo de TF Lite en el dispositivo para hacer predicciones basadas en datos de entrada.

Consta de dos componentes principales, por un lado, el **intérprete de TF Lite**, que ejecuta modelos especialmente optimizados en varios HW diferentes, como puede ser teléfonos móviles, sistemas empotrados Linux, microcontroladores… El otro componente principal es el **conversor de TF Lite**, que convierte los modelos TF en un formato eficiente para que el intérprete pueda utilizarlo, y, además, pueda implementar optimizaciones para mejorar el tamaño y el rendimiento de los objetos binarios.

TensorFlow Lite para microcontroladores [14] ha sido diseñado para ejecutar modelos de aprendizaje automático en microcontroladores y otros dispositivos usando poca memoria. El entorno de ejecución principal puede almacenarse en 16 KB en un procesador ARM Cortex M3, y puede ejecutar varios modelos básicos. Además, no requiere compatibilidad con el sistema operativo, ninguna biblioteca C o C++ estándar ni asignación de memoria dinámica. Con la incorporación del aprendizaje automático en pequeños microcontroladores se puede potenciar la inteligencia de millones de dispositivos que usan diariamente sin depender de HW costoso o conexiones a Internet estables, condicionadas por el ancho de banda y el consumo de energía.

Actualmente, el intérprete de TF Lite es compatible con un subconjunto limitado de operadores de TF, optimizados para ser usados en dispositivos, lo cual significa que algunos modelos requieren pasos adicionales para trabajar con TF Lite. De esta manera, se deben tener en cuenta las siguientes limitaciones:

- Compatibilidad con un subconjunto limitado de operaciones de TensorFlow.
- Compatibilidad con un conjunto limitado de dispositivos.
- API de C++ de bajo nivel que requiere administración manual de memoria.
- No compatible con el entrenamiento en dispositivos.

CAPÍTULO 3: Tensor Flow

3.1 Introducción

3.1.1 Redes neuronales convolucionales

Las redes neuronales convolucionales o *Convolutional Neural Networks* (CNN) se usan, principalmente para clasificar o agrupar imágenes o reconocimiento de objetos o gestos, por ejemplo, identificación de caras, individuos, signos, tumores, y muchos otros aspectos sobre datos visuales [15].

Las CNN están compuestas por múltiples capas de neuronas artificiales, las cuales son funciones matemáticas que calculan la suma del peso de diferentes entradas, representadas como *arrays* multidimensionales, proporcionando como salida un valor de activación [16].

3.1.2 Tensores

Keras utiliza un *array* multidimensional de *NumPy* [17] como estructura básica de datos y la denomina **tensor** [18]. Esta estructura tiene tres atributos principales: el número de ejes, la forma y el tipo de datos.

- Número de ejes

A un tensor que contiene un solo número se le denomina *scalar* o tensor 0D, mientras que un *array* de números es un vector o tensor 1D, y un *array* de vectores será una matriz o tensor 2D. Si esta matriz se empaqueta en un nuevo *array* se obtiene un tensor 3D, y así sucesivamente. En la librería *NumPy*, de Python, explicada más adelante, esto se llama `ndim` del tensor.

- Forma

Hace referencia a una tupla o lista secuenciada de enteros que describen cuántas dimensiones tiene el tensor en cada eje. Este atributo se denomina `shape` en la librería *NumPy*.

- Tipo de datos

Indica el tipo de datos que contiene el tensor, que pueden ser *uint8, float32, float64…* En la librería *NumPy* este atributo se denomina `dtype`.

3.2 Construcción y entrenamiento de un modelo con TensorFlow Lite

En este apartado se describe como referencia el proceso que se lleva a cabo para construir y entrenar un modelo simple desde cero. En un entorno basado en sistema operativo Windows, el hardware a utilizar puede ser cualquier dispositivo basado en MCU (*Microcontroller Unit)* para programación en lenguaje *wiring*. Además, para crear el modelo de aprendizaje automático (*Machine Learning*) es necesario hacer uso de lenguajes de programación y herramientas como Python y TensorFlow [19].

Los pasos a seguir para construir, entrenar e implementar, en términos generales, un modelo TensorFlow son los siguientes.

- Obtener un conjunto de datos.
- Entrenar el modelo de aprendizaje profundo.
- Evaluar las prestaciones del modelo.
- Convertir el modelo para ser implementado en el dispositivo.
- Implementar la aplicación en el microcontrolador.

A lo largo del presente capítulo se irán desarrollando todos y cada uno de estos pasos de manera ordenada.

El objetivo principal que se plantea como referencia de este proceso se basa en entrenar un modelo que, a partir de un valor “x”, prediga su valor en “y”. En este caso, se utiliza una sinusoide para obtener los datos de entrenamiento, siendo los valores en “x” la entrada del modelo, y la salida, sus respectivas funciones seno en “y”; es obvio que dicho valor se puede calcular directamente, pero de esta manera se demuestran los fundamentos del aprendizaje automático. Adicionalmente, el modelo se ejecuta en un dispositivo hardware teniendo en cuenta el ejemplo utilizado y su aplicación. En este caso se sabe que la curva del seno varía de -1 a 1 y viceversa, por lo que resulta adecuado para, por ejemplo, controlar el parpadeo de un LED o una animación gráfica.

3.2.1 Importación de dependencias

La primera tarea consiste en importar las librerías necesarias para entrenar y convertir el modelo, como se muestra en el código de la Figura 13.

Figura 13. Extracto de código - Importar dependencias

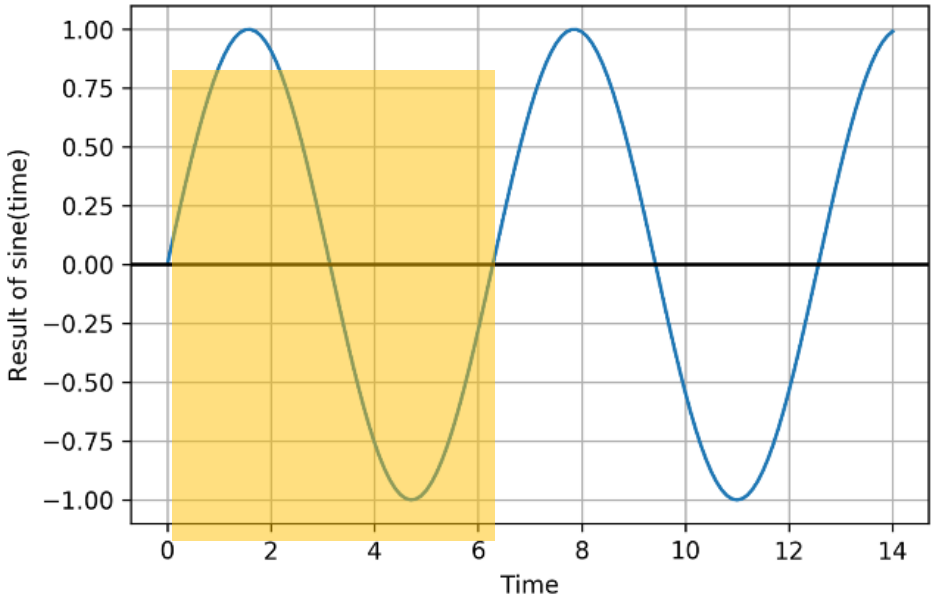
```
# TensorFlow is an open source machine learning library
!pip install tensorflow==2.0
import tensorflow as tf
# NumPy is a math library
import numpy as np
# math is Python's math library
import math
```

En el código expuesto se instala, en este caso, la librería de *TensorFlow* 2.0 usando *pip*, un gestor de paquetes para Python, además de importar *TensorFlow*, *NumPy*, que permite cálculos numéricos sencillos y eficientes, *Matplotlib*, que es una librería de visualización que se usa para representar diferentes tipos de gráficas, y la librería de operaciones matemáticas propia de Python.

3.2.2 Generación de datos

Para la obtención de datos, se generan, en este caso, 1000 valores (conjunto de datos) que representan puntos aleatorios de una forma de onda sinusoidal. Como se observa en la Figura 14, un ciclo se completa cada 6 unidades en el eje x aproximadamente, siendo este el periodo (2π); por ello, el código genera valores aleatorios de "x" desde 0 hasta 2π, calculando posteriormente el valor de la función seno para cada uno de ellos.

Figura 14. Sinusoide [19]



En la Figura 15 se expone el código a partir del cual se generan números aleatorios y se calcula su función seno usando *NumPy*.

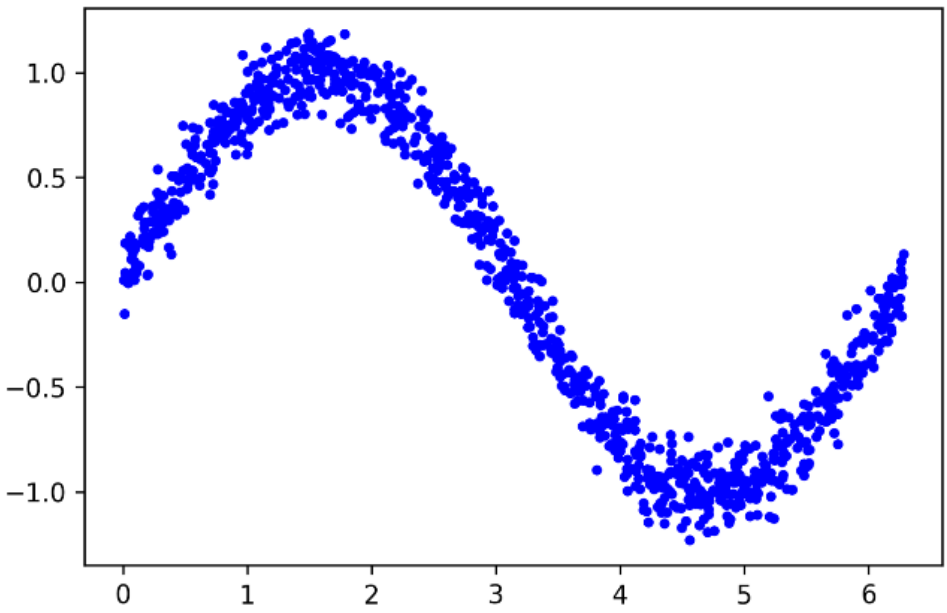
Figura 15. Extracto de código - Generación de números aleatorios y cálculo del seno

```
# We'll generate this many sample datapoints
SAMPLES = 1000

# Set a "seed" value, so we get the same random numbers each time we run this
# notebook. Any number can be used here.
SEED = 1337
np.random.seed(SEED)
tf.random.set_seed(SEED)

# Generate a uniformly distributed set of random numbers in the range from
# 0 to 2π, which covers a complete sine wave oscillation
x_values = np.random.uniform(low=0, high=2*math.pi, size=SAMPLES)

# Shuffle the values to guarantee they're not in order
np.random.shuffle(x_values)

# Calculate the corresponding sine values
y_values = np.sin(x_values)
```

Es importante destacar algunos aspectos del código presentado. En primer lugar, se utiliza el método `np.random.uniform()` para generar los valores de "x", devolviendo un *array* de números aleatorios en el rango especificado. Una vez se obtienen los datos, estos se mezclan, de esta manera se consigue que la aleatoriedad aumente aún mas. En segundo lugar, el método `np.sin()` es capaz de calcular el seno de todos los valores de "x" de una sola vez, devolviendo los resultados en un *array*.

Una vez obtenido el conjunto de datos, es importante añadirle algo de ruido, asemejándolo así a datos obtenidos en una situación real, como se representa gráficamente en la Figura 16.

Figura 16. Conjunto de datos con ruido [19]



3.2.3 Fragmentación de los datos

El siguiente paso consiste en realizar la fragmentación del conjunto de datos obtenido, de manera que en este caso se establece que un 60% irá destinado al entrenamiento, un 20% a la validación, donde se evalúa la precisión del modelo, y un 20% a los test finales, lo que suele representar una fragmentación típica en la mayor parte de las aplicaciones, tal y como se expone en el extracto de código representado en la Figura 17.

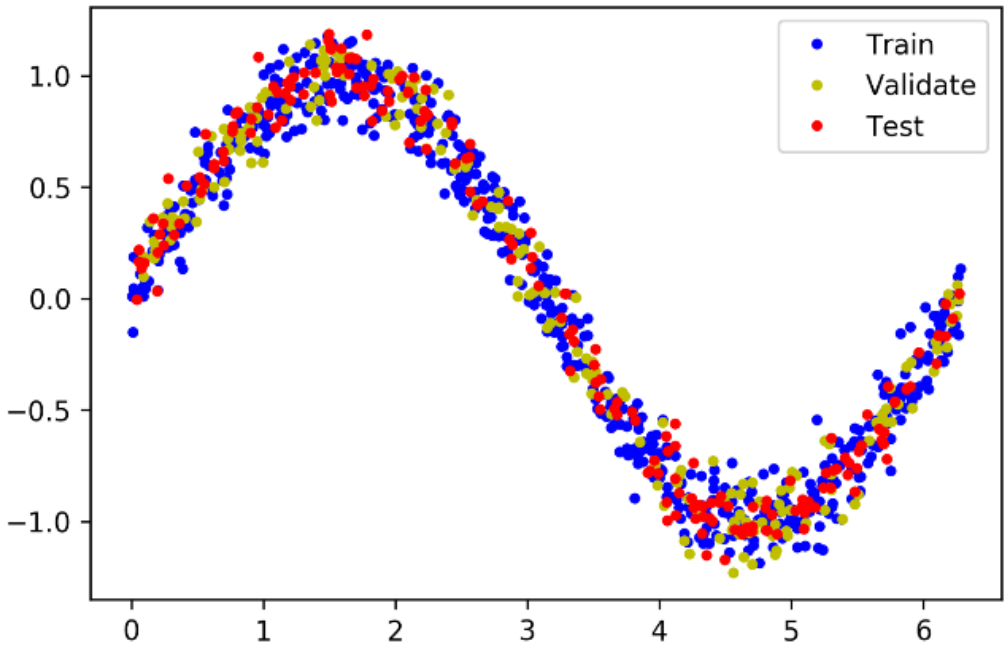
Figura 17. Extracto de código - Fragmentación

```
# We'll use 60% of our data for training and 20% for testing. The remaining 20%
# will be used for validation. Calculate the indices of each section.
TRAIN_SPLIT =  int(0.6 * SAMPLES)
TEST_SPLIT = int(0.2 * SAMPLES + TRAIN_SPLIT)

# Use np.split to chop our data into three parts.
# The second argument to np.split is an array of indices where the data will be
# split. We provide two indices, so the data will be divided into three chunks.
x_train, x_validate, x_test = np.split(x_values, [TRAIN_SPLIT, TEST_SPLIT])
y_train, y_validate, y_test = np.split(y_values, [TRAIN_SPLIT, TEST_SPLIT])

# Double check that our splits add up correctly
assert (x_train.size + x_validate.size + x_test.size) ==  SAMPLES
```

El método `split()` recibe por parámetros un *array* de datos y otro de índices, separando dichos datos en diferentes partes en función de los índices establecidos.

En la Figura 18 se pueden ver los resultados de la fragmentación de los datos.

Figura 18. Resultados de la fragmentación de los datos [19]



3.2.4 Creación de un modelo básico

Como se ha explicado previamente, se construye un modelo capaz de predecir un valor a partir de otro, lo que se conoce como regresión. Es un tipo de modelo muy utilizado para tareas que requieren salidas numéricas.

Para crear dicho modelo se diseña una red neuronal muy simple compuesta por distintas capas de neuronas, con el objetivo de aprender cualquier patrón en función de los datos entrenados y, por último, realizar las predicciones requeridas, para lo cual se usa la API de alto nivel de *TensorFlow*, *Keras,* como se observa en el código expuesto en la Figura 19.

Figura 19. Extracto de código - CNN

```
# We'll use Keras to create a simple model architecture
from tf.keras import layers
model_1 = tf.keras.Sequential()

# First layer takes a scalar input and feeds it through 16 "neurons." The
# neurons decide whether to activate based on the 'relu' activation function.
model_1.add(layers.Dense(16, activation='relu', input_shape=(1,)))

# Final layer is a single neuron, since we want to output a single value
model_1.add(layers.Dense(1))

# Compile the model using a standard optimizer and loss function for regression
model_1.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])

# Print a summary of the model's architecture
model_1.summary()
```

.

En primer lugar, se crea un modelo secuencial compuesto por dos capas. La primera capa consta de una sola entrada (el valor de “x”) y 16 neuronas. Se calcula la activación de éstas mediante *ReLU*, una función matemática utilizada para darle forma a la salida de las neuronas. Ésta activa un nodo solo si la entrada está por encima de cierto umbral. El comportamiento más habitual es que mientras la entrada tenga un valor por debajo de cero, la salida será nula, pero cuando la entrada se eleve por encima de cero, la salida se corresponde con una relación lineal con la variable de entrada de la forma $f(x) = x$. Esto implica que las neuronas pueden modelar relaciones no lineales, donde la estimación de “y” no crece de la misma manera que la de “x”, ya que los valores de activación de la primera capa servirán de entrada para la segunda.

La segunda capa consta de una sola neurona, por ello recibe 16 entradas, una por cada neurona de la capa anterior. El propósito de esta capa consiste en combinar todas las activaciones de la capa previa en un solo valor de salida que se calcula de la manera mostrada en el extracto de código de la Figura 20.

Figura 20. Extracto de código - Combinación de las activaciones

```
# Here, `inputs` and `weights` are both NumPy arrays with 16 elements each
output = sum((inputs * weights)) + bias
```

Para finalizar, se compila el modelo y se obtiene un resumen de la información sobre su arquitectura, mostrando el número de capas, las formas de la salida y el número de parámetros, tal y como se expone en la Figura 21.

Figura 21. Resumen de la información de la CNN

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 16)                32
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 17
=================================================================
Total params: 49
Trainable params: 49
Non-trainable params: 0
_________________________________________________________________
```

El tamaño del modelo depende, por lo general, del número de parámetros, siendo estos el número total de pesos y sus sesgos. Así, en este modelo de dos capas hay 16 conexiones en la primera, que conectan la entrada con cada una de las neuronas, y la segunda capa tiene otras 16 conexiones, que van desde las neuronas hasta la salida, obteniendo un total de 32. Además, cada neurona tiene un sesgo, sumando un total de 17 que añadido a las conexiones asciende a un total de 49 parámetros.

3.2.5 Entrenamiento del modelo

Para entrenar un modelo en *Keras*, simplemente se llama al método `fit()`, pasando como parámetros los datos y otros argumentos necesarios, como se expone en el código de la Figura 22.

Figura 22. Extracto de código - Método fit()

```
history_1 = model_1.fit(x_train, y_train, epochs=1000, batch_size=16,
                        validation_data=(x_validate, y_validate))
```

Se ha de tener en cuenta que solo se utilizan los datos designados a la fase de entrenamiento. Los dos primeros argumentos, `x_train`, `y_train`, se corresponden con los valores de “x” e “y” de dichos datos, respectivamente.

El argumento `epochs` especifica el número de veces que se entrena el conjunto de datos, es decir, el número de veces que los datos de entrenamiento han pasado por la red neuronal en el proceso de entrenamiento. Se debe tener especial cuidado con este parámetro ya que un número elevado de *epochs* provoca que el modelo se ajuste en exceso a los datos y puede tener problemas de generalización en el conjunto de datos de prueba y validación, lo que se conoce como *overfitting* o sobreajuste [19]. Además, aunque esto no suceda, la red dejará de mejorar tras cierto número de entrenamientos, por lo que se hace fundamental no entrenar en exceso, ya que conllevaría un gasto innecesario de tiempo y recursos.

Los datos de entrenamiento pueden particionarse en lotes (*batches*) para pasarlos por la red, siendo el argumento `batch_size` el que especifica el tamaño de estos lotes de datos de entrenamiento con los que se alimenta la red. En consecuencia, se puede deducir que el tamaño óptimo dependerá de muchos factores, entre ellos, la capacidad de memoria del computador que se usa para hacer los cálculos.

El argumento `validation_data` especifica el conjunto de datos destinado a la validación, que será ejecutado en la red durante el proceso de entrenamiento, comparándose, posteriormente, las predicciones con los valores esperados.

Así, para empezar el entrenamiento se ejecuta el código anterior y aparecen ciertos registros, que se muestran en la Figura 23.

Figura 23. Registros al inicio del entrenamiento

```
Train on 600 samples, validate on 200 samples
Epoch 1/1000
600/600 [==============================] - 1s 1ms/sample - loss: 0.7887 - mae:
0.7848 - val_loss: 0.5824 - val_mae: 0.6867
Epoch 2/1000
600/600 [==============================] - 0s 155us/sample - loss: 0.4883 -
mae: 0.6194 - val_loss: 0.4742 - val_mae: 0.6056
```

Una vez finaliza el entrenamiento, se han de revisar las métricas para verificar que la red ha aprendido correctamente, para lo cual se puede comparar el primer y último *epoch*, como se muestra en la Figura 24.

Figura 24. Registros al finalizar el entrenamiento

```
Epoch 1/1000
600/600 [==============================] - 1s 1ms/sample - loss: 0.7887 - mae:
0.7848 - val_loss: 0.5824 - val_mae: 0.6867
```

```
Epoch 1000/1000
600/600 [==============================] - 0s 124us/sample - loss: 0.1524 -
mae: 0.3039 - val_loss: 0.1737 - val_mae: 0.3249
```

Para expresar el error o pérdida (`loss`) se utiliza el error cuadrático medio representado como número positivo. Como se puede observar, la red ha mejorado el error durante el entrenamiento, pasando de un valor de 0.7887 a otro de 0.1524.

Por una parte, el parámetro `mae` es el error absoluto medio de los datos de entrenamiento, representando la diferencia, en promedio, entre la predicción de la red y los valores de "y" esperados. Es común que el error inicial sea considerablemente alto, ya que no se ha entrenado, y en este caso es de 0.7848, un valor extremadamente elevado teniendo en cuenta que el rango de valores aceptables va de -1 a 1. Sin embargo, el error `mae` después del entrenamiento es de 0.3039, que sigue siendo un error significativo.

El valor `val_loss` representa las pérdidas de los datos de validación. En el último *epoch* se observa que las pérdidas del entrenamiento (0.1524) son ligeramente más bajas que las de validación (0.1737), lo cual puede significar que la red está sufriendo problemas de *overfitting*.

Finalmente, el valor `val_mae` es el error absoluto medio de los datos de validación, que con un valor de 0.3249 es aún peor que el `mae` para los datos de entrenamiento, otro signo de que se está produciendo *overfitting* en la red generada.

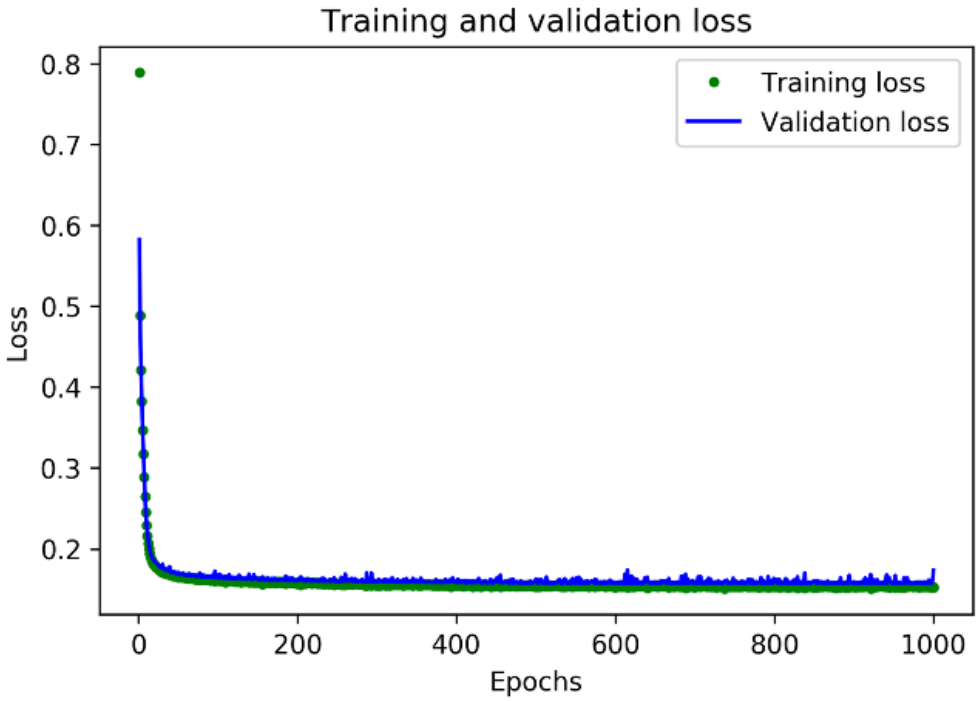
En primera instancia se puede concluir que el modelo no está realizando su trabajo correctamente, ya que la precisión de las predicciones es considerablemente baja. Para determinar el motivo, puede resultar conveniente representar gráficamente los datos obtenidos, para lo cual puede implementarse el código Python que se muestra en la Figura 25, utilizando la librería *Matplotlib.*

Figura 25. Extracto de código - Representación gráfica de los datos

```
loss = history_1.history['loss']
val_loss = history_1.history['val_loss']

epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'g.', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

En el eje "x" se representa el número de *epochs*, mientras que en el eje "y" se encuentran las pérdidas, como se muestra en la Figura 26.

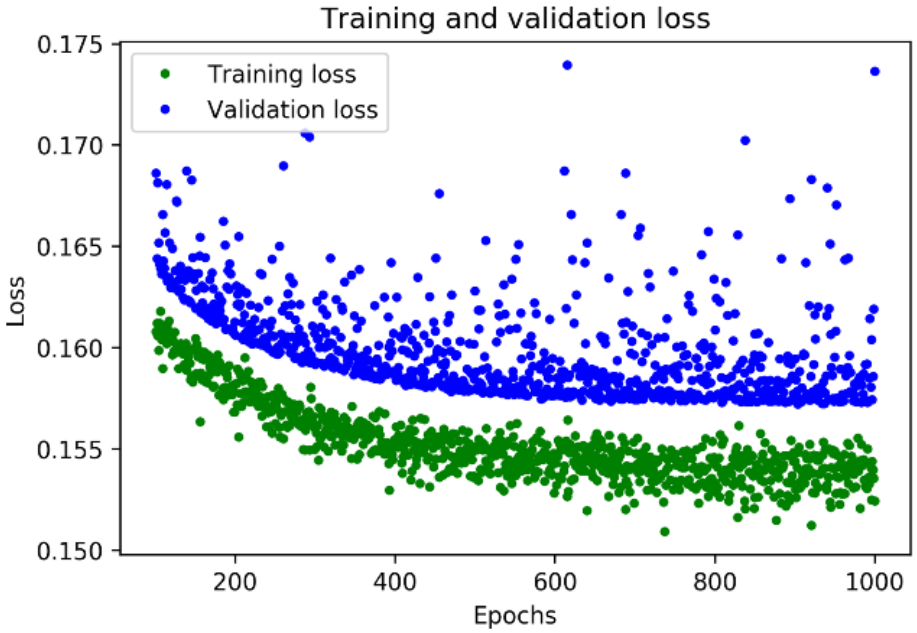
Figura 26. Número de epochs – Pérdidas [19]



Como se puede observar, las pérdidas decrecen rápidamente durante los primeros 50 *epochs* para, posteriormente, estabilizarse. Esto significa que el modelo ha mejorado considerablemente y ha generado predicciones precisas. Sin embargo, como se comentó previamente, se debe tener en cuenta que el objetivo es detener el entrenamiento cuando el modelo deje de mejorar, o cuando las pérdidas de entrenamiento sean menores que las de validación, denotando que el modelo ha aprendido a predecir los datos de entrenamiento, de tal manera que ya no se puede generalizar a nuevos datos.

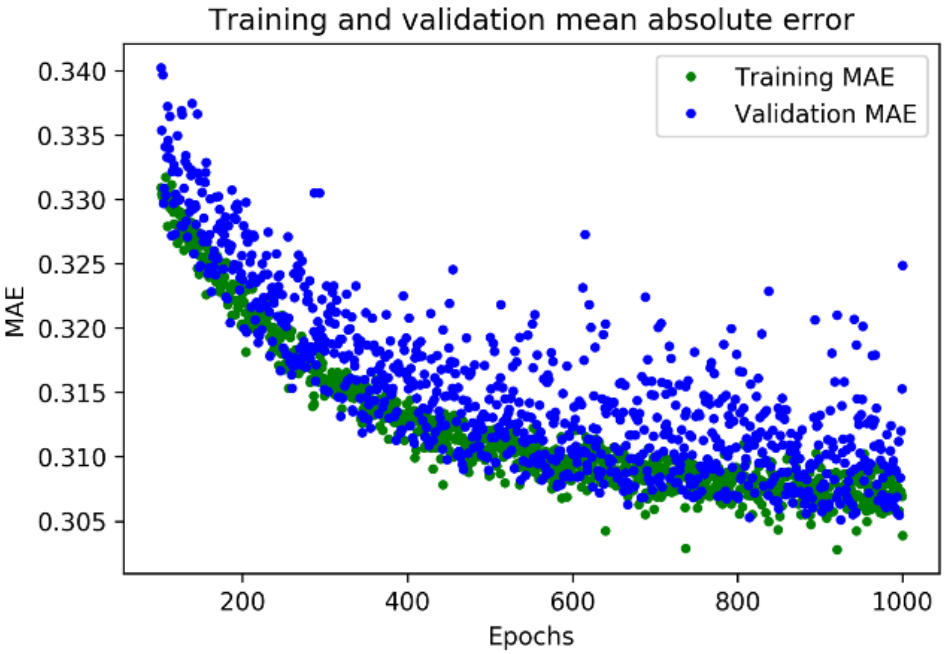
Si se amplía el gráfico, eliminando los primeros 100 *epochs*, como se muestra en la Figura 27, se observa que las pérdidas se reducen hasta estabilizarse, aproximadamente, en los 600 *epochs*. Además, el valor más bajo de las pérdidas de entrenamiento sigue siendo considerablemente alto, en torno a 0.15, y más aún lo es el valor de las pérdidas de validación.

Figura 27. Número de epochs - Pérdidas (Aumentado) [19]



Para obtener más conclusiones acerca de lo que está pasando se han de representar otros datos, como por ejemplo, el error absoluto medio mostrado en la Figura 28.

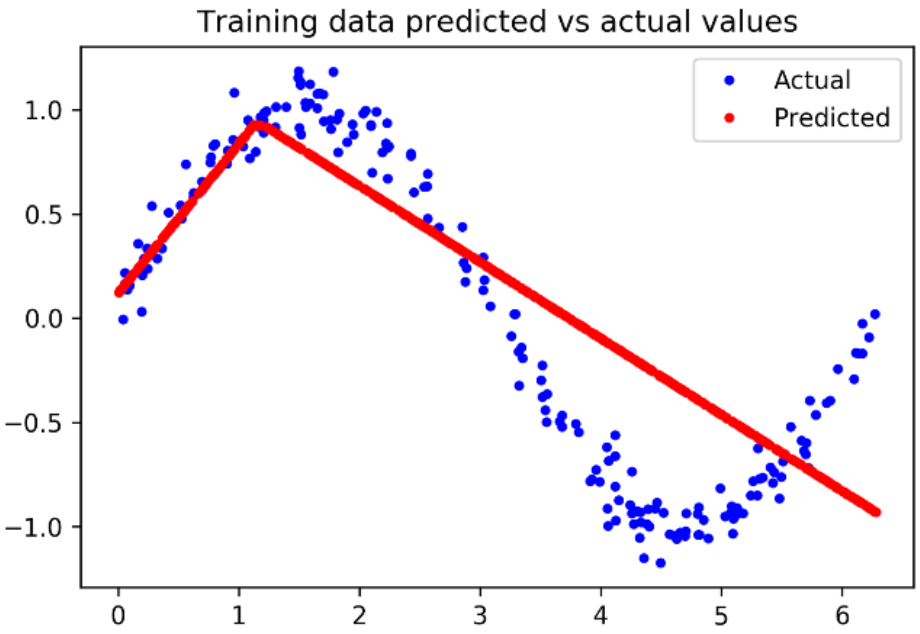
Figura 28. Error absoluto medio [19]



Así, se puede ver que, en promedio, las pérdidas de entrenamiento son más bajas que las de validación, representando esto que la red puede haber sufrido *overfitting,* o que ha aprendido los datos de entrenamiento de forma tan rigurosa que no es capaz de realizar predicciones precisas para nuevos datos. Además, el parámetro `mae` es muy elevado, alrededor de 0.31, lo que implica que las predicciones son erróneas al menos por un 0.31 (de -1 a +1), muy lejos de un modelado preciso.

Por último, se pueden comparar las predicciones de la red con los datos esperados, representados en la Figura 29.

Figura 29. Predicciones - Datos esperados [19]



Se observa claramente que la red ha aprendido de forma limitada, de manera que las predicciones son lineales y se ajustan de forma muy aproximada. Se puede concluir que el modelo no tiene suficiente capacidad para aprender la complejidad de una función seno, así que es probable que el resultado final sea solo una simple aproximación al modelo ideal.

La mejor opción para mejorar el modelo pasaría por aumentar su tamaño, incluyendo una nueva capa de neuronas. Cada capa representa una transformación de la entrada que se acercará, cada vez más, a la salida esperada; cuantas más capas se implementen, más complejas serán dichas transformaciones.

El nuevo modelo pasa a tener un total de 321 parámetros, lo que supone un aumento del tamaño del modelo de un 555%. Además, se reduce el valor de *epochs* a 600, teniendo en cuenta que el modelo anterior dejó de entrenar con relativa rapidez.

Una vez que finaliza el entrenamiento se pueden observar los registros mostrados en la Figura 30.

Figura 30. Registros al finalizar el entrenamiento (II)

```
Epoch 600/600
600/600 [==============================] - 0s 150us/sample - loss: 0.0115 -
mae: 0.0859 - val_loss: 0.0104 - val_mae: 0.0806
```

En este caso, se observa una disminución de las pérdidas de validación, pasando de obtener 0.1737 en el primer modelo, a 0.0104. Además, el error absoluto medio también cae de 0.3249 a 0.0806, una mejora más que considerable.

Representando gráficamente estos resultados se percibe que las métricas son bastante mejores para la validación que para el entrenamiento, lo que significa que no se está produciendo *overfitting*. Asimismo, el promedio de las pérdidas y el error absoluto medio mejoran claramente con respecto al modelo anterior, como se puede observar en la Figura 31, la Figura 32 y la Figura 33.

Figura 31. Número de epochs - Pérdidas (II) [19]

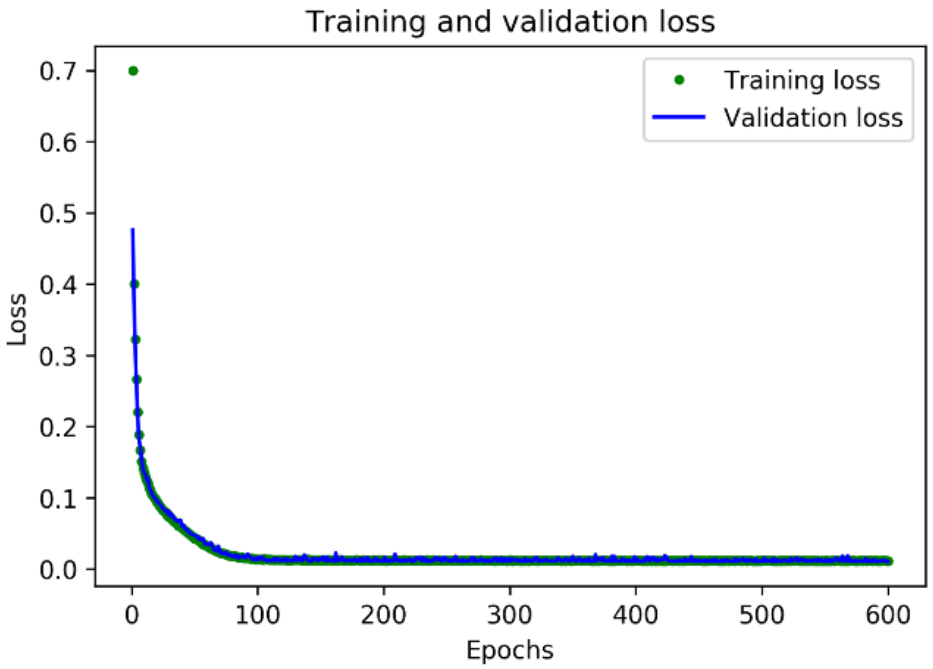


Figura 32. Número de epochs - Pérdidas (Aumentado) (II) [19]

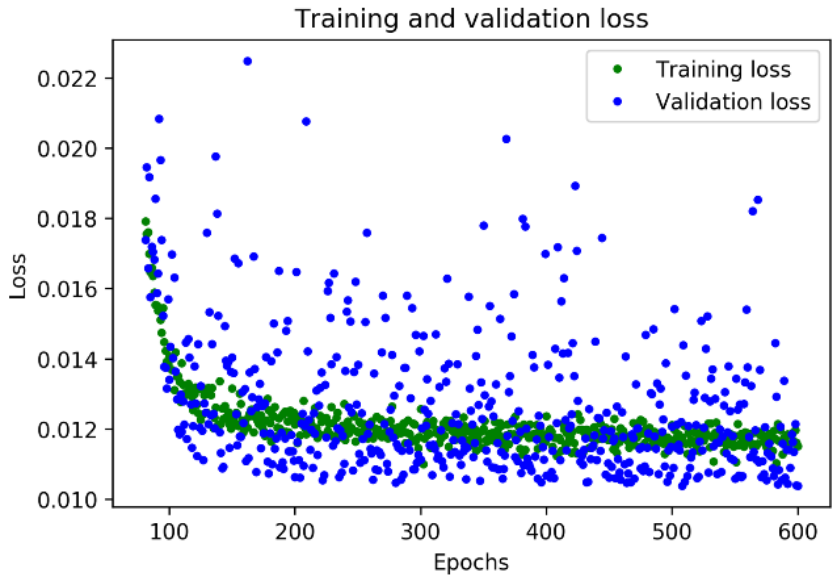
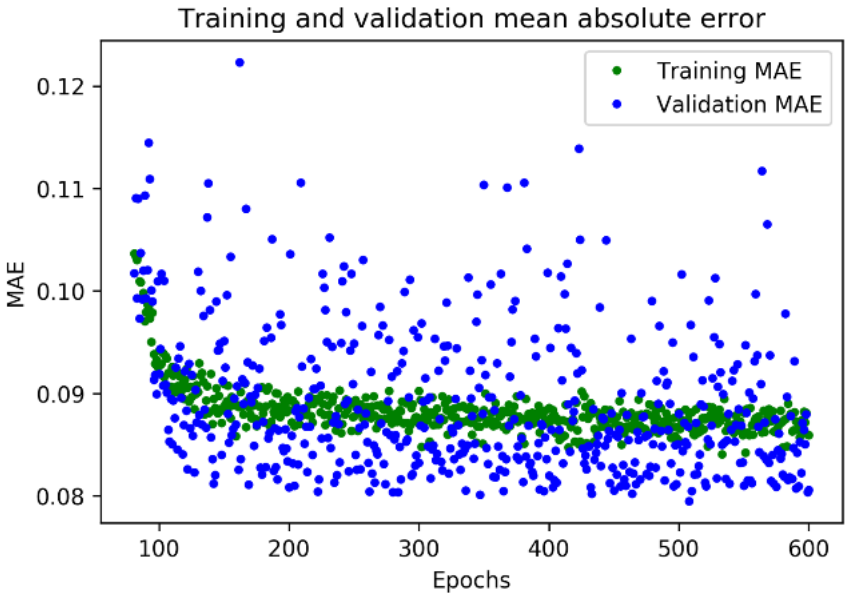


Figura 33. Error absoluto medio (II) [19]



3.2.6 Test final

Una vez que se ajusta el modelo gracias a los datos de validación, es necesario realizar un último test para asegurar su correcto funcionamiento, como se expone en el código de la Figura 34.

Se debe tener mucho cuidado después de utilizar los datos para el test, ya que, si son modificados, es más que probable que se haya producido *overfitting* y, además, no se podría comprobar, puesto que, al modificarse los datos de test, no hay nuevos datos para

volver a testear. Esto significa que, si la prueba da resultados negativos, se ha de volver a atrás y revisar la arquitectura del modelo.

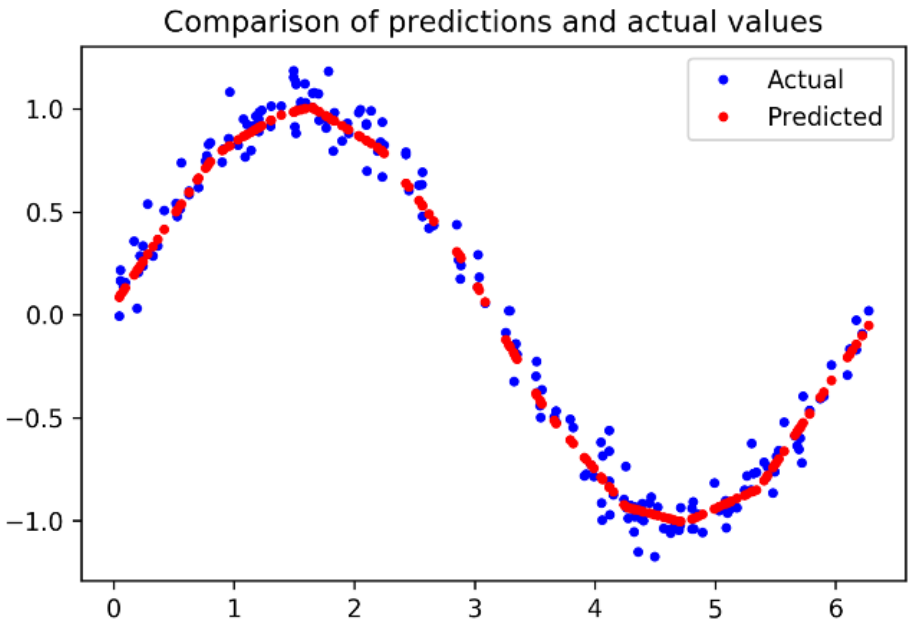
Figura 34. Extracto de código - Test

```
# Calculate and print the loss on our test dataset
loss = model_2.evaluate(x_test, y_test)

# Make predictions based on our test dataset
predictions = model_2.predict(x_test)

# Graph the predictions against the actual values
plt.clf()
plt.title('Comparison of predictions and actual values')
plt.plot(x_test, y_test, 'b.', label='Actual')
plt.plot(x_test, predictions, 'r.', label='Predicted')
plt.legend()
plt.show()
```

En primer lugar, se llama al método `evaluate()` con los datos de test, que calculará e imprimirá por pantalla las pérdidas y error absoluto medio, proporcionando información acerca de cuán diferentes son las predicciones del modelo con respecto a los valores actuales. Posteriormente, se realizan una serie de predicciones y se representa gráficamente.

Para la evaluación se han utilizado 200 puntos de test, que corresponden al conjunto entero de datos. Se observa que las pérdidas son de 0.0103, un valor más que aceptable y muy cercano a las pérdidas de validación (0.0104). Además, el error absoluto medio es de 0.0718, muy similar al 0.0806 obtenido en la validación. Por tanto, se puede concluir que el modelo funciona correctamente y no hay *overfitting*, como se expone en la Figura 35, donde se comparan las predicciones y los valores actuales.

Figura 35. Comparación de predicciones y valores reales [19]



Como se explicó al inicio, el propósito de este modelo es controlar un LED o animación, es por ello que estos resultados son suficientes y las distintas imperfecciones que se observan se considera que no influyen de manera significativa.

3.2.7 Conversión del modelo para TensorFlow Lite

Se hace necesario convertir el modelo para poder ser utilizado con *TensorFlow Lite* y ejecutarlo en un dispositivo basado en MCU. Para ello se usa la API de Python del conversor de *TensorFlow Lite*, que escribe el modelo en disco con la forma *FlatBuffer*, un formato especial designado para mejorar la eficiencia del espacio que ocupa; además, se pueden realizar optimizaciones para reducir el tamaño y/o el tiempo de ejecución. En contraposición, la eficiencia se ve reducida, aunque dicha reducción suele ser aceptable teniendo en cuenta las mejoras que introduce.

Una de las optimizaciones más utilizadas es la cuantificación, que transforma los 32 bits en coma flotante del peso del modelo, en enteros de 8 bits, de forma que la eficiencia apenas se ve afectada.

3.2.8 Conversión a un archivo de C

Para finalizar, se debe preparar el modelo para ser utilizado con *TensorFlow Lite* para Microcontroladores, y para ello se convierte en un archivo fuente de C que pueda ser incluido en la aplicación. Éste se podrá cargar directamente en memoria, ahorrando una gran cantidad de espacio.

En el fichero, el modelo se define como un *array* de bytes y se convierte al formato requerido gracias a la herramienta denominada **xxd**. En el siguiente extracto de código, que

se expone en la Figura 36, se muestra el proceso en el que se ejecuta **xxd** en el modelo cuantificado, escribe la salida en un archivo llamado `sine_model_quantized.cc`, y la imprime por pantalla.

Figura 36. Extracto de código - Conversión a archivo C

```
# Install xxd if it is not available
!apt-get -qq install xxd
# Save the file as a C source file
!xxd -i sine_model_quantized.tflite > sine_model_quantized.cc
# Print the source file
!cat sine_model_quantized.cc
```

La salida resultante se puede ver en la Figura 37, aunque teniendo en cuenta la longitud de ésta, solo se expone el principio y el final.

Figura 37. Salida. Archivo C

```
unsigned char sine_model_quantized_tflite[] = {
  0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x00, 0x00, 0x12, 0x00,
  0x1c, 0x00, 0x04, 0x00, 0x08, 0x00, 0x0c, 0x00, 0x10, 0x00, 0x14, 0x00,
  // ...
  0x00, 0x00, 0x08, 0x00, 0x0a, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x09,
  0x04, 0x00, 0x00, 0x00
};
unsigned int sine_model_quantized_tflite_len = 2512;
```

3.3 Desarrollo de una aplicación con TensorFlow Lite

Una vez concluida la implementación del modelo, en el caso particular del presente TFG será necesaria la creación de una aplicación para poder utilizar dicho modelo. Ésta configurará el entorno adecuado para ejecutarlo, proporcionando valores de entrada al modelo en tiempo real, y usando sus salidas para generar comportamientos o ejecutar acciones.

En este caso, se considera que la aplicación se ejecutará en un dispositivo tipo Arduino, de forma que consta de un *loop* que alimenta el modelo con valores de "x", ejecuta las inferencias, y usa el resultado final para controlar, en este caso, el encendido y apagado de un LED.

3.3.1 Estructura de la aplicación

La implementación de la aplicación se compone de los siguientes archivos, aunque no es necesario mantener esta estructura en todos los casos.

constants.h, constants.cc
Pareja de archivos que contienen las variables constantes que definen el comportamiento del programa que implementa la aplicación.

hello_world_test.cc
Test que ejecuta inferencias usando el modelo desarrollado.

main.cc
Es el punto de entrada del programa, ejecutándose cuando la aplicación se implementa en un dispositivo.

main_functions.h, main_functions.cc
Pareja de archivos que definen la función `setup()`, encargada de llevar a cabo las inicializaciones necesarias, así como una función `loop()`, que contiene la lógica del núcleo del programa. La función `main()` llama a estas funciones al inicio del programa.

output_handler.h, output_handler.cc
Pareja de archivos que definen una función que se utiliza para representar la salida cada vez que se ejecute una inferencia.

output_handler_test.cc
Test que comprueba el correcto funcionamiento del gestor de la salida.

sine_model_data.h, sine_model_data.cc
Pareja de archivos que definen un *array* de datos que representa el modelo.

Una vez se conoce la estructura de la aplicación a desarrollar, se expone el código de los diferentes archivos.

Función principal (*main*):

Define la función `main()` que se ejecuta cuando se inicia el programa. Contiene una declaración `#include` que permite utilizar las funciones `setup()` y `loop()`, como se muestra en la línea de código de la Figura 38, definidas en los ficheros **main_functions**.

Figura 38. Extracto de código - Include

```
#include "tensorflow/lite/micro/examples/hello_world/main_functions.h"
```

Posteriormente se declara la función `main()`, como se expone en el código de la Figura 39, que llama a la función `setup()` para, seguidamente, entrar en un `while` infinito que llama a la función `loop()`.

Figura 39. Extracto de código - Función main

```
int main(int argc, char* argv[]) {
  setup();
  while (true) {
    loop();
  }
}
```

Fichero *main_functions:*

Contiene el núcleo principal del programa. Empieza declarando ciertos `#include`, algunos ya conocidos y otros no tan familiares que se explican a continuación, y que se muestran en la Figura 40.

Figura 40. Extracto de código - Includes main_functions

```
#include "tensorflow/lite/micro/examples/hello_world/main_functions.h"
#include "tensorflow/lite/micro/examples/hello_world/constants.h"
#include "tensorflow/lite/micro/examples/hello_world/output_handler.h"
#include "tensorflow/lite/micro/examples/hello_world/sine_model_data.h"
#include "tensorflow/lite/micro/kernels/all_ops_resolver.h"
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/version.h"
```

`tensorflow/lite/micro/examples/hello_world/sine_model_data.h`
Importa el modelo sinusoidal entrenado, convertido y transformado a C++.

`tensorflow/lite/micro/kernels/all_ops_resolver.h`
Importa la clase que permite al *interpreter* cargar las operaciones usadas en el modelo.

`tensorflow/lite/micro/micro_error_reporter.h`
Importa la clase que permite registrar errores y ofrece la ayuda de la depuración.

`tensorflow/lite/micro/micro_interpreter.h`
Importa el *interpreter* de TensorFlow Lite para Microcontroladores que ejecuta el modelo.

`tensorflow/lite/schema/schema_generated.h`
Importa el esquema que define la estructura de los datos *FlatButter* de TensorFlow Lite, usado para darle sentido a los datos del modelo en `sine_model_data.h`.

`tensorflow/lite/version.h`
Importa el número actual de la versión del esquema, para poder comprobar que el modelo se ha definido con una versión compatible.

A continuación, en el código del fichero **main_functions.cc** se configuran distintas variables globales utilizadas, como se muestra en la Figura 41 .

Figura 41. Extracto de código - Variables globales main_functions

```
namespace {
tflite::ErrorReporter* error_reporter = nullptr;
const tflite::Model* model = nullptr;
tflite::MicroInterpreter* interpreter = nullptr;
TfLiteTensor* input = nullptr;
TfLiteTensor* output = nullptr;
int inference_count = 0;

// Create an area of memory to use for input, output, and intermediate arrays.
// Finding the minimum value for your model may require some trial and error.
constexpr int kTensorArenaSize = 2 * 1024;
uint8_t tensor_arena[kTensorArenaSize];
}  // namespace
```

Como se puede apreciar, éstas se encuentran incluidas en un `namespace`, de forma que solo se puede acceder a ellas desde este archivo, evitando así el problema de que dos archivos distintos definan las variables con el mismo nombre.

```
tflite::ErrorReporter* error_reporter = nullptr;
const tflite::Model* model = nullptr;
```

Se incluye la definición de la clase `ErrorReporter` que permite el envío de errores o mensajes.

```
tflite::MicroInterpreter* interpreter = nullptr;
```

Se configura el *interpreter*. La clase `MicroInterpreter` es una de las clases principales de *TensorFlow Lite* para Microcontroladores, ejecuta el modelo con los datos que se proporcionan. En este caso, se crea un puntero a dicha clase y se inicializa nulo.

```
TfLiteTensor* input = nullptr;
```

Se crea un puntero de entrada nulo.

```
TfLiteTensor* output = nullptr;
```

Se crea un puntero de salida nulo para acceder a la salida del modelo.

```
int inference_count = 0;
```

Entero que lleva la cuenta de las inferencias que realiza el programa.

```
constexpr int kTensorArenaSize = 2 * 1024;
uint8_t tensor_arena[kTensorArenaSize];
```

Se crea un área de memoria que se usará para almacenar la entrada, la salida y los tensores intermedios del modelo. El tamaño depende de la arquitectura del modelo, aconsejándose no reservar más memoria de la necesaria, ya que la RAM es limitada.

Llegados a este punto, la configuración ha finalizado, y el siguiente paso consiste en definir la lógica de la aplicación, ¿qué hace el programa?

Para realizar la demostración del modelo, se alimenta con valores numéricos comprendidos en un rango de 0 a 2π, se predice el valor de la función seno y se obtienen los valores de salida. Esto se realiza mediante la función `loop()`, expuesta en la Figura 42, y cuyo código se ejecuta constantemente; en primer lugar, se determina el valor que se introduce en el modelo (la "x") utilizando dos constantes, `kXrange` que especifica el máximo valor que puede tomar la "x", y `kInferencesPerCycle`, que define el número de inferencias que se quieren realizar mientras se pasa de 0 a 2π.

Figura 42. Extracto de código - Función loop

```
// Calculate an x value to feed into the model. We compare the current
// inference_count to the number of inferences per cycle to determine
// our position within the range of possible x values the model was
// trained on, and use this to calculate a value.
float position = static_cast<float>(inference_count) /
                 static_cast<float>(kInferencesPerCycle);
float x_val = position * kXrange;
```

El resultado obtenido `x_val`, es el valor de entrada del modelo.

Posteriormente, se escribe dicho valor en el tensor de entrada, se ejecuta la inferencia, y se obtiene el resultado del tensor de salida, es decir, el valor de la función seno, como se expone en el extracto de código de la Figura 43. Al estar ejecutándose en un *loop,* se genera una secuencia de valores sinusoidales de forma continua, resultando muy útil para controlar el encendido y apagado de un LED.

Figura 43. Extracto de código - Función loop (II)

```
// Place our calculated x value in the model's input tensor
input->data.f[0] = x_val;

// Run inference, and report any error
TfLiteStatus invoke_status = interpreter->Invoke();
if (invoke_status != kTfLiteOk) {
  error_reporter->Report("Invoke failed on x_val: %f\n",
                         static_cast<double>(x_val));
  return;
}

// Read the predicted y value from the model's output tensor
float y_val = output->data.f[0];
```

El siguiente paso consiste en gestionar dicha salida, lo cual se lleva a cabo a través de la función `HandleOutput()`, definida en el fichero **output_handler.cc**, tal y como se muestra en la Figura 44.

Figura 44. Extracto de código - Función HandleOutput

```
// Output the results. A custom HandleOutput function can be implemented
// for each supported hardware target.
HandleOutput(error_reporter, x_val, y_val);
```

Por último, se incrementa el valor del contador, si alcanza el número máximo de inferencias, definido previamente, se inicializa, como se expone en la Figura 45.

Figura 45. Extracto de código - Función HandleOutput (II)

```
// Increment the inference_counter, and reset it if we have reached
// the total number per cycle
inference_count += 1;
if (inference_count      >= kInferencesPerCycle) inference_count = 0;
```

Fichero *output_handler:*

Este fichero define la función `HandleOutput()`, como se muestra en la Figura 46.

Figura 46. Extracto de código - Función HandleOutput (III)

```
void HandleOutput(tflite::ErrorReporter* error_reporter, float x_value,
                  float y_value) {
  // Log the current X and Y values
  error_reporter->Report("x_value: %f, y_value: %f\n", x_value, y_value);
```

Se utiliza la instancia `ErrorReporter` para registrar los valores de "x" e "y". De esta manera se puede realizar un test básico sobre la funcionalidad de la aplicación. El objetivo es depurar dicha aplicación en diferentes plataformas, usando el hardware de cada una de ellas para representar la salida. Para cada uno de los dispositivos se proporciona un archivo **output_handler.cc** rediseñado que use la API de ésta para controlar la salida.

Una vez ejecutada la aplicación, se obtienen los resultados expuestos en la Figura 47.

Figura 47. Resultados de la aplicación

```
x_value: 1.4137159*2^1, y_value: 1.374213*2^-2

x_value: 1.5707957*2^1, y_value: -1.4249528*2^-5

x_value: 1.7278753*2^1, y_value: -1.4295994*2^-2

x_value: 1.8849551*2^1, y_value: -1.2867725*2^-1

x_value: 1.210171*2^2, y_value: -1.7542461*2^-1
```

Se corresponden los registros escritos por la función `HandleOutput()` en **output_handler.cc**. Se observa uno por inferencia, y el valor de la "x" incrementa gradualmente hasta llegar a 2π, donde retorna al valor 0.

3.4 Implementación en el dispositivo

Para la implementación del modelo en un microcontrolador, se ha elegido, con el fin de ejemplificar este proceso, un dispositivo tipo Arduino compatible con *TensorFlow Lite*. Además, incluye un LED que será el que se utilice para representar visualmente la salida del modelo.

Se sabe que los valores de salida varían en un rango de -1 a 1, por lo que el valor 0 representará el LED apagado, y el 1 y -1 el LED completamente iluminado. Los valores intermedios se representarán con una luz más tenue. Teniendo en cuenta que el programa se ejecuta en un *loop,* el LED se apagará y encenderá de manera continua. Por otro lado, la constante `kInferencesPerCycle` determina el número de inferencias por ciclo, modificando este parámetro se puede ajustar la velocidad de encendido y apagado del LED.

El código que implementa el gestor de la salida en Arduino se encuentra en el fichero **hello_world/arduino/output_handler.cc**, adaptado para este dispositivo en concreto.

En primer lugar, se incluyen archivos de cabecera que proporcionan la interfaz del fichero, la interfaz para la plataforma Arduino y el acceso a las constantes utilizadas.

A continuación, se define la función y lo que debe hacer la primera vez que se ejecute, como se muestra en la Figura 48.

Figura 48. Extracto de código - Función HandleOutput (IV)

```
// Adjusts brightness of an LED to represent the current y value
void HandleOutput(tflite::ErrorReporter* error_reporter, float x_value,
                  float y_value) {
// Track whether the function has run at least once
static bool is_initialized = false;

// Do this only once
if (!is_initialized) {
  // Set the LED pin to output
  pinMode(LED_BUILTIN, OUTPUT);
  is_initialized = true;
}
```

La variable `is_initialized` se utiliza para saber si el código dentro del `if (!is_initialized)` se ha ejecutado ya. Si esto es así, tomará el valor `true` para asegurar que no vuelva a ejecutarse.

La función `pinMode()` indica al microcontrolador si el pin proporcionado como parámetro debe estar en modo entrada o salida, siendo fundamental antes de usarlo. La constante `LED_BUILTIN` representa el pin conectado al LED de la placa, mientras que `OUTPUT` indica el modo salida.

El siguiente paso consiste en calcular el brillo de dicho LED, como se expone en el código de la Figura 49.

Figura 49. Extracto de código - Calculo del brillo del LED

```
// Calculate the brightness of the LED such that y=-1 is fully off
// and y=1 is fully on. The LED's brightness can range from 0-255.
int brightness = (int)(127.5f * (y_value + 1));
```

El código mapea el valor de "y" en un rango de 0 a 255, donde 0 equivale al LED apagado y 255 representa el LED encendido. De esta manera, cuando "y" tome un valor de -1, el LED estará completamente apagado, con y=0 estará encendido de manera tenue, y cuando "y" valga 1 se encenderá por completo.

Una vez calculado, se configura el brillo, como se expone en el código de la Figura 50.

Figura 50. Extracto de código - Configuración del brillo

```
// Set the brightness of the LED. If the specified pin does not support PWM,
// this will result in the LED being on when y > 127, off otherwise.
analogWrite(LED_BUILTIN, brightness);
```

La función `analogWrite()` escoge el número de un pin (determinado por la constante `LED_BUILTIN`) y un valor entre 0 y 255 (`brightness` , calculado previamente).

Por último, se usa la instancia `ErrorReporter` para registrar el valor del brillo, como se muestra en la Figura 51.

Figura 51. Extracto de código - Error Reporter

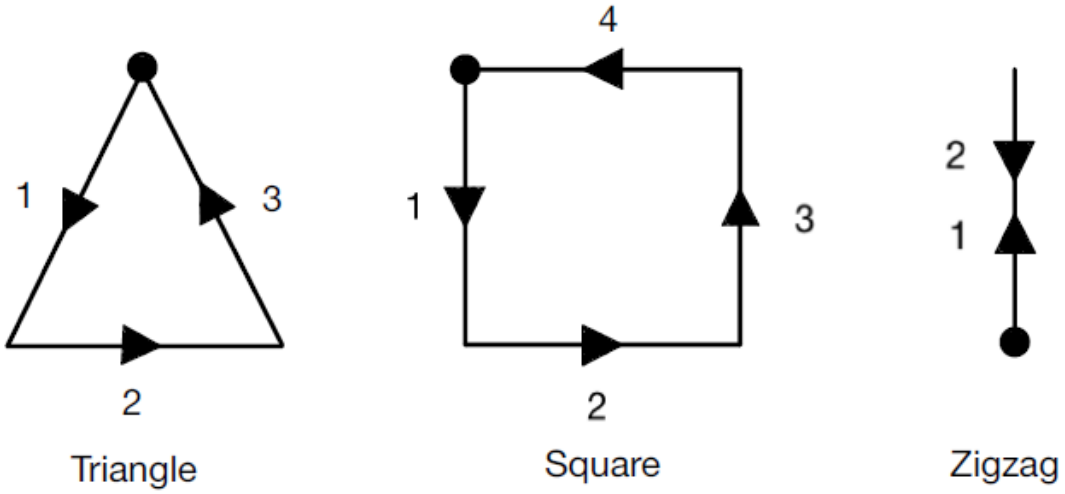
```
// Log the current brightness value for display in the Arduino plotter
error_reporter->Report("%d\n", brightness);
```

Una vez ejecutada la aplicación, se puede observar cómo el LED se enciende y apaga representando la función seno, de más a menos y de menos a más intensidad; cuando el valor de la función es distinto a 0, el LED estará encendido, y cuando dicho valor sea igual a 0, se apagará.

CAPÍTULO 4. Desarrollo del modelo Tensor Flow inicial

El objetivo del presente capítulo es desarrollar un modelo *TensorFlow* capaz de reconocer tres gestos predeterminados realizados en el aire con el dispositivo Argon sujetado con la mano, a partir de la información capturada fundamentalmente por el acelerómetro LSM9DS1. Dichos gestos se muestran en la Figura 52, además, se debe tener en cuenta que, para un correcto reconocimiento de los gestos, se ha de seguir el orden numérico y el sentido que se puede observar para la ejecución de cada uno de ellos.

Figura 52. Gestos a realizar



El primer paso para lograr este objetivo parcial consistió en desarrollar un *sketch* capaz de capturar las muestras de datos generados por el acelerómetro LSM9DS1 durante la realización de estos gestos, mostrándolos a través del puerto serie y almacenándolos para su posterior procesamiento.

A continuación, una vez obtenidos los datos generados por el acelerómetro LSM9DS1 en la realización de los gestos definidos un número mínimo de veces, se desarrollaron varios *scripts* de Python cuya función principal es preprocesar dichos datos, es decir, darles el formato adecuado para la posterior generación y entrenamiento del modelo.

El siguiente paso consistió en es la generación y entrenamiento del modelo, desarrollando igualmente *scripts* de Python que realizan esta función.

Por último, se llevó a cabo la implementación de la aplicación para poder validar experimentalmente el correcto funcionamiento del modelo desarrollado en la plataforma Argon de *Particle*. Dicha aplicación ejecuta las inferencias y utiliza el resultado final para encender el LED de un color diferente en función del gesto realizado en cada caso.

En los siguientes apartados se explica en detalle cada uno de los procesos mencionados.

4.1 Captura y generación de datos

Para obtener los datos de entrenamiento, en el presente TFG se ha desarrollado un *sketch* para el dispositivo Argon de *Particle* que captura los datos generados por el acelerómetro LSM9DS1 y los muestra posteriormente a través del puerto serie. Las medidas del acelerómetro, mostradas a través del terminal serie, se almacenan posteriormente en un fichero de nombre *outuput.txt*, en el formato que establece el *script* de entrenamiento, delimitado por comas. Con este propósito se ha creado el código del *sketch* implementado en el fichero **sketch-lsm9ds1-v01.ino** que se explica a continuación.

4.1.1 Fichero sketch-lsm9ds1-v01.ino

En el código desarrollado se añade, en primer lugar, la librería del acelerómetro LSM9DS1 para el dispositivo Argon, proporcionada por Arduino SA [20] y la propia de *Particle* mediante las sentencias `#include` que se observan en la Figura 53.

Figura 53. Extracto de código - Librerías

```
// This #include statement was automatically added by the Particle IDE.
#include <Arduino_LSM9DS1.h>
#include <Particle.h>
```

A continuación, se definen las constantes a utilizar, como se muestra en el extracto de código de la Figura 54. La constante `NUMBER_VALUES` establece el número de veces que se debe realizar el gesto al presionar el botón programable **SetUp** en la plataforma Argon para registrar los valores proporcionados por el acelerómetro en la realización de cada gesto, mientras que la constante `SAMPLE_PERIOD` determina el periodo de muestreo correspondiente a la frecuencia de muestreo de 25 Hz. Además, se define el pin asociado al LED de la placa Argon.

Figura 54. Extracto de código - Constantes

```
#define NUMBER_VALUES 10 // number of mean values per button press
#define SAMPLE_PERIOD  80    // sample period (ms) = 40ms = 1/25Hz

#define BUTTONPIN D7
```

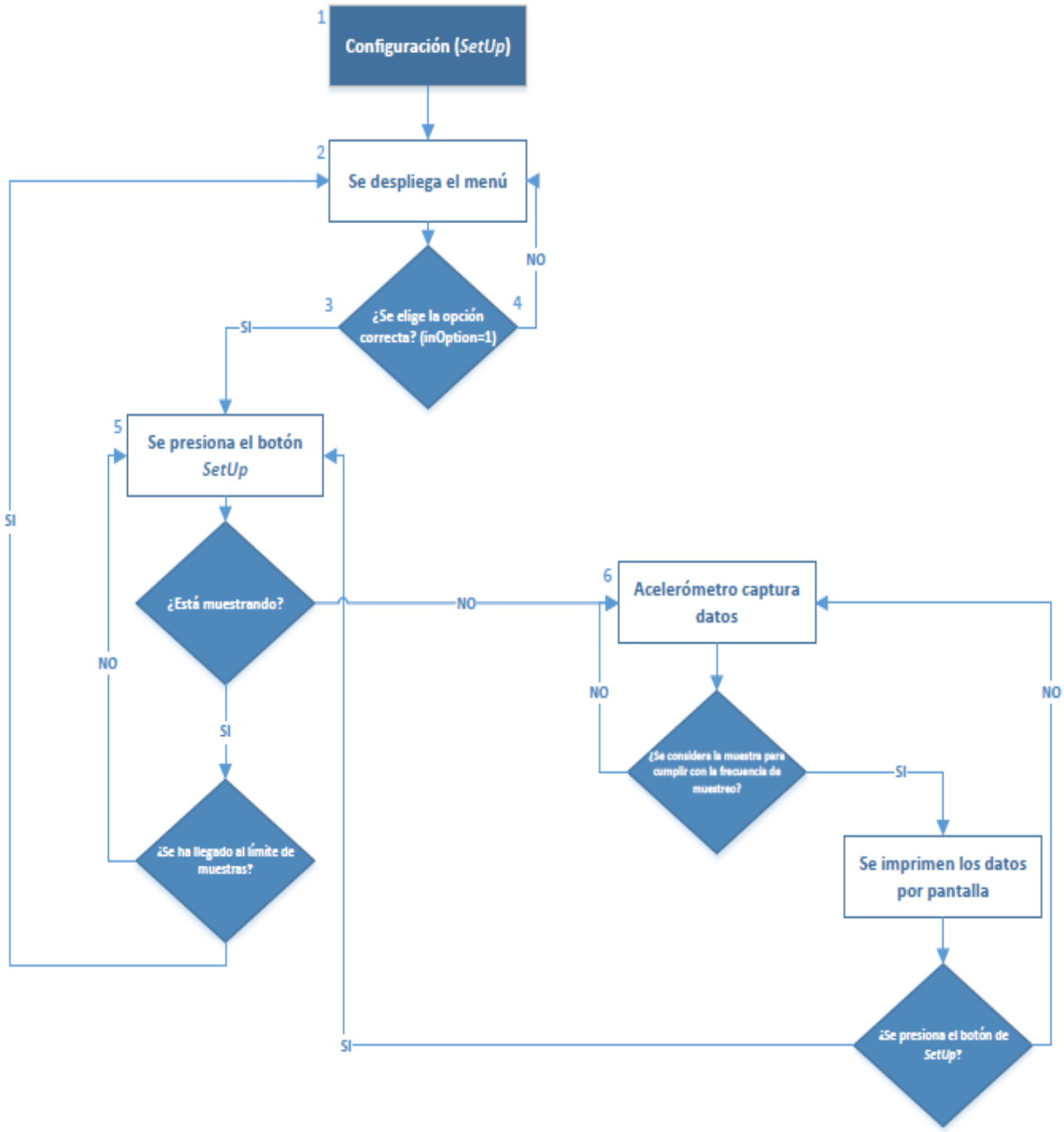
Asimismo, se definen las variables que se utilizarán a lo largo del código, como se observa en el código de la Figura 55.

Figura 55. Extracto de código - Variables

```
bool buttonAlreadyPressed;
char symbol;
bool enableSampling;
bool enableCaptureSampling;
bool enablePrediction;
bool enablePredictGesture;
int n_values;
bool isSampling;
int available;
int sample_every_n;
int sample_skip_counter;

unsigned long actual_time = 0;
unsigned long initial_time = 0;
```

La funcionalidad del código desarrollado para registrar, en el dispositivo Argon, la información del acelerómetro LSM9DS1 en la realización de cada gesto se resume, de manera gráfica, en el diagrama de flujo representado en la Figura 56, incluyendo todos los pasos a seguir, así como las distintas opciones que se pueden dar.

Figura 56. Flujograma del sketch



1. **Configuración (SetUp)**

Se inicializan todas las variables a sus valores iniciales, como se muestra en la Figura 57.

Figura 57. Extracto de código - Inicialización de variables

```
void setup()
{
  Serial.begin(9600);

  enableSampling = false;
  buttonAlreadyPressed = false;
  enableCaptureSampling = false;
  n_values = 0;
  isSampling = false;
  available = 0;
  sample_skip_counter = 1;

  if (!IMU.begin()) {
    Serial.println("Failed to initialize IMU!");
    while (1);
  }
```

Además, se requiere que la frecuencia de muestreo sea de 25 Hz, por lo que el valor de la variable `sample_rate`, que será igual a la frecuencia de muestreo del acelerómetro, se dividirá entre 25 Hz, obteniéndose el intervalo de muestras del acelerómetro de los que debe considerarse una muestra válida para satisfacer la frecuencia de muestreo establecida, tomando una muestra cada `sample_every_n`, tal y como se observa en el código de la Figura 58.

Figura 58. Extracto de código - Configuración de la frecuencia

```
float sample_rate = IMU.accelerationSampleRate();
sample_every_n = static_cast<int>(roundf(sample_rate / 25.0));

Serial.print("Accelerometer sample rate = ");
Serial.print(sample_rate);
Serial.print(" Hz (");
Serial.print(sample_every_n);
Serial.println(")");

Serial.println("Acceleration in mG's");
Serial.println("X\tY\tZ");
```

2. **Se despliega el menú**

A continuación, se desplegará el menú a través del terminal serie para elegir la opción de captura de datos, como se muestra en la Figura 59.

Figura 59. Extracto de código - Menú

```
Serial.println(" ");
Serial.println("MENU:");
Serial.println("-----------------------------------------------");
Serial.println("1. Capture Hand Gesture Data from LSM9DS1 @ 25Hz");
Serial.println("-----------------------------------------------");
Serial.print ("Enter the number corresponding to the menu option to perform: ");
```

Además, se configura el botón **SetUp** del dispositivo Argon, expuesto en la Figura 60, de manera que cuando éste sea pulsado, se invocará la función `button_clicked()` que se describe posteriormente.

Figura 60. Extracto de código - Botón SetUp

```
System.on(button_click, button_clicked);
```

3. Se elige la opción correcta (*inOption = 1*)

Si se elige la opción 1 en el menú, es decir, la captura de datos, la variable `enableSampling`, aquella que habilita el proceso de muestreo, se establecerá a nivel alto (true), y las variables `n_values` (número de capturas por gesto) y `sample_skip_counter` (contador que determina las muestras que no se consideran para cumplir con la frecuencia de muestreo) tomarán los valores por defecto, tal y como se expone en el código de la Figura 61.

Figura 61. Extracto de código - Entrada al loop

```
void loop()
{
  if ((Serial.available() > 0) && (!enableSampling)) {
     char inOption = Serial.read();

     Serial.println(inOption);

     if (inOption == '1') {
        enableSampling = true;
        n_values = 0;
        sample_skip_counter = 1;
```

A continuación, se introduce la letra o número que identifica al gesto que se desea realizar, en este caso, T de triángulo, S de cuadrado o *square* y Z de zigzag, y se mantiene a la espera de poder establecer la comunicación con el puerto serie, tal y como se observa en la Figura 62.

Figura 62. Extracto de código - Introducción de letra que identifica el gesto a realizar

```
Serial.print ("Enter the letter/number for the hand gesture data (T: Triangle, S: Square, Z: ZigZag): ");
while (!(Serial.available() > 0)) Particle.process(); // wait for serial port
char inID = Serial.read();
Serial.println(inID);
symbol = inID; // the symbol associated to the sampled values
```

4. **Se elige la opción incorrecta (*inOption ≠ 1*)**

Se muestra un mensaje en el que se advierte que no es una opción válida, desplegándose posteriormente el menú para elegir una nueva opción, tal y como se aprecia en el código de la Figura 63.

Figura 63. Extracto de código - Opción incorrecta

```
    }
    else {
        Serial.println("Not a valid option");
        Serial.println("");
        Serial.print ("Enter the number corresponding to the menu option to perform: ");
    }
}
```

5. **Se presiona el botón *SetUp***

- **Si no se está muestreando** (`isSampling=False`)

Se habilitará la captura de datos y el muestreo (`enableCaptureSampling = true`). Si la variable `isSampling` (aquella que indica si se está muestreando o no) está a nivel bajo, ésta se activará y se iniciará el contador (`sample_skip_counter`) a su valor inicial. Una vez la variable `isSampling` esté activada, se empezarán a capturar datos del acelerómetro, como se puede observar en la Figura 64.

Figura 64. Extracto de código - Captura de datos

```
////////////////////////////////////////
/ 1. Capture Hand Gesture Data @ 25Hz
////////////////////////////////////////
 if (enableSampling)
 {
   if (enableCaptureSampling)
   {
     if (!isSampling)
     {
       isSampling = true;
       Serial.println("-,-,-");
       sample_skip_counter = 1;
     }
```

- **Si está muestreando** (`isSampling=True`)

En este caso, se detiene la operación de muestreo y se muestra por pantalla el número de muestras que se han obtenido al realizar una vez el gesto (variable `available`). Ésta se inicializa al valor 0 y se incrementa el valor de `n_values` (número de veces que realiza una persona el gesto).

Si se ha llegado al límite de `n_values`, establecido en la constante `NUMBER_VALUES`, se inicializan las demás variables y vuelve a presentarse el menú inicial. En cambio, si aún no se ha superado el límite, se deberá pulsar de nuevo el botón **SetUp** en el dispositivo Argon para seguir capturando datos del acelerómetro en la realización del gesto que corresponde, como se expone en la Figura 65.

Figura 65. Extracto de código - Se detiene el muestreo

```
    else {
      if (isSampling) {

        Serial.println(available);
        Serial.println("");

        available = 0;
        n_values++;
        isSampling = false;
        sample_skip_counter = 1;

        if (n_values == NUMBER_VALUES) // NÚMERO DE MUESTRAS QUE SE DESEAN TOMAR POR CADA PERSONA
        {
          Serial.println(symbol); // print the symbol

          enableSampling = false;
          Serial.println("");
          Serial.println("MENU:");
          Serial.println("--------------------------------------------------");
          Serial.println("1. Capture Hand Gesture Data from LSM9DS1 @ 25Hz");
          Serial.println("--------------------------------------------------");
          Serial.print ("Enter the number corresponding to the menu option to perform: ");
        }
      }
    }
  }
}
```

6. Acelerómetro captura datos

- **Si no se considera la muestra obtenida para cumplir con la frecuencia de muestreo**

Esta situación se produce cuando el valor del contador `sample_skip_counter` es distinto al valor de `sample_every_n` (variable que determina cada cuántas muestras se considera una). El valor del contador `sample_skip_counter` se imprimirá por pantalla y su valor se incrementará en uno, como se observa en el código expuesto en la Figura 66.

Figura 66. Extracto de código - No se considera la muestra obtenida

```
float x, y, z;
if (IMU.accelerationAvailable()) {
  if (IMU.readAcceleration(x, y, z)) {

    if (sample_skip_counter != sample_every_n) {
      Serial.println(sample_skip_counter);
      sample_skip_counter += 1;
    }
```

- **Si se considera la muestra obtenida para cumplir con la frecuencia de muestreo**

Si el valor del contador `sample_skip_counter` es igual al de la variable `sample_every_n`, se representan por pantalla los datos del acelerómetro registrados en la muestra, aumentando en 1 el valor de la variable `available` e inicializando el contador `sample_skip_counter`, como se observa en el extracto de código de la Figura 67.

Figura 67. Extracto de código - Se considera la muestra obtenida

```
    else {
      Serial.print(sample_skip_counter);
      Serial.print(" ->");
      Serial.print(x*1000);
      Serial.print(", ");
      Serial.print(y*1000);
      Serial.print(", ");
      Serial.println(z*1000);

      available++;
      sample_skip_counter = 1;
    }
  }
```

El siguiente mensaje indica que se ha producido un error en la captura de los datos del acelerómetro, como se expone en la Figura 68.

Figura 68. Extracto de código - Error

```
else {
  Serial.println("*******************************");
  Serial.println("*** ERROR READING LSM9DS1 ***");
  Serial.println("*******************************");
}
```

Así, se seguirán capturando muestras del acelerómetro hasta que se vuelva a presionar el botón **SetUp**.

Una vez ejecutado el *sketch* descrito, en el dispositivo Argon, conectado al acelerómetro LSM9DS1 a través de I2C, se pueden observar por el terminal serie los resultados mostrados en la Figura 69.

Figura 69. Extracto de código - Resultados del sketch

```
-,-,-
-766.0,132.0,709.0
-751.0,249.0,659.0
-714.0,314.0,630.0
-709.0,244.0,623.0
-707.0,230.0,659.0
```

La primera columna representa los valores del acelerómetro correspondientes al eje “x”, la segunda los valores en el eje “y”, y la tercera en “z”. Además, los caracteres “-,-,-“ indican el comienzo de las muestras correspondientes a interpretación de un gesto.

4.2 Preprocesamiento de los datos

Una vez obtenidas las muestras correspondientes a la realización de los gestos que se desean reconocer un número de veces determinado, el siguiente paso consiste en procesar los datos previamente a la generación y entrenamiento del modelo *TF Lite*.

La función de estos *scripts* es aumentar y darles el formato adecuado a los datos experimentales obtenidos a partir del *sketch* descrito previamente para la posterior generación y entrenamiento del modelo. Concretamente, el *script* **data_prepare.py** ordena las muestras obtenidas para el proceso de entrenamiento, mientras que el *script* **data_split.py** mezcla las muestras y las separa aleatoriamente en datos para el entrenamiento, datos para la validación y datos para el test.

4.2.1 Script data_prepare.py

El *script* desarrollado a partir de [19] en lenguaje Python lee los archivos que contienen las muestras generadas de sus carpetas, elimina caracteres residuales y los escribe de una manera más ordenada y limpia en un único archivo denominada **complete_data**, en otra carpeta dentro del directorio de los *scripts* de entrenamiento denominado **/data**. En resumen, como su propio nombre indica, su función es preparar y limpiar las muestras obtenidas para el proceso de entrenamiento. Además, este *script* genera datos sintéticos a

partir de las muestras experimentales capturadas inicialmente, es decir, datos generados algorítmicamente en vez de ser capturados, con el fin de aumentar el número de muestras para el proceso de entrenamiento.

Consta de tres funciones principales que son `prepare_original_data()`, `generate_negative_data()` y `write_data()`, que son llamadas de manera secuencial, tal y como se muestra en el código de la Figura 70.

Figura 70. Extracto de código - Llamada a las funciones principales

```
if __name__ == "__main__":
  data = []  # pylint: disable=redefined-outer-name
  for idx1, folder in enumerate(folders):
    for idx2, name in enumerate(names):
      prepare_original_data(folder, name, data,
                            "./%s/output_%s_%s.txt" % (folder, folder, name))
  for idx in range(5):
    prepare_original_data("negative", "negative%d" % (idx + 1), data,
                          "./negative/output_negative_%d.txt" % (idx + 1))
  generate_negative_data(data)
  print("data_length: " + str(len(data)))
  if not os.path.exists("./data"):
    os.makedirs("./data")
  write_data(data, "./data/complete_data")
```

- Función `prepare_original_data()`

Se encarga de leer las muestras experimentales recolectadas y organizarlos para, posteriormente, poder procesarlas. Así, se diferencian los datos obtenidos en función de la carpeta de la que proceda, dividiéndose en dos grupos. Por un lado, los datos provenientes de los tres gestos a identificar, y por otro, los datos que no se relacionan con los gestos estipulados establecidos, que se denominan “negativos”, como se expone en la Figura 71.

Los conjuntos de muestras correspondientes a los datos capturados, así como los correspondientes a los datos *negative* 1-5, se escriben en el fichero de salida en el siguiente formato JSON:

`LABEL_NAME (“gesture”)` es el nombre del fichero asociado a cada gesto, o bien, *negative* `DATANAME (“accel_ms2_xyz”)`.

Figura 71. Extracto de código - Función prepare_original_data

```
def prepare_original_data(folder, name, data, file_to_read):  # pylint: disable=redefined-outer-name
  """Read collected data from files."""
  if folder != "negative":
    with open(file_to_read, "r") as f:
      lines = csv.reader(f)
      data_new = {}
      data_new[LABEL_NAME] = folder
      data_new[DATA_NAME] = []
      data_new["name"] = name
      for idx, line in enumerate(lines):  # pylint: disable=unused-variable,redefined-outer-name
        if len(line) == 3:
          if line[2] == "-" and data_new[DATA_NAME]:
            data.append(data_new)
            data_new = {}
            data_new[LABEL_NAME] = folder
            data_new[DATA_NAME] = []
            data_new["name"] = name
          elif line[2] != "-":
            data_new[DATA_NAME].append([float(i) for i in line[0:3]])
      data.append(data_new)
  else:
    with open(file_to_read, "r") as f:
      lines = csv.reader(f)
      data_new = {}
      data_new[LABEL_NAME] = folder
      data_new[DATA_NAME] = []
      data_new["name"] = name
      for idx, line in enumerate(lines):
        if len(line) == 3 and line[2] != "-":
          if len(data_new[DATA_NAME]) == 120:
            data.append(data_new)
            data_new = {}
            data_new[LABEL_NAME] = folder
            data_new[DATA_NAME] = []
            data_new["name"] = name
          else:
            data_new[DATA_NAME].append([float(i) for i in line[0:3]])
      data.append(data_new)
```

- Función `generate_negative_data()`

Es la función que crea los datos sintéticos, es decir, los datos que equivalen al movimiento del acelerómetro que no corresponde con alguno de los gestos a reconocer. Estos datos se utilizan para entrenar la categoría *unknown*, con el fin de que dicha categoría sea más robusta, sin que contribuya a un mejor reconocimiento de los gestos establecidos como objetivo, como se muestra en el código de la Figura 72 y la Figura 73.

Se generan 100 conjuntos de muestras de datos sintéticos, cada uno de ellos de 128 muestras, asociados, por un lado, a un elevado movimiento elevado, y por otro a un reducido movimiento. En cada caso, el 60% de los conjuntos de muestras de datos sintéticos se etiquetan como *negative* 6, un 20% como *negative* 7 y el 20% restante como *negative* 8.

Figura 72. Extracto de código - Función generate_negative_data

```
def generate_negative_data(data):  # pylint: disable=redefined-outer-name
  """Generate negative data labeled as 'negative6~8'."""
  # Big movement -> around straight line
  for i in range(100):
    if i > 80:
      dic = {DATA_NAME: [], LABEL_NAME: "negative", "name": "negative8"}
    elif i > 60:
      dic = {DATA_NAME: [], LABEL_NAME: "negative", "name": "negative7"}
    else:
      dic = {DATA_NAME: [], LABEL_NAME: "negative", "name": "negative6"}
    start_x = (random.random() - 0.5) * 2000
    start_y = (random.random() - 0.5) * 2000
    start_z = (random.random() - 0.5) * 2000
    x_increase = (random.random() - 0.5) * 10
    y_increase = (random.random() - 0.5) * 10
    z_increase = (random.random() - 0.5) * 10
    for j in range(128):
      dic[DATA_NAME].append([
          start_x + j * x_increase + (random.random() - 0.5) * 6,
          start_y + j * y_increase + (random.random() - 0.5) * 6,
          start_z + j * z_increase + (random.random() - 0.5) * 6
      ])
    data.append(dic)
  # Random
  for i in range(100):
    if i > 80:
      dic = {DATA_NAME: [], LABEL_NAME: "negative", "name": "negative8"}
    elif i > 60:
      dic = {DATA_NAME: [], LABEL_NAME: "negative", "name": "negative7"}
    else:
      dic = {DATA_NAME: [], LABEL_NAME: "negative", "name": "negative6"}
```

Figura 73. Extracto de código - Función generate_negative_data (II)

```
    start_x = (random.random() - 0.5) * 2000
    start_y = (random.random() - 0.5) * 2000
    start_z = (random.random() - 0.5) * 2000
    x_increase = (random.random() - 0.5) * 10
    y_increase = (random.random() - 0.5) * 10
    z_increase = (random.random() - 0.5) * 10
    for j in range(128):
      dic[DATA_NAME].append([
          start_x + j * x_increase + (random.random() - 0.5) * 6,
          start_y + j * y_increase + (random.random() - 0.5) * 6,
          start_z + j * z_increase + (random.random() - 0.5) * 6
      ])
    data.append(dic)
  # Random
  for i in range(100):
    if i > 80:
      dic = {DATA_NAME: [], LABEL_NAME: "negative", "name": "negative8"}
    elif i > 60:
      dic = {DATA_NAME: [], LABEL_NAME: "negative", "name": "negative7"}
    else:
      dic = {DATA_NAME: [], LABEL_NAME: "negative", "name": "negative6"}
    for j in range(128):
      dic[DATA_NAME].append([(random.random() - 0.5) * 1000,
                             (random.random() - 0.5) * 1000,
                             (random.random() - 0.5) * 1000])
    data.append(dic)
# Random
for i in range(100):
  if i > 80:
    dic = {DATA_NAME: [], LABEL_NAME: "negative", "name": "negative8"}
  elif i > 60:
    dic = {DATA_NAME: [], LABEL_NAME: "negative", "name": "negative7"}
  else:
    dic = {DATA_NAME: [], LABEL_NAME: "negative", "name": "negative6"}
  for j in range(128):
    dic[DATA_NAME].append([(random.random() - 0.5) * 1000,
                           (random.random() - 0.5) * 1000,
                           (random.random() - 0.5) * 1000])
  data.append(dic)
# Stay still
for i in range(100):
  if i > 80:
    dic = {DATA_NAME: [], LABEL_NAME: "negative", "name": "negative8"}
  elif i > 60:
    dic = {DATA_NAME: [], LABEL_NAME: "negative", "name": "negative7"}
  else:
    dic = {DATA_NAME: [], LABEL_NAME: "negative", "name": "negative6"}
  start_x = (random.random() - 0.5) * 2000
  start_y = (random.random() - 0.5) * 2000
  start_z = (random.random() - 0.5) * 2000
  for j in range(128):
    dic[DATA_NAME].append([
        start_x + (random.random() - 0.5) * 40,
        start_y + (random.random() - 0.5) * 40,
        start_z + (random.random() - 0.5) * 40
    ])
  data.append(dic)
```

En este caso, los conjuntos de muestras sintéticos se encuentran en el fichero de salida en el siguiente formato JSON `DATA_NAME ("accel_ms2_xyz")`, seguido de los valores correspondientes a las muestras de cada conjunto.

- Función `write_data()`

Función que, como su propio nombre indica, se encarga de escribir los datos obtenidos a partir de los conjuntos de muestras capturados junto con los generados sintéticamente, en el archivo **complete_data** de la carpeta **\data**, como se expone en la Figura 74.

Figura 74. Extracto de código - Función write_data

```
# Write data to file
def write_data(data_to_write, path):
  with open(path, "w") as f:
    for idx, item in enumerate(data_to_write):  # pylint: disable=unused-variable,redefined-outer-name
      dic = json.dumps(item, ensure_ascii=False)
      f.write(dic)
      f.write("\n")
```

De esta manera, tras la ejecución del *script* **data_prepare.py**, se genera el fichero **complete_data** en la carpeta **\data**, conteniendo en el formato especificado los conjuntos de muestras correspondientes a los datos capturados y los datos negativos iniciales, así como los datos negativos generados sintéticamente, mostrándose por pantalla el número total de conjuntos de muestras.

En la Figura 75 se muestra un pequeño extracto del contenido del fichero generado **complete_data**.

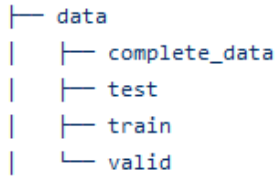
Figura 75. Extracto de código - Fichero complete_data


```
{"gesture": "triangle", "accel_ms2_xyz": [[24.17, -22.95, 968.63], [23.8, -54.81, 1000.61], [19.9, -29.79, 1007.2], [40.77, -53.22, 986.94],
{"gesture": "triangle", "accel_ms2_xyz": [[146.24, -5.25, 971.31], [147.71, -0.24, 967.65], [147.58, 21.48, 997.8], [143.55, 34.06, 1000.49],
{"gesture": "square", "accel_ms2_xyz": [[151.98, 6.47, 968.26], [133.67, 8.91, 967.04], [123.78, 24.29, 976.81], [116.7, -6.47, 921.14], [183.
{"gesture": "square", "accel_ms2_xyz": [[80.93, 39.79, 992.43], [70.43, 21.73, 959.23], [77.64, 27.1, 996.09], [70.43, 56.4, 984.01], [78.12,
{"gesture": "square", "accel_ms2_xyz": [[109.25, 56.15, 959.35], [92.41, 56.88, 975.34], [65.8, 49.44, 1005.49], [102.54, 95.34, 999.51], [79.
{"gesture": "zigzag", "accel_ms2_xyz": [[140.14, 153.44, 1005.49], [104.13, 149.29, 1013.06], [32.59, 91.43, 995.24], [48.83, 92.41, 984.86], [47.85, 92.53, 965
{"gesture": "zigzag", "accel_ms2_xyz": [[60.91, 111.94, 990.6], [73.97, 117.07, 970.34], [48.1, 92.77, 965.7], [42.36, 115.84, 995.73], [39.67, 111.45, 999.15],
{"gesture": "negative", "accel_ms2_xyz": [[0.02, -0.12, -0.72], [-0.01, -0.16, -0.71], [0.0, -0.13, -0.72], [-0.01, -0.05, -0.73], [-0.01, -0.06, -0.75], [-0.02,
{"gesture": "negative", "accel_ms2_xyz": [[-0.12, -0.13, -0.71], [-0.09, -0.11, -0.71], [-0.08, -0.14, -0.74], [-0.08, -0.19, -0.72], [-0.11, -0.3, -0.64], [-0.
{"accel_ms2_xyz": [[768.3482233721289, 238.5735633984065, -180.12059441764785], [772.4290204099102, 234.03988572330067, -184.1678048260298], [771.91170464664328, 2
{"accel_ms2_xyz": [[-560.445950871831, 965.564008035161, 678.0926197969582], [-555.0699361210706, 964.6549311069958, 679.0675799047449], [-557.4490049777216, 971
{"accel_ms2_xyz": [[518.7036495645141, 141.38634701859849, -104.00148666757045], [519.0336054086918, 142.21272970805344, -102.06102376027448], [518.7439789583506
{"accel_ms2_xyz": [[-233.59017236197042, -892.1123752953357, 499.3708450766594], [-237.84346250564056, -891.9042261729531, 501.22320822294085], [-243.64539486932
```


4.2.2 *Script* data_split.py

Es el *script* encargado de mezclar los datos obtenidos y dividirlos aleatoriamente en datos para entrenamiento, datos para validación, y datos para test. Además, estos datos se almacenarán de manera independiente, generándose nuevos ficheros con la estructura que se expone en la Figura 76.

Figura 76. Estructura de ficheros test, train, valid



Para ello, en este *script* de Python se definen dos funciones, `read_data()` y `split_data()`. En el código expuesto en la Figura 77 se observa la llamada a cada una de dichas funciones.

Figura 77. Extracto de código - Llamada a las funciones principales (II)

```
if __name__ == "__main__":
  data = read_data("./data/complete_data")
  train_data, valid_data, test_data = split_data(data, 0.6, 0.2)
  write_data(train_data, "./data/train")
  write_data(valid_data, "./data/valid")
  write_data(test_data, "./data/test")
```

- Función `read_data()`

Se crea un *array* vacío donde, una vez se leen los datos línea a línea desde el fichero **./data/complete_data**, se decodifican del formato JSON original, se almacenan y retornan, indicándose además la longitud de los datos, como se muestra en el extracto de código expuesto en la Figura 78.

Figura 78. Extracto de código - Función read_data

```
# Read data
def read_data(path):
  data = []  # pylint: disable=redefined-outer-name
  with open(path, "r") as f:
    lines = f.readlines()
    for idx, line in enumerate(lines):  # pylint: disable=unused-variable
      dic = json.loads(line)
      data.append(dic)
  print("data_length:" + str(len(data)))
  return data
```

- Función `split_data()`

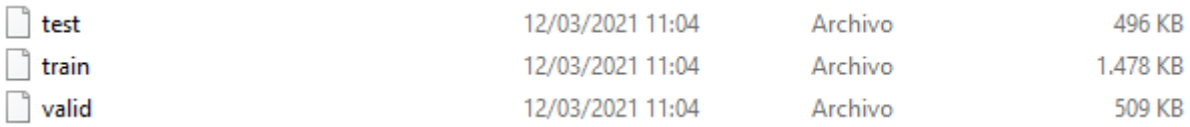
En esta función se aleatorizan los conjuntos de muestras y se separan en entrenamiento, validación y test, en una proporción típica, en los vectores `train_data`, `valid_data` y `test_data`, de 60%, 20% y 20% respectivamente, tal y como se observa en la Figura 79.

Figura 79. Extracto de código - Función split_data

```
def split_data(data, train_ratio, valid_ratio):  # pylint: disable=redefined-outer-name
  """Splits data into train, validation and test according to ratio."""
  train_data = []  # pylint: disable=redefined-outer-name
  valid_data = []  # pylint: disable=redefined-outer-name
  test_data = []  # pylint: disable=redefined-outer-name
  num_dic = {"SymbolD": 0, "SymbolE": 0, "SymbolF": 0, "negative": 0}
  for idx, item in enumerate(data):  # pylint: disable=unused-variable
    for i in num_dic:
      if item["gesture"] == i:
        num_dic[i] += 1
  print(num_dic)
  train_num_dic = {}
  valid_num_dic = {}
  for i in num_dic:
    train_num_dic[i] = int(train_ratio * num_dic[i])
    valid_num_dic[i] = int(valid_ratio * num_dic[i])
  random.seed(30)
  random.shuffle(data)
  for idx, item in enumerate(data):
    for i in num_dic:
      if item["gesture"] == i:
        if train_num_dic[i] > 0:
          train_data.append(item)
          train_num_dic[i] -= 1
        elif valid_num_dic[i] > 0:
          valid_data.append(item)
          valid_num_dic[i] -= 1
        else:
          test_data.append(item)
  print("train_length:" + str(len(train_data)))
  print("test_length:" + str(len(test_data)))
  return train_data, valid_data, test_data
```

En la Figura 80 se muestran los ficheros generados conteniendo de datos de entrenamiento, validación y test.

Figura 80. Carpetas test, validación y entrenamiento



Además, en la Figura 81 se expone como referencia un pequeño extracto del contenido de uno de dichos ficheros.

Figura 81. Extracto del contenido de uno de los ficheros de test, validación y entrenamiento


```
{"gesture": "zigzag", "accel_ms2_xyz": [[44.56, 54.57, 991.58], [11.11, 27.95, 976.32], [19.53, 46.39, 958.86], [-4.52, 59.94, 984.25], [-4.03, 50.66, 993.41], [
{"gesture": "zigzag", "accel_ms2_xyz": [[-26.49, -60.42, 1019.17], [-49.93, -55.54, 984.62], [11.84, 48.1, 993.77], [-53.83, -2.44, 975.1], [-26.61, -37.72, 1017
{"accel_ms2_xyz": [[-642.6176829726517, -73.09749571660551, -683.4564815733122], [-640.4528382124341, -71.9785156628472, -675.2995149486543], [-636.851341510929,
{"accel_ms2_xyz": [[734.3742342708997, 184.2493883725384, 910.159458181741], [745.2419148538322, 157.4718899464375, 900.1956411277749], [746.640936490379, 169.19
{"accel_ms2_xyz": [[616.1581200521318, -711.8619001753208, 676.3983934549543], [615.71511735337, -716.4490224839525, 676.1779863947021], [623.929899257142, -718.
{"accel_ms2_xyz": [[-440.684424555681, 262.46633438439835, 351.154556019481841, [178.9569547567764, -116.6895888151518, -386.5701104331669], [53.78617459534196,
{"accel_ms2_xyz": [[-243.56762159571156, -267.67525539801306, 457.41297553110881, [-316.10871565630003, 303.4394234216912, -275.1968523069944], [160.204179841571
{"accel_ms2_xyz": [[518.7036495645141, 141.38634701859849, -104.00148666757045], [519.0336054086918, 142.21272970805344, -102.06102376027448], [518.7439789583506
{"accel_ms2_xyz": [[350.2779121826193, 133.44019837320653, 169.141674782661281, [-361.77254634284384, 95.91225329328645, 222.289688221786971, [-88.50431944369419
{"accel_ms2_xyz": [[768.3482233721289, 238.5735633984065, -180.120594417647851, [772.4290204099102, 234.03988572330067, -184.1678048260298], [771.911704664328, 2
{"accel_ms2_xyz": [[232.9189606488974, -291.52507512633304, -229.263101052766961, [220.24417389310958, -282.3756103807063, -230.919210495222671, [200.29299656486
{"gesture": "negative", "accel_ms2_xyz": [[0.02, -0.01, 0.97], [0.07, -0.03, 1.0], [0.08, 0.03, 1.0], [0.08, 0.04, 0.94], [0.11, -0.02, 0.98], [0.13, -0.01, 1.0]
{"accel_ms2_xyz": [[-285.83426825412874, 541.8284466595757, 570.2557869205413], [-288.33074592300994, 544.7635915258433, 573.6139633926688], [-292.06683294439404
```

4.3 Generación del modelo TensorFlow

El *script* **train.py** es el encargado de generar el modelo TensorFlow, estableciendo las capas necesarias, así como todos sus parámetros. Además, se basa en dos *scripts* denominados **data_load.py** y **data_augmentation.py.** El primero de ellos es el encargado de implementar la lectura de los conjuntos de muestras en función de la estructura de directorios y ficheros. Asimismo, realiza la llamada al *script* **data_augmentation.py** que, a partir del conjunto de muestras de entrenamiento, genera nuevas versiones de ellas con el fin de sacar el máximo partido al reducido conjunto de muestras iniciales.

A continuación, se explica detalladamente cada uno de estos *scripts* mencionados.

4.3.1 *Script* data_load.py

En el fichero ***data_load.py*** se implementa, en lenguaje Python, la lectura de los conjuntos de muestras de acuerdo con la estructura de directorios y ficheros, asociados a los datos de entrenamiento, validación y test, generados a partir de la ejecución del código presentado en el fichero ***data_split.py***, con el fin de adecuarlos al proceso de generación del modelo *TensorFlow (TF)*.

Así, la clase **DataLoader** definida en el fichero ***data_load.py***, y cuyo código se representa en la Figura 82, se utiliza en el código del archivo ***train.py*** que se presentará posteriormente. Esta clase incluye una serie de funciones para la lectura y adecuación de los diferentes conjuntos de muestras al proceso de entrenamiento y validación del modelo TensorFlow a generar. Como se indica en el código mostrado en `__init__`, es una función de inicialización de la clase **DataLoader** donde se establece el parámetro `dim`, que representa el número de valores de cada una de las muestras de los conjuntos asociados a cada uno de los gestos. Dicho valor es 3, de acuerdo con los valores x,y,z proporcionados por el acelerómetro LSM9DS1 en el proceso de captura de datos. Además, se establece el valor del parámetro `seq_length`, que representa el número máximo de muestras establecido para cada conjunto, que en este caso particular será de 128. Adicionalmente, desde esta función se hace una llamada a la función `get_data_file` con el fin de cargar los conjuntos de etiquetas asociados a cada uno de los gestos que se verán en los procesos de entrenamiento, validación y test del modelo TF, a partir de las rutas proporcionadas como parámetros en `train_data/train_label`, `valid_data/valid_label` y `test_data/test_label`, respectivamente.

Figura 82. Extracto de código - Clase DataLoader

```
class DataLoader(object):
"""Loads data and prepares for training."""

def __init__(self, train_data_path, valid_data_path, test_data_path,
seq_length):
self.dim = 3
self.seq_length = seq_length
self.label2id = {"triangle": 0, "square": 1, "zigzag": 2, "negative": 3}

self.train_data, self.train_label, self.train_len = self.get_data_file(
train_data_path, "train")
self.valid_data, self.valid_label, self.valid_len = self.get_data_file(
valid_data_path, "valid")
self.test_data, self.test_label, self.test_len = self.get_data_file(
test_data_path, "test")
```

Por un lado, la función `get_data_file` es la que obtiene, por filas, los conjuntos de muestras asociados a los procesos de entrenamiento, validación y test, junto a sus etiquetas, a partir de los ficheros generados tras la ejecución del código implementado en el fichero **data_split.py**. Así, una vez finalizado el proceso de carga en los *arrays* `data[]` y `label[]` para cada uno de los tipos de conjuntos de muestras asociados a los procesos de entrenamiento, validación y test, se mostrará por pantalla, en cada caso, el número de conjuntos de muestras determinado por el valor de la variable `length`, proporcionando como salida el conjunto de muestras asociado a cada tipo junto a sus etiquetas y su tamaño, como se muestra en la Figura 83.

Figura 83. Extracto de código - Función get_data_file

```
def get_data_file(self, data_path, data_type):
"""Get train, valid and test data from files."""
data = []
label = []
with open(data_path, "r") as f:
lines = f.readlines()
for idx, line in enumerate(lines): # pylint: disable=unused-variable
dic = json.loads(line)
data.append(dic[DATA_NAME])
label.append(dic[LABEL_NAME])
if data_type == "train":
data, label = augment_data(data, label)
length = len(label)
print(data_type + "_data_length:" + str(length))
return data, label, length
```

En el caso particular de los conjuntos de muestras asociados al proceso de entrenamiento del modelo, con `data_type="train"`, en la función `get_data_file()` se realiza, adicionalmente, un proceso de *data augmentation* implementado en el fichero **data_augmentation.py**, que se describirá posteriormente, en el que a partir de los conjuntos de muestras originales asociados a cada gesto, se genera un número de nuevas

versiones, cada una de ellas modificadas ligeramente. Estos conjuntos de muestras aumentadas se utilizan, junto con las originales, para entrenar el modelo TF, contribuyendo a sacar el máximo partido al reducido número de conjuntos de muestras de partida.

Por otro lado, la función `format()` realiza la adecuación del formato de los conjuntos de muestras asociados al entrenamiento, validación y test, generando el conjunto de muestras en el formato adecuado para la creación del modelo TF. En esta función se realiza, en cada caso, una llamada a la función auxiliar `format_support_func()` para formatear los conjuntos de muestras asociados a los procesos de entrenamiento, validación y test, además de añadir un *padding* anterior y posterior a los conjuntos de muestras originales mediante la función `pad()`, con `padded_num=2`. Para cada caso, la función `format_support_func()` inicializa, en primer lugar, `2*length` matrices, siendo `length` el tamaño del conjunto de muestras de cada tipo, cada una de las cuales tiene un tamaño de `seq_length(128)*3(dim)`, denominada `features`, con el valor inicial 0, así como el valor `labels`, de tamaño `2*length`, como se muestra en la sección de código de la Figura 84.

Figura 84. Extracto de código - Función format

```
def format(self):
"""Format data(including padding, etc.) and get the dataset for the model."""
padded_num = 2
self.train_len, self.train_data = self.format_support_func(
padded_num, self.train_len, self.train_data, self.train_label)
self.valid_len, self.valid_data = self.format_support_func(
padded_num, self.valid_len, self.valid_data, self.valid_label)
self.test_len, self.test_data = self.format_support_func(
padded_num, self.test_len, self.test_data, self.test_label)
```

Una vez completado el proceso de *padding*, en la función `format_support_func()`, cuyo código se muestra en la Figura 85, se crea un *DataSet* de TF a partir de la dupla de matrices generadas y sus etiquetas correspondientes, que junto al número de conjuntos de muestras resultantes, es lo que se proporciona como resultado a la función `format()` para cada uno de los tipos de conjuntos de muestras.

Figura 85. Extracto de código - Función format_support_func

```
def format_support_func(self, padded_num, length, data, label):
"""Support function for format.(Helps format train, valid and test.)"""
# Add 2 padding, initialize data and label
length *= padded_num
features = np.zeros((length, self.seq_length, self.dim))
labels = np.zeros(length)
# Get padding for train, valid and test
for idx, (data, label) in enumerate(zip(data, label)):
padded_data = self.pad(data, self.seq_length, self.dim)
for num in range(padded_num):
features[padded_num * idx + num] = padded_data[num]
labels[padded_num * idx + num] = self.label2id[label]
# Turn into tf.data.Dataset
dataset = tf.data.Dataset.from_tensor_slices(
(features, labels.astype("int32")))
return length, dataset
```

En la función `pad()`,cuyo código se muestra en la Figura 86, se genera, en el primer *padding*, una matriz denominada `tmp_data` de tamaño `seq_length(128)` muestras, de `dim(3)` valores cada una, inicializada a partir de los valores de la primera de las muestras del conjunto (`data[0]`) y valores aleatorios (entre -0.5 y +0.5), obtenidos a partir del producto de números multiplicados por el nivel de ruido definido en la variable `noise_level`, que en este caso se ha establecido, por defecto, al valor `20`.

A continuación, para cada uno de los conjuntos de muestras generados en `tmp_data`, se sustituyen los valores desde la muestra `(seq_length - min(len(data), seq_length)` hasta la muestra `seq_length(128)`, por las muestras del conjunto de partida, desde cero hasta `min(len(data), seq_length)`, manteniendo el resto de valores generados aleatoriamente, lo que resulta en un *padding* anterior que permite ajustar el tamaño de todos los conjuntos demuestra al valor `seq_length(128)`.
En el segundo *padding*, se genera una nueva matriz denominada, igualmente, `tmp_data` de tamaño de `seq_length(128)` muestras, de `dim(3)` valores cada una, pero inicializada, esta vez, a partir de los valores de la última de las muestras del conjunto (`data[-1]`) y valores obtenidos a partir del producto de números aleatorios comprendidos entre -0.5 y +0.5, multiplicados por el nivel de ruido definido en `noise_level`.

Además, para cada uno de los conjuntos de muestras generados en `tmp_data`, se sustituyen los valores desde la muestra `0` hasta la muestra `min(len(data), seq_length)` por las muestras del conjunto original, desde `0` hasta `min(len(data), seq_length)`, manteniendo el resto de valores generados aleatoriamente, lo que resulta, en este caso, en un *padding* posterior que permite, igualmente, ajustar el tamaño de todos los conjuntos de muestras al valor `seq_length(128)`.

Figura 86. Extracto de código - Función pad

```
def pad(self, data, seq_length, dim):
"""Get neighbour padding."""
noise_level = 20
padded_data = []
# Before- Neighbour padding
tmp_data = (np.random.rand(seq_length, dim) - 0.5) * noise_level + data[0]
tmp_data[(seq_length -
min(len(data), seq_length)):] = data[:min(len(data), seq_length)]
padded_data.append(tmp_data)
# After- Neighbour padding
tmp_data = (np.random.rand(seq_length, dim) - 0.5) * noise_level + data[-1]
tmp_data[:min(len(data), seq_length)] = data[:min(len(data), seq_length)]
padded_data.append(tmp_data)
return padded_data
```

4.3.2 *Script* data_augmentation.py

En el fichero **data_augmentation.py** se implementa la función `augment_data()` que, a partir del conjunto de muestras originales asociado al proceso de entrenamiento, genera un número de nuevas versiones de ellas, cada una modificada ligeramente.

Estas modificaciones incluyen desplazamiento y traslación temporal, así como la adición de ruido aleatorio y el incremento de la cantidad de aceleración. Este conjunto de muestras aumentadas, junto con el conjunto de muestras de partida, se usará para entrenar el modelo TF a generar, sacando el mayor partido posible al reducido conjunto de muestras iniciales.

Así, en la función `augment_data()` se generan, inicialmente, cinco nuevas versiones de cada uno de los conjuntos de muestras mediante desplazamiento temporal, sumando al valor de cada muestra original un valor aleatorio comprendido entre -0,5 y +0,5 multiplicado por la constante 200, como se muestra en el código de la Figura 87.

Figura 87. Extracto de código - Función augment_data

```
def augment_data(original_data, original_label):
  """Perform data augmentation."""
  new_data = []
  new_label = []
  for idx, (data, label) in enumerate(zip(original_data, original_label)):  # pylint: disable=unused-variable
    # Original data
    new_data.append(data)
    new_label.append(label)
    # Sequence shift
    for num in range(5):  # pylint: disable=unused-variable
      new_data.append((np.array(data, dtype=np.float32) +
                       (random.random() - 0.5) * 200).tolist())
      new_label.append(label)
```

A continuación, se generan cinco nuevas versiones de cada uno de los conjuntos de muestras mediante la adición de ruido aleatorio, siendo generado el valor de cada muestra a partir de la suma del valor original y un valor aleatorio comprendido entre 0 y 1, multiplicado por la constante 5, como se muestra en el código de la Figura 88.

Figura 88. Extracto de código - Ruido aleatorio

```
# Random noise
tmp_data = [[0 for i in range(len(data[0]))] for j in range(len(data))]
for num in range(5):
  for i in range(len(tmp_data)):
    for j in range(len(tmp_data[i])):
      tmp_data[i][j] = data[i][j] + 5 * random.random()
  new_data.append(tmp_data)
  new_label.append(label)
```

Posteriormente, a partir de la función `time_wrapping()` se generan nuevas versiones de cada uno de los conjuntos de datos de muestras mediante *time warping*, siendo generado el valor de la muestra `[denominator * i + k][j]` a partir de la adición del valor original de la muestra `[molecule * i + k][j]` multiplicado por el factor `(denominator - k)`. Además, el valor original `[molecule * i + k + 1][j]` se multiplica por el factor `k / denominator` con `k in range (denominator)`.

Los valores `molecule` y `denominator` se corresponden en cada caso con los definidos en el vector multidimensional `fractions`. a partir de [19], como se muestra en el código de la Figura 89.

Figura 89. Extracto de código - Time warping

```
# Time warping
fractions = [(3, 2), (5, 3), (2, 3), (3, 4), (9, 5), (6, 5), (4, 5)]
for molecule, denominator in fractions:
  new_data.append(time_wrapping(molecule, denominator, data))
  new_label.append(label)

def time_wrapping(molecule, denominator, data):
  """Generate (molecule/denominator)x speed data."""
  tmp_data = [[0
               for i in range(len(data[0]))]
              for j in range((int(len(data) / molecule) - 1) * denominator)]
  for i in range(int(len(data) / molecule) - 1):
    for j in range(len(data[i])):
      for k in range(denominator):
        tmp_data[denominator * i +
                 k][j] = (data[molecule * i + k][j] * (denominator - k) +
                          data[molecule * i + k + 1][j] * k) / denominator
  return tmp_data
```

Finalmente, se generan nuevas versiones de cada uno de los conjuntos de muestras mediante la amplificación del movimiento, siendo generado el valor de cada muestra a partir de la adición del valor original y la relación `molecule/denominator` definidos en el vector multidimensional `fractions`, como se observa en la Figura 90.

Figura 90. Extracto de código - Amplificación del movimiento

```
  # Movement amplification
  for molecule, denominator in fractions:
    new_data.append(
        (np.array(data, dtype=np.float32) * molecule / denominator).tolist())
    new_label.append(label)
return new_data, new_label
```

4.3.3 *Script* train.py

Este *script* tiene como función principal generar el modelo *TF Lite*, estableciendo todas sus capas y sus diferentes parámetros. Para ello, cuenta con dos funciones principales. La función `build_cnn()`, que es la encargada de generar el modelo a partir de la construcción de la red neuronal y estableciendo sus capas y algunos de sus parámetros. La función `train_net()` se encarga, principalmente, de realizar el proceso de entrenamiento, así como de establecer sus parámetros.

- Función `build_cnn()`

Existen dos formas de operar modelos *Keras*, de forma **secuencial**, que es la más básica, y **funcional**, orientada a redes complejas. En el caso particular del presente TFG se hará uso de la primera de ellas, y se debe saber que un modelo secuencial consiste en una pila lineal de capas. Esto significa que la API secuencial permite crear modelos capa a capa, lo cual resulta ventajoso en el desarrollo de modelos de *Deep learning.* Una de las principales características de este tipo de modelos es que la salida de cada capa se pasa directamente a la entrada de la siguiente.

Capa *Conv2D*

En el caso concreto de la creación del modelo *Keras* definido en el fichero **train.py**, la primera capa se corresponde con una convolución estándar con 8 filtros, cada uno de los cuales tiene un Kernel de tamaño (4,3), como se muestra en el código de la Figura 91.

Figura 91. Extracto de código - Capa Conv2D

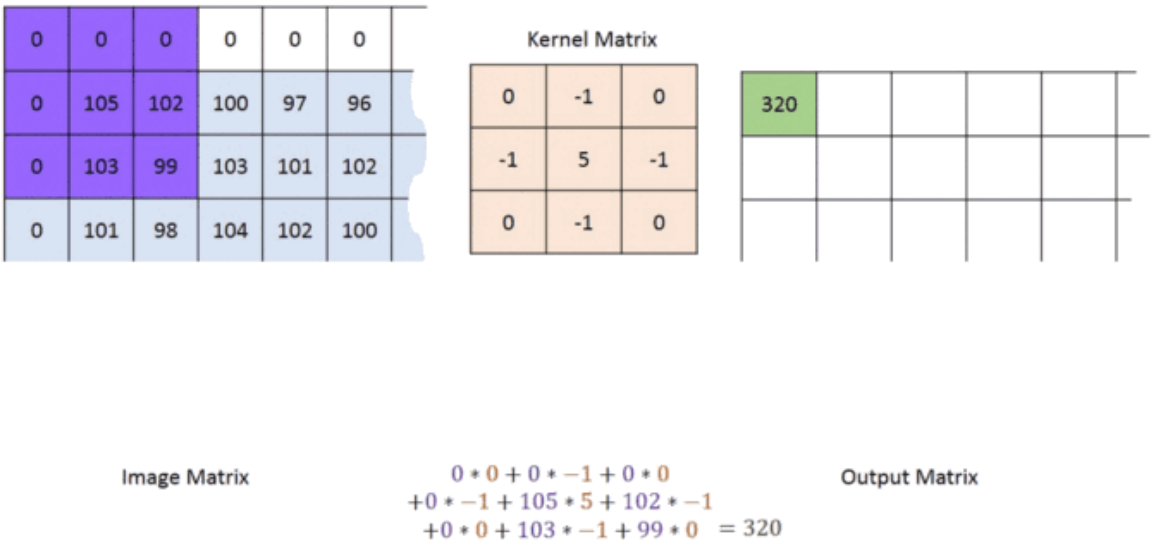
```
tf.keras.layers.Conv2D(
    8, (4, 3),
    padding="same",
    activation="relu",
    input_shape=(seq_length, 3, 1))
```

La función principal es disponer de los datos crudos de entrada y extraer de su procesamiento algunas características básicas que puedan ser interpretadas por las capas

ulteriores. Así, cada uno de los 8 filtros se dedica a identificar algunos aspectos de los datos de entrada. Durante la fase de entrenamiento, cada filtro aprende a identificar una característica particular de dichos datos, por ejemplo, un filtro puede aprender a identificar los indicios de un movimiento asociado a un eje del acelerómetro. Los patrones que codifican los filtros son más complejos a medida que se profundiza en las capas convolucionadas de la red. Cabe destacar que, para cada filtro, la capa proporciona un mapa de características (*features map)* que representa, en los datos de entrada, cuándo se produce la particularidad que ha aprendido.

Por otro lado, un Kernel es un *grid* de valores que se pasa sobre el conjunto de datos de entrada y los transforma en función de los valores contenidos en dicho *grid*, tal y como se expone gráficamente en la Figura 92.

Figura 92. Ejemplo del Kernel [21]



Cuando el Kernel se encuentre en el borde de los datos de entrada debe tomar una decisión cuando no existan valores, rellenándose en este caso con 0, de acuerdo a un proceso llamado *zero padding*, para lo cual se incluye en el código la opción `padding="same"`. De esta manera se asegura que la dimensión de los datos de entrada no se reduzca después de la convolución.

Por otro lado, el parámetro `input_shape` puede interpretarse como la forma de un conjunto de datos de entrada que se introduce en la red, siendo en este caso de (128x3x1).

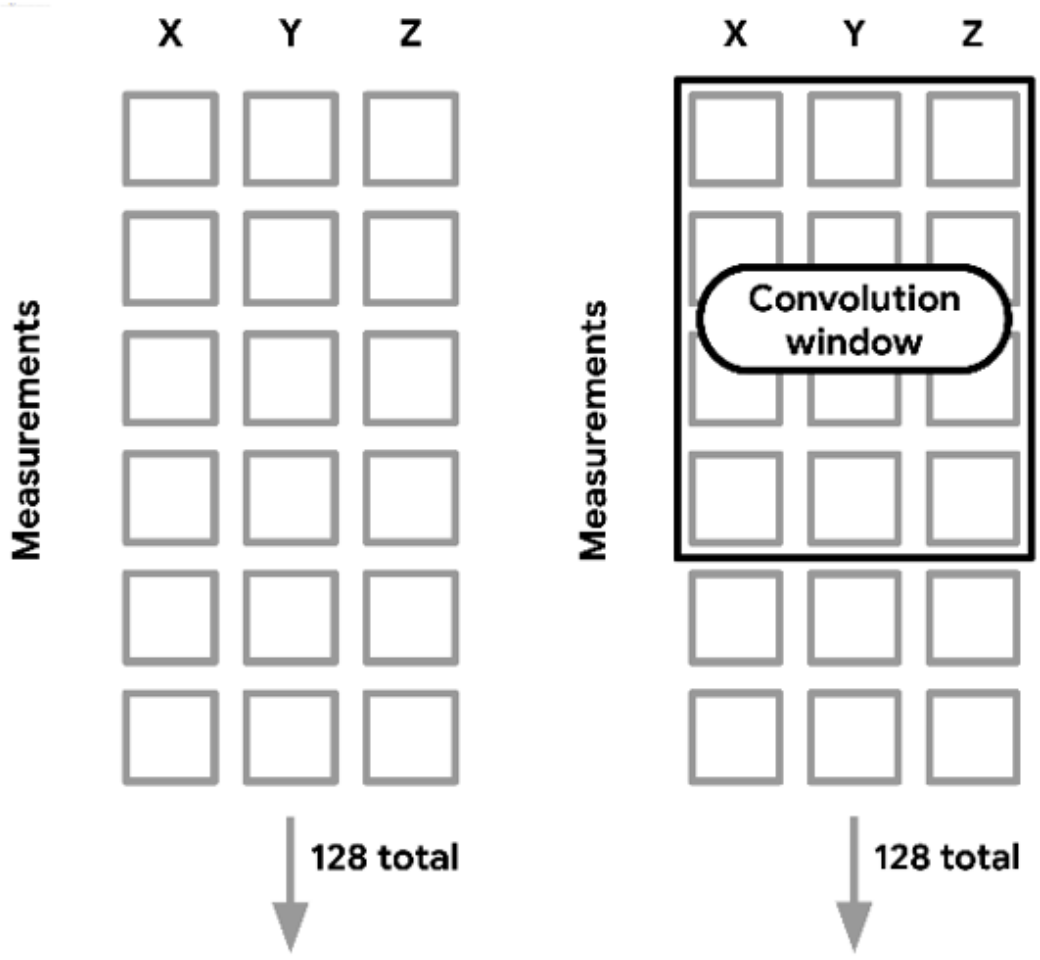
Finalmente, la función de activación `relu` (*Rectified Linear Unit)* devuelve el valor proporcionado como entrada directamente, o el valor 0,0 si la entrada es 0,0 o inferior. Dicha función de activación asegura que los valores que se pasan se encuentran dentro de un rango determinado, en este caso discriminando los valores negativos.

Con esto, la definición de la primera capa del código tiene 8 filtros, de forma que aprenderá a reconocer 8 tipos diferentes de características de los datos de entrada, representándolo

también el parámetro `output_shape`, donde el último valor (8) indica que se generan 8 *features channels*, uno para cada particularidad. El valor en cada uno de estos canales indica el grado en el que una característica está presente en los datos de entrada.

Esta capa convolucional se basa en desplazar un kernel de un determinado tamaño sobre los datos de entrada, con el fin de decidir si una determinada característica está presente en esa ventada o no. El tamaño se especifica en el segundo parámetro de dicha capa, que es de (**4**,**3**), lo cual implica que las particularidades de los datos de entrada que buscan los 8 filtros comprenden **4** medidas consecutivas de los **3** valores asociados a los ejes x,y,z del acelerómetro. Puesto que el Kernel comprende 4 medidas consecutivas, cada uno de los 8 filtros analiza un breve intervalo de tiempo de los datos de entrada, lo cual implica que puede generar características que representen un cambio en los datos de entrada a lo largo del tiempo. En la Figura 93 se expone un ejemplo de ventana convolucional.

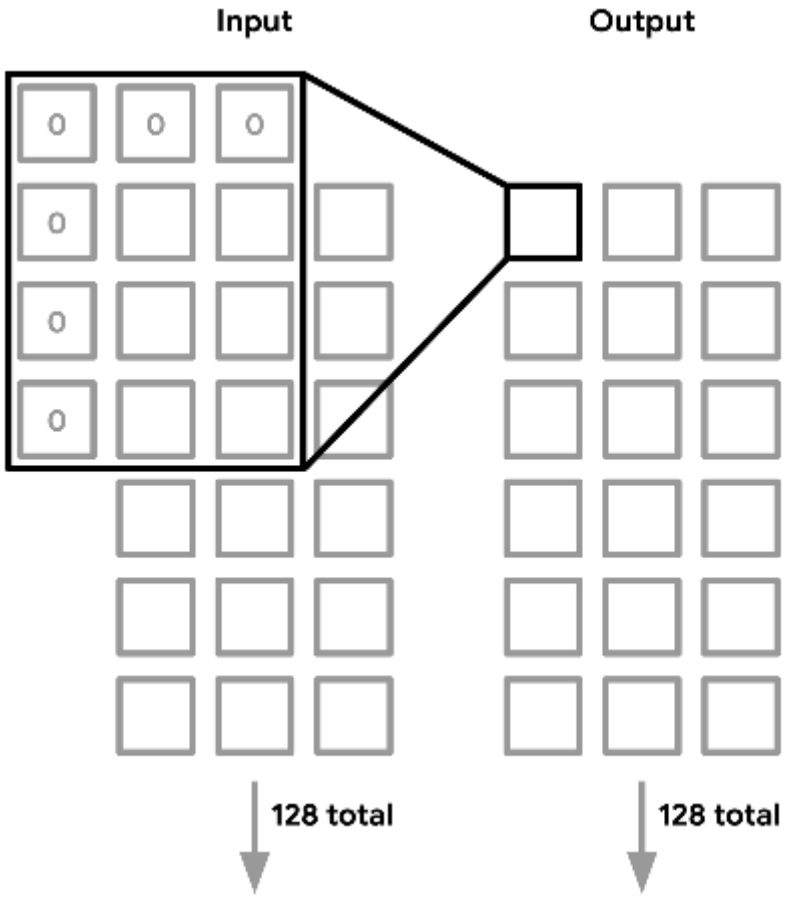
Figura 93. Ventana de convolución [19]



El argumento `padding` determina cómo se desplaza el kernel a través de los datos de entrada. Cuando se establece a "`same`", como en este caso, la salida de la capa tendrá el mismo tamaño que los datos de entrada (128x3). Puesto que cada posición del Kernel genera un valor de salida, el parámetro "`same`" implica que dicho Kernel debe desplazarse 3 veces a través de los datos, y 128 veces hacia abajo. Así, teniendo en cuenta que el ancho

del Kernel es 3, debe comenzar por el extremo superior izquierdo de los datos de entrada, de forma que los espacios vacíos, donde el Kernel no cubre un valor, son rellenados con ceros (*zero padding)*, tal y como se expone en la Figura 94. Del mismo modo, para desplazarse un total de 128 veces hacia abajo, el Kernel también debe cubrir los datos localizados en la parte superior.

Figura 94. Desplazamiento del Kernel a través de los datos de entrada [19]



Capa *MaxPool2D*

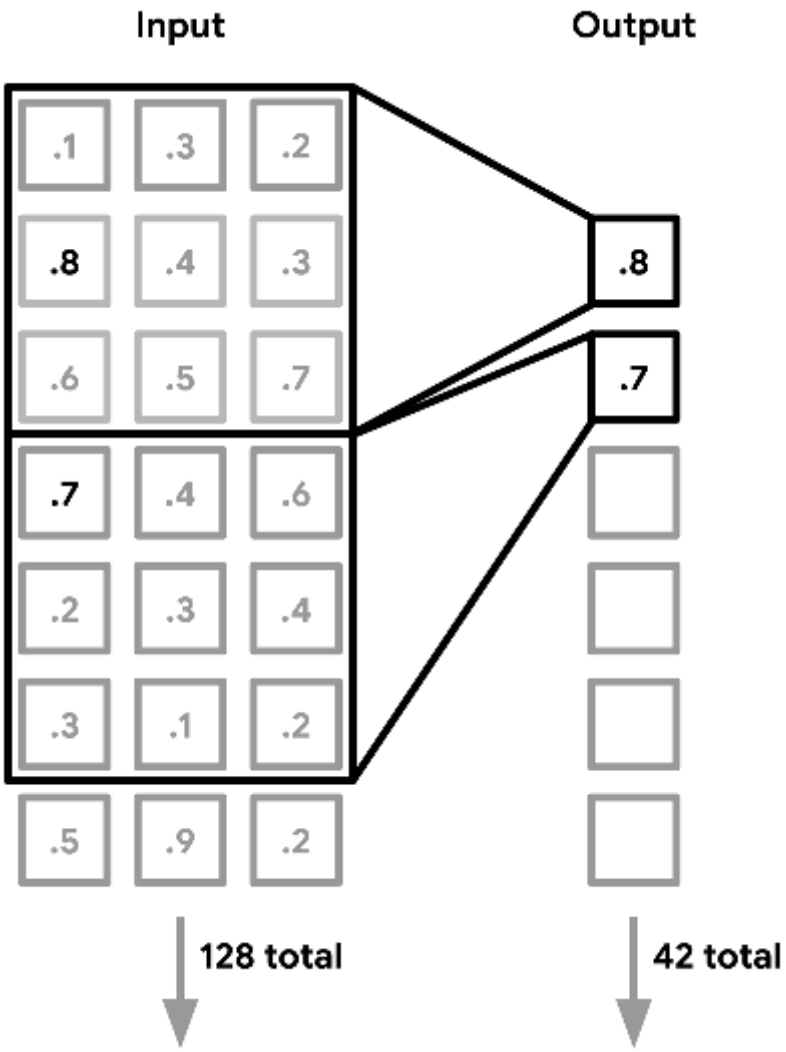
La segunda capa del modelo, una vez el Kernel se ha desplazado por todos los datos de entrada usando cada filtro para crear 8 mapas de características diferentes, consiste en una función de *max-pooling* que toma la salida de la capa anterior, un tensor de **(128, 3, 8)**, y reduce su dimensión a un tensor de **(42,1,8)**, lo que corresponde en este caso a 1/3 de su tamaño original. En la Figura 95 se muestra la sección de código donde se crea dicha capa.

Figura 95. Extracto de código - Capa MaxPool2D

```
tf.keras.layers.MaxPool2D((3, 3))
```

Este proceso se realiza considerando los datos de entrada filtrados y contenidos en una ventana, en este caso, de tamaño **(3x3)**, y representando estos valores de salida por un único valor, correspondiente, en este caso, al máximo de los valores contenidos en dicha ventana, como se representa de forma gráfica en la Figura 96.

Figura 96. Ventana de 3x3 [19]



Este proceso se repite con la siguiente ventana de datos filtrados, de manera que, por defecto, la ventana se desplaza de forma que contenga datos de entrada nuevos hasta cubrir la totalidad. Se ha de tener en cuenta que los datos de entrada contienen 8 canales de características por elemento, por lo que este proceso se realiza en todos ellos.

Puesto que el propósito de un CNN es transformar un tensor de entrada grande y complejo en un tensor de salida pequeño y simple, la capa ***MaxPool2D*** contribuye a su consecución, reduciendo la salida de la primera capa de convolución en una representación concentrada de alto nivel de la información relevante que contiene.

Así, concentrando la información, se empiezan a eliminar elementos que no resultan relevantes para la tarea de identificar qué gesto está contenido en los datos de entrada. Por

tanto, aunque los datos de entrada originales están formados por 3 valores (ejes x,y,z) para cada medida, la combinación de las capas ***Conv2D*** y ***MaxPool2D*** los ha combinado en 1 valor cada 3 medidas.

Capa *Dropout*

Esta capa establece, aleatoriamente, algunos valores del tensor a 0 durante el entrenamiento; en este caso, mediante el parámetro 0,1 se fija un 10% de los valores a 0, como se indica en la sentencia de la Figura 97.

Figura 97. Extracto de código - Capa Dropout

```
tf.keras.layers.Dropout(0.1)
```

Esta es una técnica de regularización que consiste en mejorar los modelos de *Machine Learning* limitando los problemas de *overfitting*, forzando a la red neuronal a aprender cómo lidiar con variaciones y ruidos inesperados. Cabe destacar que esta capa solo se encuentra activa en el proceso de entrenamiento, de manera que no tiene ningún efecto durante la inferencia de los datos de entrada, considerándose todos los datos capturados.

Capa *Conv2D*

La siguiente capa se corresponde con una nueva convolución ***Conv2D***, en este caso con el doble de filtros que la anteriormente implementada, como se indica en la sentencia de la Figura 98, lo cual es una práctica habitual al aproximarse a las últimas capas de la red neuronal. Al igual que la primera capa, ésta aprende a reconocer patrones en valores adyacentes que contengan información relevante. Las características que identifica son combinaciones de las ya distinguidas en la primera capa.

Figura 98. Extracto de código - Capa Conv2D (II)

```
tf.keras.layers.Conv2D(16, (4, 1), padding="same",
                       activation="relu"),
```

Capas *MaxPool2D* y *Dropout*

Tras esta segunda capa de convolución, se implementa una nueva capa de ***MaxPool2D*** y ***Dropout***, continuando así con el proceso de destilación de los datos de entrada originales en una representación menor y más manejable. Las salidas de éstas, con una forma de **(14,1,16)**, representan un tensor multidimensional que muestra, de manera simbólica, únicamente los extractos más significativos contenidos en los datos de entrada originales.
El número de capas que se implementan mediante procesos de convolución y *pooling* puede continuar, como se indica en el código representado en la Figura 99, siendo necesario ajustar su valor durante el desarrollo del modelo.

Figura 99. Extracto de código - Capas MaxPool2D y Dropout

```
tf.keras.layers.MaxPool2D((3, 1), padding="same"),
tf.keras.layers.Dropout(0.1),
```

Capas *Flatten* y *Dense*

Hasta este punto, se han estado procesando los datos de entrada a través de capas de convolución que, únicamente, atienden a relaciones entre valores adyacentes. Sin embargo, como se dispone de una serie de representaciones a alto nivel de las principales características contenidas en los datos de entrada, se puede ampliar la visión y estudiarlos de manera conjunta. Para ello se utiliza, en un primer lugar, la capa ***Flatten***, que transforma un tensor multidimensional en uno con una única dimensión. En este caso, el tensor de tamaño **(14,1,16)** se reduce a una única dimensión de tamaño **224**. La salida se introduce como entrada en una capa ***Dense*** de 16 neuronas, donde cada entrada se conecta a cada neurona, como se indica en el código mostrado en la Figura 100. Considerando todos los datos a la vez, esta capa puede aprender el significado de varias combinaciones de entradas, siendo ésta una de las herramientas más básicas del *toolbox* de *DeepLearning.*

Figura 100. Extracto de código - Capas Flatten y Dense

```
tf.keras.layers.Flatten(),
tf.keras.layers.Dense(16, activation="relu")
```

Capas *Dropout* y *Dense*

La tarea final consiste en comprimir estos 16 valores en 4 clases (símbolo 1, símbolo 2, símbolo 3 y *unknown).* Para ello, en primer lugar, se añade alguna capa más de ***Dropout*** y ***Dense***. La capa final genera 4 neuronas, las cuales están asociadas a una clase de gesto a identificar a partir de los datos de entrada; cada una de ellas está conectada a las 16 salidas de la capa anterior. Durante el entrenamiento, cada neurona aprende la combinación de activaciones de las capas previas que corresponden a cada uno de los gestos que representa.

La capa se configura como una función de activación `softmax`, como se muestra en el código de la Figura 101, lo que resulta en una salida formada por un conjunto de probabilidades que suman 1, constituyendo así el tensor de salida del modelo.

Figura 101. Extracto de código - Capas Dropout y Dense

```
tf.keras.layers.Dropout(0.1),
tf.keras.layers.Dense(4, activation="softmax")
```

- Función `train_net()`

Esta función realiza el entrenamiento del modelo, estableciendo parámetros como el número de *epochs* a 50, el *batch size* a 64 y los *steps per epochs* al valor de 1000, después de un proceso experimental, tal y como se observa en el pequeño extracto de código mostrado en la Figura 102.

Figura 102. Extracto de código - Función train_net

```
def train_net(
    model,
    model_path,  # pylint: disable=unused-argument
    train_len,  # pylint: disable=unused-argument
    train_data,
    valid_len,
    valid_data,  # pylint: disable=unused-argument
    test_len,
    test_data,
    kind):
  """Trains the model."""
  calculate_model_size(model)
  epochs = 50
  batch_size = 64
  model.compile(
      optimizer="adam",
      loss="sparse_categorical_crossentropy",
      metrics=["accuracy"])
  if kind == "CNN":
    train_data = train_data.map(reshape_function)
    test_data = test_data.map(reshape_function)
    valid_data = valid_data.map(reshape_function)
  test_labels = np.zeros(test_len)
  idx = 0
  for data, label in test_data:  # pylint: disable=unused-variable
    test_labels[idx] = label.numpy()
    idx += 1
  train_data = train_data.batch(batch_size).repeat()
  valid_data = valid_data.batch(batch_size)
  test_data = test_data.batch(batch_size)
    model.fit(
        train_data,
        epochs=epochs,
        validation_data=valid_data,
        steps_per_epoch=1000,
        validation_steps=int((valid_len - 1) / batch_size + 1),
        callbacks=[tensorboard_callback])
    loss, acc = model.evaluate(test_data)
    pred = np.argmax(model.predict(test_data), axis=1)
    confusion = tf.math.confusion_matrix(
        labels=tf.constant(test_labels),
        predictions=tf.constant(pred),
        num_classes=4)
    print(confusion)
    print("Loss {}, Accuracy {}".format(loss, acc))
```

4.4 Resultados de la generación del modelo TensorFlow inicial

Durante la fase de entrenamiento se pueden observar distintos resultados obtenidos a partir de la ejecución del *script* **train.py**. En primer lugar, se expone la longitud de los datos de entrenamiento, test y validación utilizados, en relación a los porcentajes expuestos previamente. El 60% del total del conjunto de datos va destinado al entrenamiento, y el otro 40% restante se divide en partes iguales entre validación y test, tal y como se muestra en la Figura 103.

Figura 103. Longitud de los datos de entrenamiento, test y validación

```
train_data_length:5550
valid_data_length:73
test_data_length:77
```

Por otro lado, en la Figura 104 se muestra la estructura del modelo secuencial que se ha generado.

Figura 104. Modelo secuencial generado

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 128, 3, 8)         104
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 42, 1, 8)          0
_________________________________________________________________
dropout (Dropout)            (None, 42, 1, 8)          0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 42, 1, 16)         528
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 1, 16)         0
_________________________________________________________________
dropout_1 (Dropout)          (None, 14, 1, 16)         0
_________________________________________________________________
flatten (Flatten)            (None, 224)               0
_________________________________________________________________
dense (Dense)                (None, 16)                3600
_________________________________________________________________
dropout_2 (Dropout)          (None, 16)                0
_________________________________________________________________
dense_1 (Dense)              (None, 4)                 68
=================================================================
Total params: 4,300
Trainable params: 4,300
Non-trainable params: 0
```

Así, el modelo generado consta de 10 capas, tal y como se describió anteriormente. Dos capas de convolución, que extraen las características básicas de los datos de entrada, dos

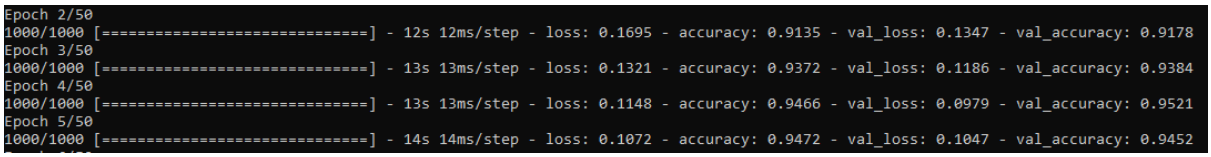
capas de *pooling*, cuya función es reducir la dimensión del tensor de entrada, tres capas de *dropout*, donde se lleva a cabo la regularización, una capa *flatten,* convirtiendo un tensor multidimensional en uno de una sola dimensión, y dos capas *dense*, donde cada entrada se conecta a cada neurona. Por último, se observa que el número total de parámetros es de 4300.

Además, el peso del modelo es de 16,8 KB aproximadamente, como se muestra en la Figura 105.

Figura 105. Tamaño del modelo

Al comenzar el entrenamiento se muestran los registros representados en la Figura 106.

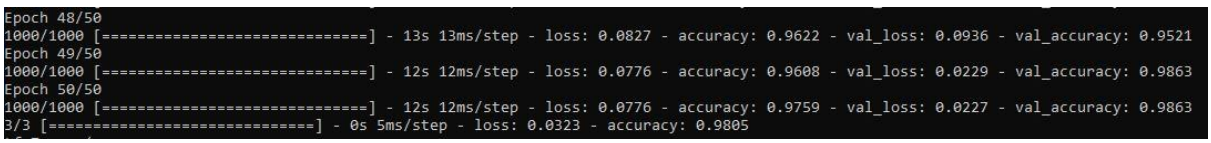
Figura 106. Métricas al inicio del entrenamiento



En ellos se observan las métricas del entrenamiento, en este caso al inicio del proceso. Se puede ver, por un lado, las pérdidas y precisión de los datos de entrenamiento, y por otro, las pérdidas y precisión de los datos de validación. Asimismo, se observa cómo a medida que avanza el entrenamiento, disminuyen las pérdidas y aumenta la precisión.

Si se analizan los últimos *epochs* representados en la Figura 107, es decir, cuando el entrenamiento está finalizando, se observa una diferencia sustancial de las métricas con respecto al inicio del proceso.

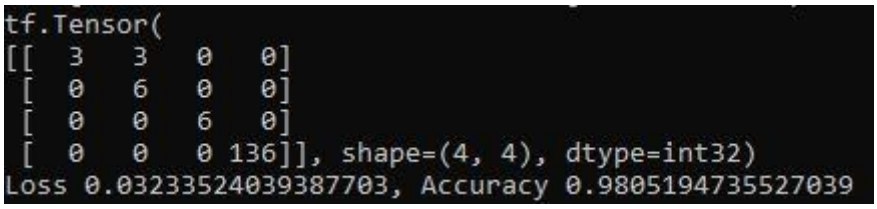
Figura 107. Métricas al finalizar el entrenamiento



Así, las pérdidas son prácticamente insignificantes, y, además, las de validación son, por lo general, más bajas que las de entrenamiento, indicando que no se está produciendo *overfitting.* Por otro lado, la precisión del modelo es muy cercana a 1 (100%), rondando el 98-99%, por lo tanto, se puede concluir que el modelo se ha generado correctamente y cumple con los requisitos necesarios.

En la Figura 108 se expone la matriz de confusión obtenida, que permite visualizar las prestaciones y precisión del algoritmo, representando cuán bien coincide la clase predicha de cada entrada en el conjunto de datos de test con su valor real.

Figura 108. Matriz de confusión

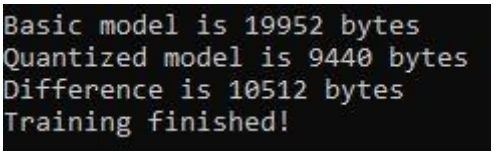


Cada columna de la matriz corresponde a un gesto predicho, en este caso triángulo, cuadrado, zigzag y *unknown*, mientras que cada fila corresponde al valor real del gesto (en el mismo orden). Asimismo, se puede observar que prácticamente todas las predicciones coinciden con su valor real. Sin embargo, se produce una pequeña confusión, donde algunas entradas correspondientes al gesto del triángulo han sido clasificadas como cuadrado.

Dicha matriz da una idea de dónde puede fallar el modelo, en este caso, sería beneficioso obtener más datos de entrenamiento del gesto correspondiente al triángulo para evitar la confusión, aunque es tan pequeña e insignificante que, en principio, se puede dar por válida. También se puede deducir que las pérdidas son mínimas y la precisión es muy cercana al 100%.

Por último, en la Figura 109 se expone el tamaño de los modelos *Tensor Flow* generados.

Figura 109. Tamaño de los modelos TF



En este caso, el modelo básico generado tiene un tamaño de 19,95 KB, mientras que el modelo optimizado (*quantized)* pesa 9,4 KB. En total existe una diferencia de 10,5 KB, puesto que el modelo cuantificado reduce en gran medida el tamaño descartando parte de la información que puede resultar útil, por ello, se tendrán que implementar en un dispositivo ambos modelos antes de elegir uno.

4.4.1 Conversión a un archivo de C

Para finalizar, el modelo TensorFlow Lite generado se convierte en un archivo fuente de C con el fin de poder implementarlo en la aplicación a desarrollar en el dispositivo Argon. A través de *cygwin64* se pude obtener la herramienta **xxd**, que es la encargada de convertir ambos modelos en archivos fuente de C para poder ser implementados en microcontroladores, mediante la sentencia mostrada en la Figura 110.

Figura 110. Cigwin64

```
C:\cygwin64\bin>xxd -i model_quantized.tflite>model_quantized.cc

C:\cygwin64\bin>xxd -i model.tflite>model.cc
```

Como se observa en la Figura 111, a partir de este proceso se obtienen los archivos *.tflite* correspondientes al modelo básico y optimizado, con lo que solo restaría implementarlos en la plataforma *hardware*.

Figura 111. Archivos .cc y .tflite

model.cc	11/03/2021 12:23	Archivo CC	121 KB
model.tflite	11/03/2021 11:45	Archivo TFLITE	20 KB
model_quantized.cc	11/03/2021 12:22	Archivo CC	57 KB
model_quantized.tflite	11/03/2021 11:45	Archivo TFLITE	10 KB

4.5 Implementación de la aplicación

El siguiente paso, una vez se dispone de un modelo *TF Lite* válido, es crear la aplicación que cumpla con los requisitos necesarios para poder implementarlo en la plataforma Argon de *Particle*. La aplicación desarrollada en este TFG consta de un *loop* que alimenta el modelo con valores de x,y,z proporcionados por el acelerómetro LSM9DS1, ejecutando las inferencias y utilizando el resultado final para encender el LED de un color u otro en función del gesto identificado.

4.5.1 Estructura de la aplicación

La Figura 112 muestra la estructura de la aplicación, compuesta por 6 partes fundamentales.

Main Loop
La aplicación se ejecuta en un *loop* continuo, realizándose múltiples inferencias por segundo.

Accelerometer handler
Este componente captura datos del acelerómetro LSM9DS1 y los escribe en el tensor de entrada del modelo utilizando un *buffer.*

TF Lite Interpreter
Ejecuta el modelo TensorFlow Lite.

Modelo
Contiene el modelo básico TF Lite, en este caso particular, incluido como un *array* de datos, utilizado por TF Lite *interpreter*.

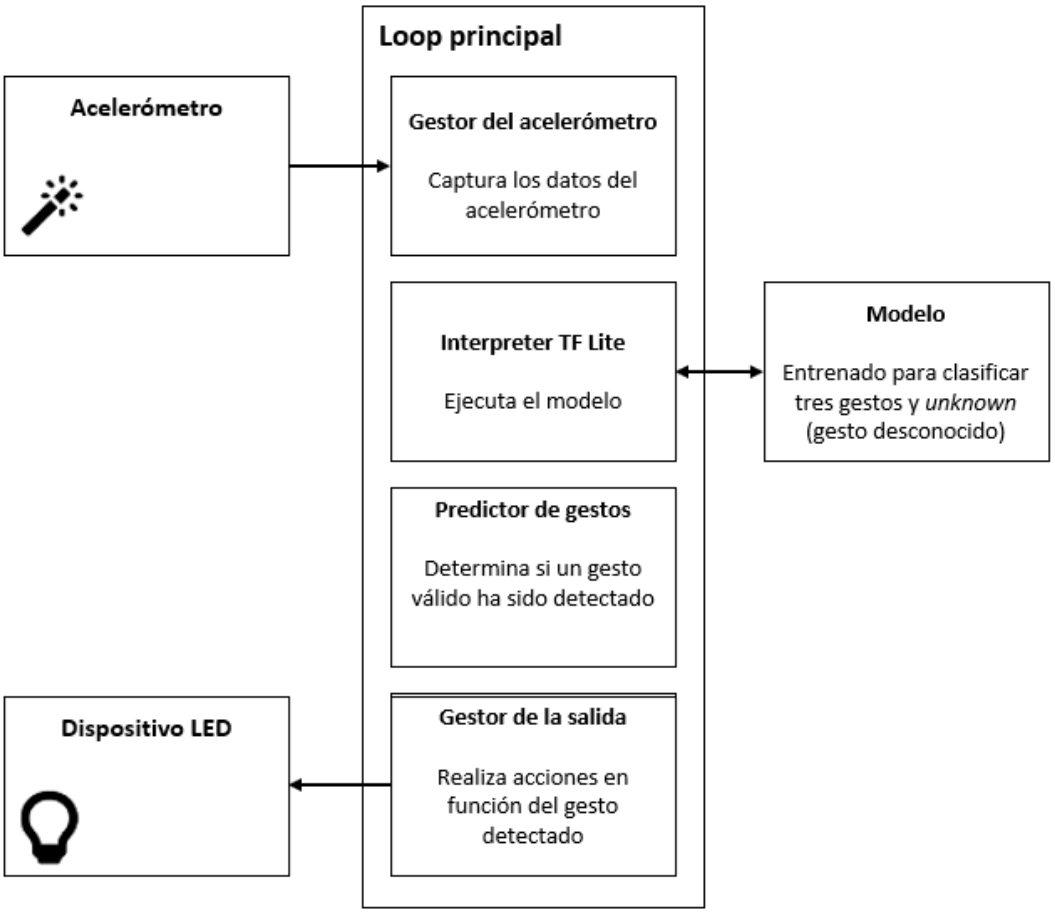
Gesture predictor

Este componente decide, en función de la salida obtenida, si alguno de los gestos definidos ha sido detectado, basándose en el valor umbral de la probabilidad y el número de predicciones positivas consecutivas.

Output handler

El gestor de la salida ilumina el LED RGB con el color asociado a cada uno de los gestos a identificar, e imprime dicha salida a través del puerto serie en función del gesto que ha sido reconocido.

Figura 112. Estructura de la aplicación



A partir de la estructura definida, la aplicación desarrollada en el presente TFG se compone de los siguientes ficheros.

constants.h, constants.cpp

Archivos que contienen las constantes que definen el comportamiento de la aplicación.

main_acc.cpp

Es la función *main* del programa que implementa la aplicación y se ejecuta cuando dicha aplicación se implementa en un dispositivo.

main_functions.h

Archivo que define la función `setup()`, encargada de llevar a cabo las inicializaciones necesarias, así como una función `loop()`, que contiene la lógica del núcleo del programa. La función `main` llama a estas funciones al inicio del programa.

gesture_predictor.h, gesture predictor.cpp

Archivos donde se determina si el gesto realizado coincide con alguno de los definidos o no.

output_handler.h, particle_output_handler.cpp

En estos archivos se define la función que se utiliza para representar la salida cada vez que se ejecute una inferencia.

accelerometer_handler.h, particle_accelerometer_handler.cpp

El código incluido en estos archivos gestiona los datos de entrada del acelerómetro.

acc_ model_data.h, acc_ model_data.cpp

Archivos donde se representa el modelo.

En los siguientes apartados se describe la implementación del código desarrollado en cada uno de los archivos asociados a la aplicación.

4.5.1.1 Fichero accelerometer_handler

Como se ha mencionado previamente, tiene la función de capturar datos del acelerómetro LSM9DS1 y escribirlos en el buffer de entrada del modelo.

Cabe recordar que la captura de datos se realizó a una frecuencia de 25 Hz, por lo que es necesario que se alimente el modelo a la misma frecuencia. Esto significa que, además de capturar datos, es necesario muestrear los datos para alimentar el modelo. Para ello se descartan parte de las muestras obtenidas, como se puede ver en el código que se expone a continuación.

En primer lugar, en el código desarrollado se incluyen los *headers*, como se muestra en la Figura 113.

Figura 113. Cabecera

```
#include "accelerometer_handler.h"

#include <Particle.h>
#include <Arduino_LSM9DS1.h>

#include "constants.h"
```

El archivo *particle.h* proporciona algunas definiciones básicas de la plataforma, mientras que el archivo *Arduino_LSM9DS1.h* es parte de la librería **Arduino_LSMD9DS1**, utilizada para la

comunicación con el acelerómetro LSM9DS1. Además, se configuran algunas variables, como se muestra en la Figura 114.

Figura 114. Configuración de variables

```
// A buffer holding the last 200 sets of 3-channel values
float save_data[600] = {0.0};
// Most recent position in the save_data buffer
int begin_index = 0;
// True if there is not yet enough data to run inference
bool pending_initial_data = true;
// How often we should save a measurement during downsampling
int sample_every_n;
// The number of measurements since we last saved one
int sample_skip_counter = 1;
```

Así, se incluye un *buffer* que se rellenará con los datos `save_data`, junto a diversas variables para rastrear la posición actual en el *buffer* y valorar si hay suficientes datos para comenzar a ejecutar inferencias. Las variables `sample_every_n` y `sample_skip_counter` se usan en el proceso de muestreo que se explicará posteriormente.

La función `SetupAccelerometer()`, cuyo código se muestra en la Figura 115, es llamada desde el *loop* principal de la aplicación con el fin de preparar el dispositivo para capturar datos del acelerómetro.

Figura 115. Extracto de código - Función SetUpAccelerometer

```
TfLiteStatus SetupAccelerometer(tflite::ErrorReporter *error_reporter)
{
  // Wait until we know the serial port is ready
  while (!Serial)
  {
  }

  // Switch on the IMU
  if (!IMU.begin())
  {
    error_reporter->Report("Failed to initialize IMU");
    return kTfLiteError;
  }
```

Se comprueba que el puerto serie del dispositivo está preparado y se inicializa la comunicación con el acelerómetro, concretamente con el IMU (*Inertial Measurement Unit)*.

A continuación, comienza el proceso de muestreo. En primer lugar, se accede a la librería IMU para determinar la frecuencia de muestreo del acelerómetro, y posteriormente se divide ese valor entre la frecuencia de muestreo que se desea, como se muestra en la Figura 116, definida en la constante `kTargetHz`, en el archivo **constants.h**.

Figura 116. Extracto de código - Comienzo del proceso de muestreo

```
// Determine how many measurements to keep in order to
// meet kTargetHz
float sample_rate = IMU.accelerationSampleRate();
sample_every_n = static_cast<int>(roundf(sample_rate / kTargetHz));
```

Teniendo en cuenta que la frecuencia de muestreo del acelerómetro es de 104 Hz, y la frecuencia deseada es de 25 Hz, el resultado de la división es 4.16, indicando cuántas muestras tomadas a una frecuencia de 104 Hz se han de seleccionar para conseguir la frecuencia de muestreo deseada de 25 Hz, que será de 1 cada 4.16.

Sin embargo, se hace complicado trabajar con números fraccionarios, por lo que se utiliza la función `roundf()` que redondea al número más cercano, en este caso 4. Así, para muestrear la señal original, se considerará 1 de cada 4 muestras, resultando una frecuencia de muestreo de 26 Hz, muy cercana a los 25Hz deseados. Finalmente, se almacena este valor en la variable `sample_every_n` para usarlo en el proceso de muestreo que alimenta el modelo.

Una vez establecidos los parámetros del proceso de muestreo, se informa al usuario de que la aplicación está lista, así como del valor obtenido previamente, y se retorna de la función `SetupAccelerometer()`, como se observa en el código de la Figura 117.

Figura 117. Extracto de código - Error Reporter

```
error_reporter->Report("Magic starts!");
error_reporter->Report("%d", sample_every_n);


return kTfLiteOk;
```

A continuación, se define la función `ReadAccelerometer()`, cuyo objetivo es capturar nuevos datos y escribirlos en el tensor de salida del modelo. Así, se comienza limpiando el *buffer* interno después de que un gesto haya sido reconocido, como se muestra en el código representado en la Figura 118.

Figura 118. Extracto de código - Función ReadAccelerometer

```
bool ReadAccelerometer(tflite::ErrorReporter *error_reporter, float *input,
                       int length, bool reset_buffer)
{
  // Clear the buffer if required, e.g. after a successful prediction
  if (reset_buffer)
  {
    memset(save_data, 0, 600 * sizeof(float));
    begin_index = 0;
    pending_initial_data = true;
  }
```

A continuación, se hace uso de las funciones definidas en la librería **Arduino_LSMD9DS1** para comprobar si hay datos disponibles en el *loop*, y en caso de que la respuesta sea afirmativa, se leen, tal y como se expone en el código de la Figura 119.

Figura 119. Extracto de código - Lectura

```
// Keep track of whether we stored any new data
bool new_data = false;
// Loop through new samples and add to buffer
while (IMU.accelerationAvailable())
{
  float x, y, z;
  // Read each sample, removing it from the device's FIFO buffer
  if (!IMU.readAcceleration(x, y, z))
  {
    error_reporter->Report("Failed to read data");
    break;
  }
```

El siguiente paso es implementar el algoritmo que realiza el proceso de muestreo de los datos proporcionados por el acelerómetro LSM9DS1, tal y como se muestra en la Figura 120.

Figura 120. Extracto de código - Algoritmo para el proceso de muestreo

```
// Throw away this sample unless it's the nth
if (sample_skip_counter != sample_every_n)
{
  sample_skip_counter += 1;
  continue;
}
```

Como se explicó previamente, el objetivo es considerar una cada “n” muestras, donde “n” se corresponde en este caso con el valor almacenado en la variable `sample_every_n`. Para ello, se emplea el contador `sample_skip_counter`, que cuenta el número de muestras que se han leído desde que se consideró la última. Si el número de la muestra que se lee no coincide con el valor de “n” el *loop* continua sin considerar estos datos y descartándolos.

Para almacenar los datos, se escriben en posiciones consecutivas en el buffer `save_data`, como se muestra en el código de la Figura 121.

Figura 121. Extracto de código - Almacenamiento de datos (buffer)

```
// Write samples to our buffer, converting to milli-Gs
// and flipping y and x order for compatibility with
// model (sensor orientation is different on Arduino
// Nano BLE Sense compared with SparkFun Edge)
save_data[begin_index++] = x * 1000;
save_data[begin_index++] = y * 1000;
save_data[begin_index++] = z * 1000;
```

El modelo toma medidas del acelerómetro LSM9DS1 en el orden x,y,z. Las últimas líneas del *loop* configuran algunas variables de estado que se usan en dicho *loop.*

Así, se inicializa el contador `sample_skip_counter`, se comprueba que no se sobrepasa el límite del tamaño del *buffer* de muestreo, y se configura la variable que indica que se han almacenado nuevos datos, como se observa en el código de la Figura 122.

Figura 122. Extracto de código - Inicio del contador

```
  // Since we took a sample, reset the skip counter
  sample_skip_counter = 1;
  // If we reached the end of the circle buffer, reset
  if (begin_index >= 600)
  {
    begin_index = 0;
  }
  new_data = true;
}
```

Además, se debe verificar que hay suficientes datos para ejecutar inferencias, y si no es así, se retorna de la función sin hacer nada, tal y como se expone en el código de la Figura 123.

Figura 123. Extracto de código - Comprobación de datos

```
// Skip this round if data is not ready yet
if (!new_data)
{
  return false;
}

// Check if we are ready for prediction or still pending more initial data
if (pending_initial_data && begin_index >= 200)
{
  pending_initial_data = false;
}

// Return if we don't have enough data
if (pending_initial_data)
{
  return false;
}
```

Al retornar `false` cuando no hay nuevos datos, se asegura que la función que se está llamando no interrumpa las inferencias que se puedan estar ejecutando.

Para finalizar, una vez se obtienen nuevos datos, se copia la cantidad apropiada, incluyendo nuevas muestras, en el tensor de entrada, como se muestra en el código de la Figura 124.

Figura 124. Extracto de código - Copia de datos en el tensor de entrada

```
// Copy the requested number of bytes to the provided input tensor
for (int i = 0; i < length; ++i)
{
  int ring_array_index = begin_index + i - length;
  if (ring_array_index < 0)
  {
    ring_array_index += 600;
  }
  input[i] = save_data[ring_array_index];
}

return true;
```

De esta manera ya está todo preparado para ejecutar inferencias, pasándose los resultados al fichero *gesture predictor*, que determinará si el gesto realizado es válido.

4.5.1.2 Fichero gesture_predictor

Cuando se produce una inferencia, el tensor de salida es rellenado con probabilidades que indican el gesto que se ha realizado, en caso de que se haya identificado alguno de los definidos inicialmente. Sin embargo, *Machine Learning* no es una ciencia exacta y puede fallar, de forma que una inferencia puede resultar en un falso positivo.

Para reducir el impacto de estos falsos positivos se ha establecido que para que un gesto se reconozca, se debe detectar en un número mínimo de inferencias consecutivas. Asimismo, se ejecutan múltiples inferencias por segundo, determinándose rápidamente si el resultado es válido. Esta es la funcionalidad que realiza el predictor de gestos o *gesture predictor.*

Éste define una única función, `PredictGesture()`, que toma como entrada el tensor de salida del modelo. Para determinar si se ha detectado un gesto, esta función realiza dos operaciones.

1. Comprueba que la probabilidad del gesto cumple con el mínimo valor del umbral establecido.
2. Comprueba si el gesto se ha detectado en un número determinado de inferencias consecutivas.

El número mínimo de inferencias requeridas varía según el gesto, ya que algunos pueden tomar tiempo para su ejecución. Igualmente, varía en función del dispositivo utilizado, ya que la velocidad de estos influye en el número de inferencias que se ejecutan por segundo. En este caso, se pueden observar los valores por defecto en el fichero **particle_constants.cpp**, representado en la Figura 125.

Figura 125. Extracto de código - Número mínimo de inferencias

```
const int kConsecutiveInferenceThresholds[3] = {3, 3, 3};
```

De manera experimental se ha ido variando el número de inferencias mínimas requeridas para reconocer los gestos definidos (triángulo, cuadrado y zigzag) y se ha concluido que los valores con los que se obtiene una mayor probabilidad de acierto son 3,3,3.

El código del predictor de gestos se expone a continuación. En primer lugar, se definen algunas variables que se usan para hacer un seguimiento del último gesto que se ha realizado, y cuántas veces se ha identificado la realización del gesto de forma consecutiva, como se muestra en la Figura 126.

Figura 126. Extracto de código - Definición de variables

```
// How many times the most recent gesture has been matched in a row
int continuous_count = 0;
// The result of the last prediction
int last_predict = -1;
```

El siguiente paso consiste en definir la función `PredictGesture()` y determinar si alguna de las categorías de los gestos definidos tiene una probabilidad mayor a 0.8 en la inferencia más reciente, como se observa en la Figura 127.

Figura 127. Extracto de código - Función PredictGesture

```
// Return the result of the last prediction
// 0: square("S"), 1: triangle("T"), 2: zigzag("Z"), 3: unknown
int PredictGesture(float* output) {
  // Find whichever output has a probability > 0.8 (they sum to 1)
  int this_predict = -1;
  for (int i = 0; i < 3; i++) {

    if (output[i] > 0.8) this_predict = i;
  }
```

Para almacenar el índice del gesto predicho, se usa la variable `this_predict`. La variable `continuous_count` se utiliza para comprobar cuántas veces consecutivas se ha predicho el gesto más reciente. Si ninguna de las categorías del gesto sobrepasa la probabilidad de 0.8 (80%), se inicializa cualquier proceso de detección configurando el valor de la variable `continuous_count` a 0 y `last_predict` al valor 3, que corresponde al índice de la categoría *unknown* (gesto no reconocido), como se expone en la Figura 128.

Figura 128. Extracto de código – Gesto no detectado

```
// No gesture was detected above the threshold
if (this_predict == -1) {
  continuous_count = 0;
  last_predict = 3;
  return 3;
}
```

Si el gesto más reciente coincide con el gesto previo, se incrementa el valor de la variable `continuous_count`, si no es así, ésta se inicializa al valor 0. Además, se almacena la predicción más reciente en la variable `last_predict`, como se muestra en la Figura 129.

Figura 129. Extracto de código - Variable last_predict

```
if (last_predict == this_predict) {
  continuous_count += 1;
} else {
  continuous_count = 0;
}
last_predict = this_predict;
```

En la siguiente sección, se utiliza la variable `continuous_count` para comprobar si el gesto actual se ha identificado el número de veces consecutivas establecido; si no es así se retorna el valor 3, indicando un gesto desconocido, tal y como se muestra en el código de la Figura 130.

Figura 130. Extracto de código - Variable continous_count

```
// If we haven't yet had enough consecutive matches for this gesture,
// report a negative result
if (continuous_count < kConsecutiveInferenceThresholds[this_predict]) {
  return 3;
}
```

Si se llega hasta este punto es porque se ha confirmado un gesto válido, tras ello, se inicializan todas las variables, mostradas en la Figura 131.

Figura 131. Extracto de código - Inicialización de las variables

```
// Otherwise, we've seen a positive result, so clear all our variables
// and report it
continuous_count = 0;
last_predict = -1;
return this_predict;
```

La función termina retornando la predicción actual, que se pasará al fichero *output handler*, imprimiendo el resultado por el terminal serie al usuario.

4.5.1.3 Fichero output_handler (Gestor de la salida)

La función principal es enviarle información a través del puerto serie en función del gesto que se ha detectado, y conmutar el LED de la placa Argon, conectado al pin D7, cada vez que se ejecuta una inferencia.

La primera vez que se ejecuta la función, el LED está configurado como salida, como se muestra en el código de la Figura 132.

Figura 132. Extracto de código - Función HandleOutput

```
void HandleOutput(tflite::ErrorReporter *error_reporter, int kind)
{
  // The first time this method runs, set up our LED
  static bool is_initialized = false;
  if (!is_initialized)
  {
    pinMode(D7, OUTPUT);
    is_initialized = true;
  }
```

A continuación, se enciende y se apaga con cada inferencia, como se muestra en la Figura 133.

Figura 133. Extracto de código - Función HandleOutput (II)

```
// Toggle the LED every time an inference is performed
static int count = 0;
++count;
if (count & 1)
{
  digitalWrite(D7, HIGH);
}
else
{
  digitalWrite(D7, LOW);
}
```

Por último, se imprime por pantalla el código ASCII correspondiente y se ilumina el LED RGB de la placa Argon con el color correspondiente, dependiendo del gesto que se haya reconocido, como se indica en el código mostrado en la Figura 134.

Figura 134. Extracto de código - Código ASCII para cada gesto

```
// Print some ASCII art for each gesture
if (kind == 0)
{
  RGB.control(true);
  RGB.color(0, 255, 0); // Green

  error_reporter->Report(
      "*** TRIANGLE\n\r");
}
else if (kind == 1)
{
  RGB.control(true);
  RGB.color(0, 0, 255); // Blue

  error_reporter->Report(
      "*** SQUARE\n\r");
}
else if (kind == 2)
{
  RGB.control(true);
  RGB.color(255, 0, 0); // Red

  error_reporter->Report(
      "*** ZIGZAG\n\r");
}
/  else if (kind == 3)
/  {
/error_reporter->Report(
/        "--UNKNOWN\n\r");
/  }
else
{
  RGB.control(false);
```

Así, en caso de que el gesto detectado se corresponda con el gesto de índice 0, es decir, el triángulo, el LED RGB de la placa Argon se ilumina de color verde. Si se detecta un cuadrado (índice = 1), el LED se ilumina de color azul; si el gesto detectado es el zigzag (índice = 2), el color del LED es rojo.

4.5.1.4 Función principal (main)

Estos tres componentes de los que se ha hablado previamente, se unen en la función **main_acc.cpp**, que es la función principal de la aplicación (*main).*

En primer lugar, se configuran diversas variables, así como algunos extras mostrados en el código de la Figura 135.

Figura 135. Extracto de código - Configuración de variables

```
// Globals, used for compatibility with Arduino-style sketches.
namespace
{
tflite::ErrorReporter *error_reporter = nullptr;
const tflite::Model *model = nullptr;
tflite::MicroInterpreter *interpreter = nullptr;
TfLiteTensor *model_input = nullptr;
int input_length;

// Create an area of memory to use for input, output, and intermediate arrays.
// The size of this will depend on the model you're using, and may need to be
// determined by experimentation.
// If you're getting Red SOS Heap fragmentation errors on Argon or Boron devices,
// reduce the size of the tensor arena product from 60 to 50 or 40.
constexpr int kTensorArenaSize = 60 * 1024;
uint8_t tensor_arena[kTensorArenaSize];

// Whether we should clear the buffer next time we fetch data
bool should_clear_buffer = false;
} // namespace
```

La variable `input_length` almacena la longitud del tensor de entrada del modelo, mientras que la variable `should_clear_buffer` es un *flag* que indica si el *buffer* del acelerómetro debe ser limpiado en la próxima ejecución, lo cual se realiza después de reconocer adecuadamente un gesto.

La función `setup()` realiza todos los preparativos necesarios para poder ejecutar nuevas inferencias, a partir del código mostrado en la Figura 136 y en la Figura 137.

Figura 136. Extracto de código - Función SetUp

```
// The name of this function is important for Arduino compatibility.
void setup()
{
  // Set up logging. Google style is to avoid globals or statics because of
  // lifetime uncertainty, but since this has a trivial destructor it's okay.
  static tflite::MicroErrorReporter micro_error_reporter; // NOLINT
  error_reporter = &micro_error_reporter;

  // Map the model into a usable data structure. This doesn't involve any
  // copying or parsing, it's a very lightweight operation.
  model = tflite::GetModel(g_magic_wand_model_data);
  if (model->version() != TFLITE_SCHEMA_VERSION)
  {
    error_reporter->Report(
        "Model provided is schema version %d not equal "
        "to supported version %d.",
        model->version(), TFLITE_SCHEMA_VERSION);
    return;
  }

  // Pull in only the operation implementations we need.
  // This relies on a complete list of all the ops needed by this graph.
  // An easier approach is to just use the AllOpsResolver, but this will
  // incur some penalty in code space for op implementations that are not
  // needed by this graph.
  static tflite::MicroMutableOpResolver micro_mutable_op_resolver; // NOLINT
  micro_mutable_op_resolver.AddBuiltin(
      tflite::BuiltinOperator_DEPTHWISE_CONV_2D,
      tflite::ops::micro::Register_DEPTHWISE_CONV_2D());
micro_mutable_op_resolver.AddBuiltin(
    tflite::BuiltinOperator_MAX_POOL_2D,
    tflite::ops::micro::Register_MAX_POOL_2D());
micro_mutable_op_resolver.AddBuiltin(tflite::BuiltinOperator_CONV_2D,
                                     tflite::ops::micro::Register_CONV_2D());
micro_mutable_op_resolver.AddBuiltin(
    tflite::BuiltinOperator_FULLY_CONNECTED,
    tflite::ops::micro::Register_FULLY_CONNECTED());
micro_mutable_op_resolver.AddBuiltin(tflite::BuiltinOperator_SOFTMAX,
                                     tflite::ops::micro::Register_SOFTMAX());



micro_mutable_op_resolver.AddBuiltin(tflite::BuiltinOperator_RESHAPE,
                                     tflite::ops::micro::Register_RESHAPE());



// Build an interpreter to run the model with
static tflite::MicroInterpreter static_interpreter(
    model, micro_mutable_op_resolver, tensor_arena, kTensorArenaSize,
    error_reporter);
interpreter = &static_interpreter;

// Allocate memory from the tensor_arena for the model's tensors
interpreter->AllocateTensors();
```

Figura 137. Extracto de código - Función SetUp (II)

```
// Obtain pointer to the model's input tensor
model_input = interpreter->input(0);
if ((model_input->dims->size != 4) || (model_input->dims->data[0] != 1) ||
    (model_input->dims->data[1] != 128) ||
    (model_input->dims->data[2] != kChannelNumber) ||
    (model_input->type != kTfLiteFloat32))
{
  error_reporter->Report("Bad input tensor parameters in model");
  return;
}

input_length = model_input->bytes / sizeof(float);

TfLiteStatus setup_status = SetupAccelerometer(error_reporter);
if (setup_status != kTfLiteOk)
{
  error_reporter->Report("Set up failed\n");
}
```

Por otra parte, en la Figura 138 y en la Figura 139 se muestra el código correspondiente a la función `loop()`.

Figura 138. Extracto de código - Función loop

```
void loop()
{
  // Attempt to read new data from the accelerometer
  bool got_data = ReadAccelerometer(error_reporter, model_input->data.f,
                                    input_length, should_clear_buffer);

  //Serial.println("Data: %d", got_data);

  // Don't try to clear the buffer again
  should_clear_buffer = false;
  // If there was no new data, wait until next time
  if (!got_data)
    return;
  // Run inference, and report any error
  TfLiteStatus invoke_status = interpreter->Invoke();
  if (invoke_status != kTfLiteOk)
  {
    error_reporter->Report("Invoke failed on index: %d\n", begin_index);
    return;
  }
```

Figura 139. Extracto de código - Función loop (II)

```
// Analyze the results to obtain a prediction


  int j0 = (int)(100 * interpreter->output(0)->data.f[0]);
  int j1 = (int)(100 * interpreter->output(0)->data.f[1]);
  int j2 = (int)(100 * interpreter->output(0)->data.f[2]);

  error_reporter->Report("T: %d%%    S: %d%%    Z: %d%%", j0, j1, j2);


int gesture_index = PredictGesture(interpreter->output(0)->data.f);
// Clear the buffer next time we read data
should_clear_buffer = gesture_index < 3;
// Produce an output
HandleOutput(error_reporter, gesture_index);
```

En primer lugar, se leen los valores del acelerómetro LSM9DS1 para, posteriormente, ajustar el valor de la variable `should_clear_buffer` a *false*, asegurando así que no se siga intentando limpiar dicho buffer más de una vez.

Si no se obtienen nuevos datos, la función `ReadAccelerometer()` retorna *false* y se retornará del *loop,* intentándolo de nuevo la próxima vez que se llame a la función.

Si el valor que retorna `ReadAccelerometer()` es *true*, se ejecuta la inferencia en el tensor de entrada y se pasa el resultado a la función `PredictGesture()`, que proporciona el índice del gesto que ha sido detectado. Si dicho índice es menor a 3, el gesto es válido, por lo que se reajusta el valor del *flag* `should_clear_buffer` para limpiar el *buffer* la próxima vez que se llame a la función `ReadAccelerometer()`. Por último, se llama a la función `HandleOutput()` para reportar los resultados al usuario.

Además, en el archivo **particle_main.cpp** se llama automáticamente a las funciones `setup()` y `loop()`, por lo que no es necesario utilizar su propia rutina para ello, de esta manera se consigue ahorrar en recursos.

4.5.2 Validación experimental

El último paso consistió en validar los resultados obtenidos a partir de la ejecución de la aplicación desarrollada en la placa Argon integrando el acelerómetro LSM9DS1, una vez compilado el código y cargado en el dispositivo, y así comprobar el correcto funcionamiento del modelo, satisfaciendo las necesidades iniciales y cumpliendo los objetivos marcados.

Los gestos a identificar son los expuestos en la Figura 140, manteniendo en la mano en el dispositivo Argon como se muestra en la Figura 141. Para una correcta ejecución de cada uno de ellos, se parte del punto inicial y se sigue el orden numérico hasta retornar al punto de inicio.

Figura 140. Gestos a realizar

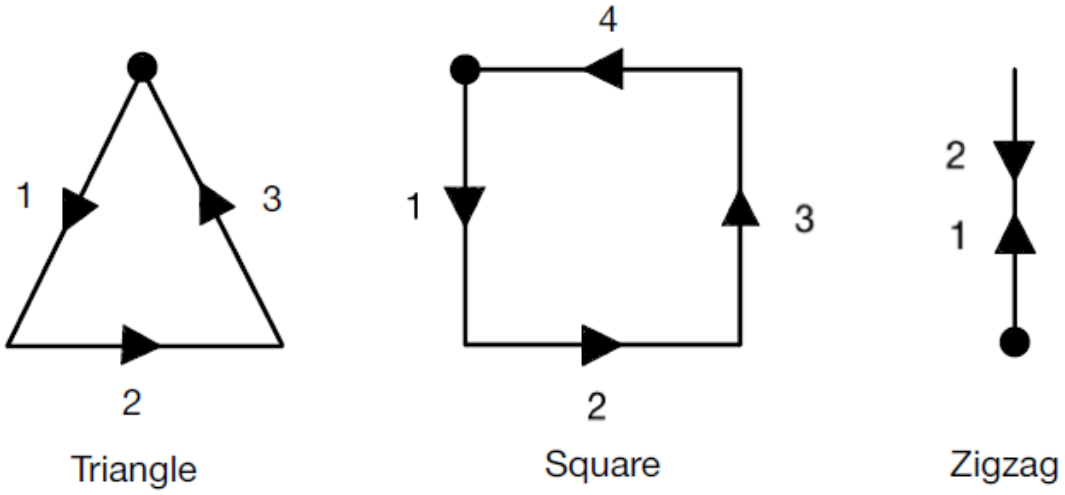
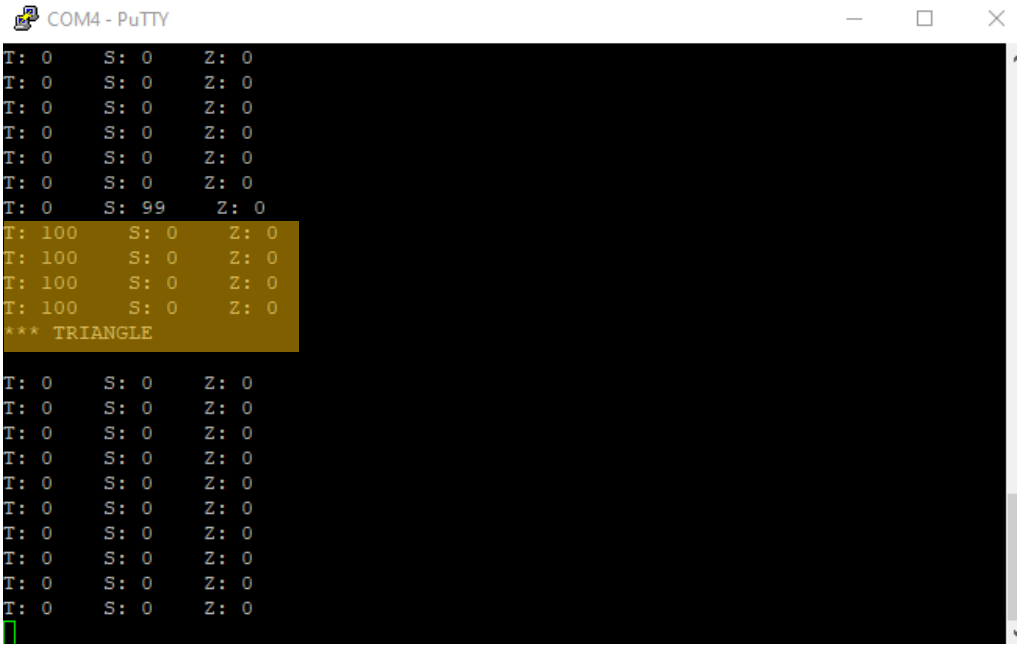


Figura 141. Orientación en mano del dispositivo Argon

Tras ejecutar la aplicación en la placa Argon, se obtienen a través del terminal serie los resultados mostrados en la Figura 142 para uno de los casos validados.

Figura 142. Lectura del terminal puerto-serie



En este caso particular, se reconoce el gesto correspondiente al triángulo. Como se puede observar, se ha detectado después de identificarse el número de veces establecido (4) a partir de las inferencias identificadas por el modelo, cumpliéndose así con el valor umbral estipulado en la constante `kConsecutiveInferenceThresholds` de la aplicación desarrollada, mostrada en la Figura 143.

Figura 143. Extracto de código - Número mínimo de inferencias consecutivas

```
const int kConsecutiveInferenceThresholds[3] = {3, 3, 3};
```

Dicho valor se ha configurado previamente y es 3, es decir, con el contador empezando a 0, se deben identificar 4 inferencias consecutivas asociadas al gesto en el modelo, como ocurre en este caso, para ser válidamente identificado.

Además, como se ha comentado previamente, el LED RGB de la placa Argon se debe iluminar de color verde, indicando que se trata del primer gesto con índice 0, el triángulo.

Con esto se valida experimentalmente la aplicación desarrollada inicialmente en el presente TFG y el correcto funcionamiento del modelo generado, concluyendo con éxito la creación y desarrollo de un modelo capaz de detectar tres gestos utilizando un acelerómetro integrado en la placa Argon, así como una aplicación capaz de leer, interpretar y representar los resultados obtenidos.

CAPÍTULO 5. Desarrollo de un modelo Tensor Flow para el reconocimiento de letras del alfabeto dactilológico

A partir del desarrollo presentado en el capítulo anterior, se ha desarrollado un modelo capaz de reconocer letras del alfabeto dactilológico de la lengua de signos española. Para ello, se ha decidido integrar una serie de resistencias *flex* junto con el acelerómetro LSM9DS1 en el dispositivo Argon incluyéndolos en un guante para, posteriormente, generar un modelo TF a partir del cual reconocer diferentes símbolos asociados al alfabeto dactilológico de la lengua de signos española [22].

En la Figura 144 se muestra la integración en un guante de los cinco sensores *flex* correspondientes a cada uno de los dedos de la mano, tomando como referencia el circuito expuesto en capítulos anteriores en la Figura 6.

Figura 144. Integración en el guante de los sensores flex

En la Figura 145 se expone el guante terminado con todas las conexiones realizadas y totalmente preparado para ser utilizado.

Figura 145. Guante terminado

5.1 Captura y generación de datos

Después de analizar los distintos gestos que se podían identificar, así como el campo de aplicación del modelo, se ha decidido desarrollar un modelo capaz de reconocer símbolos asociados a letras del alfabeto dactilológico de la lengua de signos española, concretamente las letras D, E y F que se muestran en la Figura 146.

Figura 146. Letras D, E y F del alfabeto dactilológico de la lengua de signos española [23]

Para obtener los datos de entrenamiento se vuelve a desarrollar un programa capaz de muestrear los datos, ya no solo del acelerómetro sino también de las resistencias *flex*, y mostrar los valores de los datos muestreados a través del puerto serie mientras se realizan los gestos. Por ello, se han realizado ciertas modificaciones sobre los ficheros inicialmente desarrollados para incluir los valores de las muestras asociadas a dichas resistencias que aportarán nuevos datos con los que generar el modelo TF. Con este propósito se ha creado el *sketch* ***gesture_xyzflex.ino*** que se explica a continuación.

5.1.1 *Sketch* gesture_xyzflex.ino

En primer lugar, se incluye la librería del acelerómetro **Arduino_LSM9DS1.h** y la propia de Particle **Particle.h** mediante las sentencias `#include`. Además, en esta ocasión se añade también la librería **math.h**, que ofrece funciones matemáticas para implementar operaciones trigonométricas y exponenciales con números en punto flotante, tal y como se puede observar en el extracto de código expuesto en la Figura 147.

Figura 147. Extracto de código - Sentencias #include

```
// This #include statement was automatically added by the Particle IDE.
#include <Arduino_LSM9DS1.h>
#include <Particle.h>
#include <math.h>
```

A continuación, se definen las constantes a utilizar. La primera indica el número de veces que se debe realizar el gesto al presionar el botón **SetUp** de la placa Argon, la segunda determina el periodo de muestreo. Además, se identifica con un nombre específico el pin asociado al LED de la placa, como se expone en la Figura 148.

Figura 148. Extracto de código - Definición de constantes

```
#define NUMBER_VALUES 10 // number of mean values per button press
#define SAMPLE_PERIOD  80    // sample period (ms) = 40ms = 1/25Hz

#define BUTTONPIN D7
```

En esta ocasión se añaden también el número de sensores adicionales a integrar, que corresponde con el número de resistencias *flex*, así como la relación de los pines analógicos conectados a cada una de las resistencias, como se expone en el código de la Figura 149.

Figura 149. Extracto de código - Definición de sensores y relación de pines

```
#define NUMBER_SENSORS 5  // number of sensors = VEC_DIM
#define FLEX_PIN4 A4 // Pin connected to voltage divider output sensor 0
#define FLEX_PIN3 A3 // Pin connected to voltage divider output sensor 0
#define FLEX_PIN2 A2 // Pin connected to voltage divider output sensor 0
#define FLEX_PIN1 A1 // Pin connected to voltage divider output sensor 0
#define FLEX_PIN0 A0 // Pin connected to voltage divider output sensor 0
```

Además, han de modificarse los valores de las constantes VCC y R_DIV en función del valor de la tensión proporcionada por el dispositivo Argon. Asimismo, los valores `STRAIGHT_RESISTANCE` y `BEND_RESISTANCE` se obtienen midiendo con un polímetro el valor real de la resistencia que presenta cada sensor *flex* completamente estirado y flexionado, respectivamente, tal y como se expone en la Figura 150.

Figura 150. Extracto de código - Valores de las constantes

```
const float VCC = 3.28; // Measured voltage of vcc line
const float R_DIV = 51000.0; // Measured resistance
const float STRAIGHT_RESISTANCE0 = 10000.0;
const float BEND_RESISTANCE0 = 60000.0; // resistance at 90 deg
const float STRAIGHT_RESISTANCE1 = 50000.0;
const float BEND_RESISTANCE1 = 160000.0; // resistance at 90 deg
const float STRAIGHT_RESISTANCE2 = 30000.0;
const float BEND_RESISTANCE2 = 180000.0; // resistance at 90 deg
const float STRAIGHT_RESISTANCE3 = 10000.0;
const float BEND_RESISTANCE3 = 100000.0; // resistance at 90 deg
const float STRAIGHT_RESISTANCE4 = 30000.0;
const float BEND_RESISTANCE4 = 260000.0; // resistance at 90 deg
```

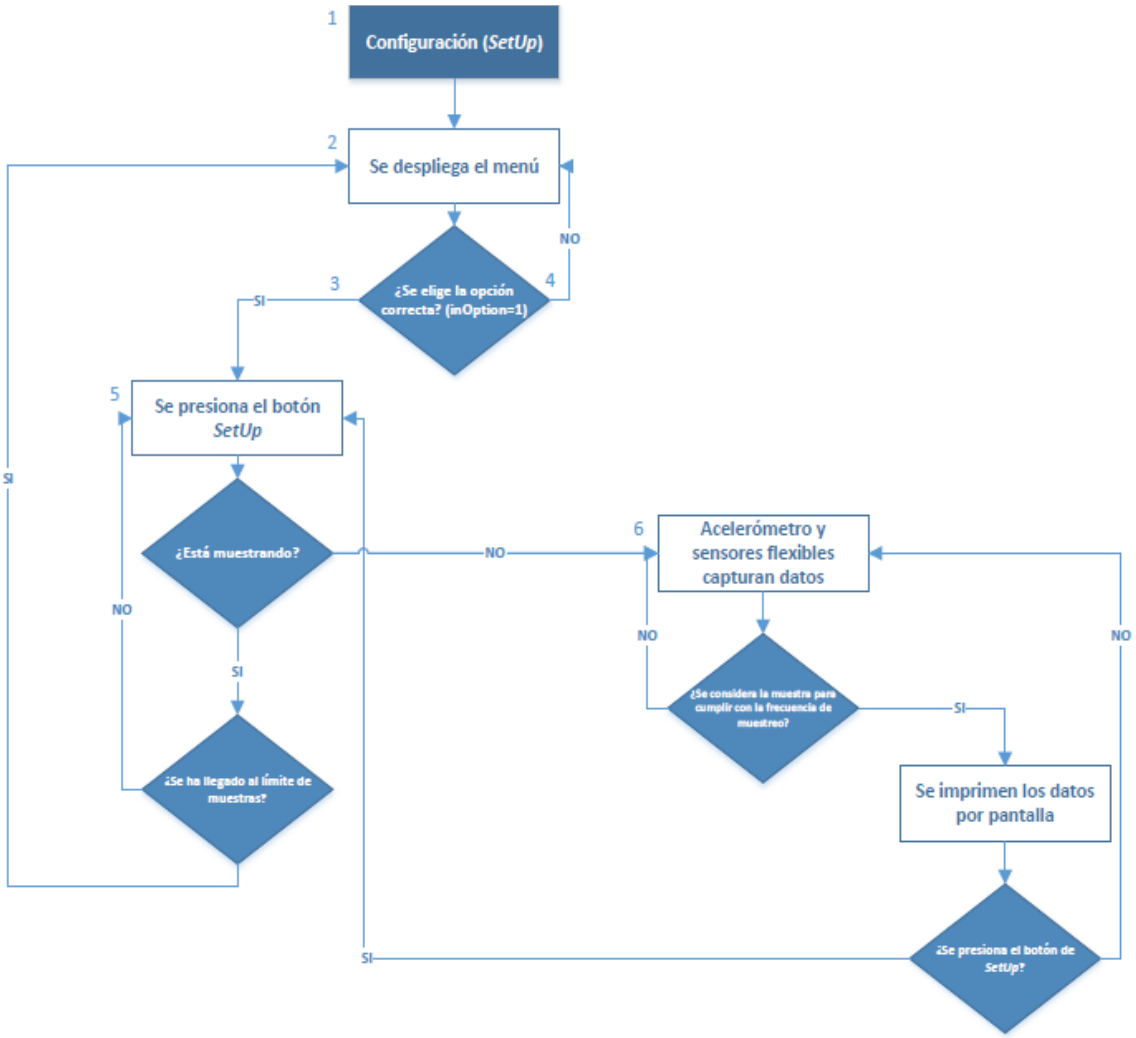
Por último, se definen las variables que se utilizarán a lo largo del código con diferentes finalidades, como se observa en el código expuesto en la Figura 151.

Figura 151. Extracto de código - Definición de variables

```
bool buttonAlreadyPressed;
char symbol;
bool enableSampling;
bool enableCaptureSampling;
bool enablePrediction;
bool enablePredictGesture;
int n_values;
bool isSampling;
int available;
int sample_every_n;
int sample_skip_counter;

unsigned long actual_time = 0;
unsigned long initial_time = 0;
```

El proceso implementado es prácticamente idéntico al desarrollado en el *sketch* presentado en el capítulo anterior, tal y como se deduce a partir del flujograma representado en la Figura 152. En este caso, se ha tenido en cuenta que no solo es necesario capturar datos del acelerómetro, sino también las muestras correspondientes a los diferentes sensores flexibles integrados en la plataforma.

Figura 152. Flujograma del sketch



1. **Configuración (SetUp)**

En primer lugar, se inicializan todas las variables definidas a sus valores iniciales, tal y como se observa en el extracto de código expuesto en la Figura 153.

Figura 153. Extracto de código - Inicialización de variables

```
void setup()
{
  Serial.begin(9600);

  enableSampling = false;
  buttonAlreadyPressed = false;
  enableCaptureSampling = false;
  n_values = 0;
  isSampling = false;
  available = 0;
  sample_skip_counter = 1;

  if (!IMU.begin()) {
    Serial.println("Failed to initialize IMU!");
    while (1);
  }
```

Además, se requiere que la frecuencia de muestreo se mantenga en 25 Hz, para lo cual, la variable `sample_rate` será igual a la frecuencia de muestreo del acelerómetro, y ésta a su vez se dividirá entre el valor de 25 Hz, determinándose así que, para satisfacer la frecuencia de muestreo establecida, debería considerarse una muestra cada `sample_every_n`, tal y como se expone en la Figura 154.

Figura 154. Extracto de código - Frecuencia de muestreo

```
float sample_rate = IMU.accelerationSampleRate();
sample_every_n = static_cast<int>(roundf(sample_rate / 25.0));

Serial.print("Accelerometer sample rate = ");
Serial.print(sample_rate);
Serial.print(" Hz (");
Serial.print(sample_every_n);
Serial.println(")");

Serial.println("Acceleration in mG's");
Serial.println("X\tY\tZ");
```

2. **Se despliega el menú**

A continuación, como se muestra en la Figura 155, se desplegará el menú para elegir la opción de captura de datos.

Figura 155. Extracto de código - Menú

```
Serial.println(" ");
Serial.println("MENU:");
Serial.println("-------------------------------------------------");
Serial.println("1. Capture Hand Gesture Data from LSM9DS1 & FlexSensors @ 25Hz ");
Serial.println("-------------------------------------------------");
Serial.print ("Enter the number corresponding to the menu option to perform: ");
```

Además, se configura el botón **SetUp** del dispositivo Argon, como se muestra en el extracto de código de la Figura 156, de manera que, cuando éste sea pulsado, se invocará la función `button_clicked` que se describe más adelante.

Figura 156. Extracto de código - Botón SetUp

```
System.on(button_click, button_clicked);
```

3. Se elige la opción correcta (inOption = 1)

Si se elige la opción 1 en el menú, es decir, la captura de datos, la variable `enableSampling`, aquella que habilita el proceso de muestreo, se establecerá a nivel alto, y las variables `n_values` (número de capturas por gesto) y `sample_skip_counter` (contador que determina las muestras que no se consideran para cumplir con la frecuencia de muestreo) tomarán los valores por defecto, como se expone en la Figura 157.

Figura 157. Extracto de código - Loop

```
void loop()
{
  if ((Serial.available() > 0) && (!enableSampling)) {
     char inOption = Serial.read();

     Serial.println(inOption);

     if (inOption == '1') {
        enableSampling = true;
        n_values = 0;
        sample_skip_counter = 1;
```

A continuación, se introduce la letra o símbolo que identifica al gesto que se desea realizar, en este caso las letras D, E y F del alfabeto dactilológico de la lengua de signos española, y se mantiene a la espera de poder establecer la comunicación con el puerto serie, como se puede observar en la Figura 158.

Figura 158. Extracto de código - Introducción de la letra a realizar

```
Serial.print ("Enter the letter/symbol for the hand gesture data (D: Symbol D, E: Symbol E, F: Symbol F): ");
while (!(Serial.available() > 0)) Particle.process(); // wait for serial port
char inID = Serial.read();
Serial.println(inID);
symbol = inID; // the symbol associated to the sampled values
```

4. **Se elige la opción incorrecta (inOption ≠ 1)**

Se muestra un mensaje en el que se advierte que no es una opción válida, desplegándose posteriormente el menú para elegir una nueva opción, como se observa en la Figura 159.

Figura 159. Extracto de código - Opción incorrecta

```
    }
    else {
        Serial.println("Not a valid option");
        Serial.println("");
        Serial.print ("Enter the number corresponding to the menu option to perform: ");
    }
}
```

5. **Se presiona el botón *SetUp***

- **Si no se está muestreando (*isSampling=False*)**

Se habilitará la captura de datos y el muestreo (`enableCaptureSampling = true`). Si la variable `isSampling` (aquella que indica si se está muestreando o no) se encuentra a nivel bajo, ésta se activará y se iniciará el contador (`sample_skip_counter`) a su valor inicial. Una vez la variable `isSampling` esté activada, se empezará a capturar datos del acelerómetro y de los diferentes sensores *flex*, como se muestra en la Figura 160.

Figura 160. Extracto de código - Captura de datos y muestreo habilitados

```
////////////////////////////////////////
// 1. Capture Hand Gesture Data @ 25Hz
////////////////////////////////////////
 if (enableSampling)
 {
   if (enableCaptureSampling)
   {
     if (!isSampling)
     {
       isSampling = true;
       Serial.println("-,-,-");
       sample_skip_counter = 1;
     }
```

- **Si está muestreando (*isSampling=True)***

En este caso, se detiene la operación de muestreo y se presenta por pantalla el número de muestras que se han obtenido al realizar una vez el gesto (variable `available`*)*. Ésta se inicializa al valor 0 y se incrementa el valor de `n_values` (número de veces que realiza una persona el gesto).

Si se ha llegado al límite de `n_values`, establecido en la constante `NUMBER_VALUES`, se inicializan las demás variables y vuelve a presentarse el menú inicial. En cambio, si aún no se ha superado el límite, se deberá pulsar de nuevo el botón **SetUp** en el dispositivo Argon para seguir capturando datos del acelerómetro y de los sensores *flex* en la realización del gesto que corresponda, como se muestra en la Figura 161.

Figura 161. Extracto de código - Límite de n_values

```
else  {
   if (isSampling) {

      sampleTimer.end();
     Serial.println(available);
     Serial.println("");
     Serial.printlnf("%lu", millis()-initial_time);

     available = 0;
     n_values++;
     isSampling = false;
     sample_skip_counter = 1;

     if (n_values == NUMBER_VALUES) // NÚMERO DE MUESTRAS QUE SE DESEAN TOMAR POR CADA PERSONA
     {
       Serial.println(symbol); // print the symbol

       // stop recording
       enableSampling = false;
       Serial.println("");
       Serial.println("MENU:");
       Serial.println("------------------------------------------------");
       Serial.println("1. Capture Hand Gesture Data from LSM9DS1 & FlexSensors @ 25Hz");
       Serial.println("------------------------------------------------");
       Serial.print ("Enter the number corresponding to the menu option to perform: ");
```

6. Acelerómetro y sensores *flex* capturan datos

- **Si no se considera la muestra obtenida para cumplir con la frecuencia de muestreo**

Esta situación se produce cuando el valor del contador `sample_skip_counter` es distinto al valor de `sample_every_n` (variable que determina cada cuántas muestras se considera una). El contador `sample_skip_counter` se imprimirá por pantalla y su valor se incrementará en uno, como se observa en la Figura 162.

Figura 162. Extracto de código - No se considera la muestra obtenida

```
float x, y, z;
if (IMU.accelerationAvailable()) {
  if (IMU.readAcceleration(x, y, z)) {

    if (sample_skip_counter != sample_every_n) {
      Serial.println(sample_skip_counter);
      sample_skip_counter += 1;
    }
```

- **Si se considera la muestra obtenida para cumplir con la frecuencia de muestreo**

Si el valor del contador `sample_skip_counter` es igual al de la variable `sample_every_n`, en primer lugar, se lee el valor proporcionado en ese momento por el conversor Analógico-Digital (ADC) integrado en el pin de la placa correspondiente, para calcular el valor del voltaje y de la resistencia medida en cada uno de los sensores *flex*, tal y como se expone en el extracto de código de la Figura 163.

Figura 163. Extracto de código - sample_skip_counter = sample_every_n

```
// Read the ADC, and calculate voltage and resistance from it
int flexADC0 = analogRead(FLEX_PIN0);
float flexV0 = flexADC0 * VCC / 4095.0;
float flexR0 = R_DIV*(VCC/flexV0-1.0);

int flexADC1 = analogRead(FLEX_PIN1);
float flexV1 = flexADC1 * VCC / 4095.0;
float flexR1 = R_DIV*(VCC/flexV1-1.0);

int flexADC2 = analogRead(FLEX_PIN2);
float flexV2 = flexADC2 * VCC / 4095.0;
float flexR2 = R_DIV*(VCC/flexV2-1.0);

int flexADC3 = analogRead(FLEX_PIN3);
float flexV3 = flexADC3 * VCC / 4095.0;
float flexR3 = R_DIV*(VCC/flexV3-1.0);

int flexADC4 = analogRead(FLEX_PIN4);
float flexV4 = flexADC4 * VCC / 4095.0;
float flexR4 = R_DIV*(VCC/flexV4-1.0);
```

Se ha de tener en cuenta que el conversor Analógico-Digital asociado a la función `analogRead()` presenta una resolución de 12 bits, por lo que las tensiones de entrada comprendidas entre 0 y 3.3 V se mapean en valores enteros en el rango comprendido entre 0 y 4095. En consecuencia, para obtener la tensión en voltios correspondiente a la variable `flexV`, es necesario multiplicar el valor `flexADC` (salida de la función `analogRead()`) por `VCC` y dividirlo entre el valor 4095.

Tomando como ejemplo las tres primeras líneas correspondientes al sensor *flex* 0, en la primera de ellas se almacena el valor entero obtenido a partir de la tensión existente en el pin *A0*, que corresponde a la tensión del divisor de tensión asociado. Posteriormente, a partir de este valor se obtiene la tensión en voltios que permitirá calcular el valor de la resistencia equivalente a la flexión del sensor en la última línea de código.

A continuación, una vez calculado el valor de la resistencia, se estima el ángulo de doblez del sensor flexible. Para ello se realiza un mapeo con dicho valor, tomando como referencia los valores de la resistencia completamente estirada y doblada, que dará resultado en rango de

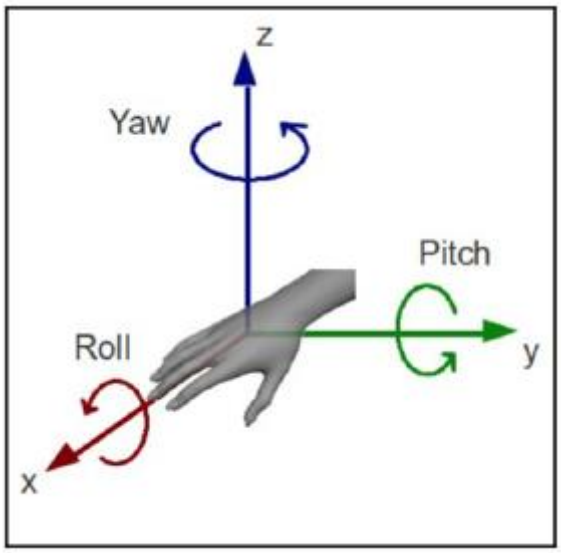
0 a 90, haciendo referencia a los grados del ángulo de doblez mencionado para, posteriormente, mostrarlos a través del terminal serie, como se expone en la Figura 164.

Figura 164. Extracto de código - Estimación de los ángulos de doblez de los sensores

```
// Use the calculated resistance to estimate the sensor's bend angle:
float value[NUMBER_SENSORS];

value[0] = map(flexR0, STRAIGHT_RESISTANCE0, BEND_RESISTANCE0, 0.0, 90.0);
value[1] = map(flexR1, STRAIGHT_RESISTANCE1, BEND_RESISTANCE1, 0.0, 90.0);
value[2] = map(flexR2, STRAIGHT_RESISTANCE2, BEND_RESISTANCE2, 0.0, 90.0);
value[3] = map(flexR3, STRAIGHT_RESISTANCE3, BEND_RESISTANCE3, 0.0, 90.0);
value[4] = map(flexR4, STRAIGHT_RESISTANCE4, BEND_RESISTANCE4, 0.0, 90.0);

Serial.print(value[0]);
Serial.print(", ");
Serial.print(value[1]);
Serial.print(", ");
Serial.print(value[2]);
Serial.print(", ");
Serial.print(value[3]);
Serial.print(", ");
Serial.print(value[4]);
Serial.print(", ");
```

Además de los valores de los sensores flexibles, se ha de conocer la orientación de la mano, que viene dada por los valores *Roll, Pitch y Yaw*, como se muestra en la Figura 165, aunque para el presente TFG solo se tienen en cuenta el *Pitch* y el *Roll*.

Figura 165. Orientación de la mano (Pitch, Roll y Yaw) [24]



Para ello, se obtienen los valores del acelerómetro de los ejes x,y,z y se convierten, mediante funciones matemáticas, a los valores de *Pitch* y *Roll* equivalentes (en radianes), para realizar posteriormente la conversión de radianes a grados e imprimirlos por pantalla, como se puede observar en la Figura 166.

Figura 166. Extracto de código - Pitch y Roll

```
float valueacc[2];
int16_t a[3];
a[0] = x;
a[1] = y;
a[2] = z;

valueacc[0] = atan2(-a[0], sqrt(a[1] * a[1] + a[2] * a[2])); // pitch
valueacc[1] = atan2(a[1], a[2]);  // roll
// Convert from radians to degrees:
valueacc[0] *= 180.0 / M_PI;
valueacc[1] *= 180.0 / M_PI;

Serial.print(valueacc[0]);
Serial.print(", ");
Serial.print(valueacc[1]);
```

Por último, se incrementa el valor de la variable `available`, y se reinicia el contador de muestras `sample_skip_counter`, como se expone en la Figura 167.

Figura 167. Extracto de código - Reinicio del contador

```
available++;
sample_skip_counter = 1;
```

Finalmente, en caso de que se produzca un error en la captura de los datos del acelerómetro, se muestra un mensaje por pantalla, tal y como se muestra en la Figura 168.

Figura 168. Extracto de código - Error

```
else {
  Serial.println("********************************");
  Serial.println("*** ERROR READING LSM9DS1 ***");
  Serial.println("********************************");
}
```

En caso contrario, se seguirá capturando datos tanto del acelerómetro como de los sensores *flex* hasta que se vuelva a presionar el botón **SetUp**.

Una vez ejecutado el *sketch*, se observan por el terminal serie resultados como los mostrados en la Figura 169.

Figura 169. Extracto de código - Resultados del sketch

```
-,-,-,-,-,-,-
39.38, 10.28, 25.68, 19.36, 18.04,9.380884432285614,5.32318633106943
39.20, 5.35, 24.59, 18.74, 17.80,10.10850326189681,6.463489195271751
41.35, 5.58, 24.13, 19.92, 17.65,10.250898258852716,5.7955946883130185
39.97, 6.55, 26.03, 19.45, 17.80,11.264757635413016,2.8762593140183754
42.89, 9.48, 26.93, 20.84, 18.50,11.258922936621838,5.439046074782783
39.32, 7.30, 25.37, 19.57, 17.74,10.617236249578102,6.195509925668143
43.08, 11.56, 26.30, 20.90, 18.35,8.550419395421,4.986084468428347
41.78, 11.25, 26.57, 20.84, 18.87,9.145765256395233,4.704136764344235
40.63, 16.36, 26.70, 21.13, 18.17,10.913807412234691,7.042757175941518
41.72, 13.74, 26.43, 20.87, 18.38,10.830137655711058,6.108417618419826
41.66, 13.52, 26.89, 21.16, 18.38,11.837269575373174,6.291790399643434
```

5.2 Preprocesamiento de los datos

Tal y como se expuso en el capítulo previo del presente TFG, a continuación, se lleva a cabo el procesamiento de los datos experimentales obtenidos a partir del *sketch* descrito, siendo éste el paso previo a la generación y entrenamiento del modelo.

Los *scripts* generados son similares a los expuestos en el capítulo anterior, así como la función que desempeña cada uno de ellos, si bien cabe destacar que existen ciertas diferencias que se exponen a continuación.

5.2.1 *Script* data_prepare.py

En este caso, el *script* es prácticamente el mismo que el utilizado en el capítulo anterior, y cumple la misma función. Sin embargo, hay ciertas diferencias:

- La primera y más evidente es que hay un cambio en cuanto a los valores de entrada asociados a cada una de las muestras, pasando de tres a siete, siendo estos los cinco correspondientes a los sensores flexibles más los valores de *pitch* y *roll* que se obtienen a partir de los datos proporcionados por el acelerómetro.

- Por otro lado, después del desarrollo del modelo del capítulo previo, se ha llegado a la conclusión de que la generación de nuevos valores negativos ayuda al reconocimiento de estos, pero no contribuye significativamente al reconocimiento de los gestos en sí, es por ello que se ha decidido prescindir de la función `generate_negative_data()`.

Por tanto, las funciones que componen este *script* son las ya conocidas `prepare_original_data()` y `write_data()`.

- Función `prepare_original_data()`

Esta función, cuyo código queda expuesto en la Figura 170, es muy parecida a la expuesta en el capítulo anterior, con el único cambio que se ha comentado previamente, de tres valores asociados a cada una de las muestras, se pasa a siete. Como se explicó, se encarga de leer los datos capturados en cada una de las muestras y organizarlos para facilitar su posterior procesamiento.

Figura 170. Extracto de código - Función preapre_original_data

```
def prepare_original_data(folder, name, data, file_to_read):  # pylint: disable=redefined-outer-name
  """Read collected data from files."""
  if folder != "negative":  # process symbol folders
    with open(file_to_read, "r") as f:
      lines = csv.reader(f)
      data_new = {}
      data_new[LABEL_NAME] = folder
      data_new[DATA_NAME] = []
      data_new["name"] = name
      for idx, line in enumerate(lines):  # pylint: disable=unused-variable,redefined-outer-name
        if len(line) == 7:
          if line[2] == "-" and data_new[DATA_NAME]:
            data.append(data_new)
            data_new = {}
            data_new[LABEL_NAME] = folder
            data_new[DATA_NAME] = []
            data_new["name"] = name
          elif line[2] != "-":
            data_new[DATA_NAME].append([float(i) for i in line[0:7]]) #SOLO CONSIDERAR LOS 5 PRIMEROS VALORES (FLEX)
      data.append(data_new)
  else:  # process negative folder
    with open(file_to_read, "r") as f:
      lines = csv.reader(f)
      data_new = {}
      data_new[LABEL_NAME] = folder
      data_new[DATA_NAME] = []
      data_new["name"] = name
      for idx, line in enumerate(lines):
        if len(line) == 7 and line[2] != "-":
          if len(data_new[DATA_NAME]) == 120:
            data.append(data_new)
            data_new = {}
            data_new[LABEL_NAME] = folder
            data_new[DATA_NAME] = []
            data_new["name"] = name
          else:
            data_new[DATA_NAME].append([float(i) for i in line[0:7]]) #SOLO CONSIDERAR LOS 5 PRIMEROS VALORES (FLEX)
      data.append(data_new)
```

- Función `write_data()`

En este caso, la función es idéntica a la presentada en el capítulo anterior, siendo la encargada de escribir los datos en los archivos correspondientes, como se puede observar en la Figura 171.

Figura 171. Extracto de código - Función write_data

```
# Write data to file
def write_data(data_to_write, path):
  with open(path, "w") as f:
    for idx, item in enumerate(data_to_write):  # pylint: disable=unused-variable,redefined-outer-name
      dic = json.dumps(item, ensure_ascii=False)
      f.write(dic)
      f.write("\n")
```

Por último, una vez se tienen los datos capturados y organizados, se escriben en los ficheros correspondientes, como se expone en la Figura 172. En este caso, se puede observar que se omite la llamada a la función `generate_negative_data()`, tal y como se expuso anteriormente.

Figura 172. Extracto de código - Main

```
if __name__ == "__main__":
  data = []  # pylint: disable=redefined-outer-name
  for idx1, folder in enumerate(folders):
    for idx2, name in enumerate(names):
      prepare_original_data(folder, name, data,
                            "./%s/output_%s_%s_flex_pr.txt" % (folder, folder, name))
  for idx in range(5):
    prepare_original_data("negative", "negative%d" % (idx + 1), data,
                          "./negative/output_negative_%d_flex_pr.txt" % (idx + 1))
#  generate_negative_data(data)
  print("data_length: " + str(len(data)))
  if not os.path.exists("./data"):
    os.makedirs("./data")
  write_data(data, "./data/complete_data")
```

5.2.2 *Script* data_split.py

Este *script* es idéntico al descrito en el capítulo anterior, compuesto por las funciones `read_data()` y `split_data()`, expuestas en la Figura 173 y en la Figura 174, respectivamente.

Figura 173. Extracto de código - Función read_data

```
# Read data
def read_data(path):
  data = []  # pylint: disable=redefined-outer-name
  with open(path, "r") as f:
    lines = f.readlines()
    for idx, line in enumerate(lines):  # pylint: disable=unused-variable
      dic = json.loads(line)
      data.append(dic)
  print("data_length:" + str(len(data)))
  return data
```

Figura 174. Extracto de código - Función split_data

```
def split_data(data, train_ratio, valid_ratio):  # pylint: disable=redefined-outer-name
  """Splits data into train, validation and test according to ratio."""
  train_data = []  # pylint: disable=redefined-outer-name
  valid_data = []  # pylint: disable=redefined-outer-name
  test_data = []  # pylint: disable=redefined-outer-name
  num_dic = {"symbold": 0, "symbole": 0, "symbolf": 0, "negative": 0}
  for idx, item in enumerate(data):  # pylint: disable=unused-variable
    for i in num_dic:
      if item["gesture"] == i:
        num_dic[i] += 1
  print(num_dic)
  train_num_dic = {}
  valid_num_dic = {}
  for i in num_dic:
    train_num_dic[i] = int(train_ratio * num_dic[i])
    valid_num_dic[i] = int(valid_ratio * num_dic[i])
  random.seed(30)
  random.shuffle(data)
  for idx, item in enumerate(data):
    for i in num_dic:
      if item["gesture"] == i:
        if train_num_dic[i] > 0:
          train_data.append(item)
          train_num_dic[i] -= 1
        elif valid_num_dic[i] > 0:
          valid_data.append(item)
          valid_num_dic[i] -= 1
        else:
          test_data.append(item)
  print("train_length:" + str(len(train_data)))
  print("valid_length:" + str(len(valid_data)))
  print("test_length:" + str(len(test_data)))
  return train_data, valid_data, test_data
```

5.3 Generación del modelo TensorFlow

Para la generación del modelo TF se han utilizado los mismos *scripts* que en el capítulo previo, aunque es importante destacar que alguno de ellos ha sufrido determinados cambios que se detallan a continuación. Además, se han desarrollado dos *scripts* Python adicionales, denominados **data_flex_pr.py** y **data_matplotlib_pr.py**, encargados de darle el formato adecuado a los datos para poder representarlos gráficamente y con ello facilitar un análisis previo.

5.3.1 *Script* data_load.py

El fichero **data_load.py** es, de nuevo, prácticamente idéntico al utilizado en el modelo anterior. El único cambio viene dado por el valor de la dimensión (`dim`), que anteriormente era 3 y ahora 7.

Las funciones que componen el *script* son `_init_()`, `get_data_file()`, `pad()`, `format_support_func()` y `format()`, explicadas en el capítulo previo y expuestas en la Figura 175 y en la Figura 176.

Figura 175. Extracto de código - Funciones _init_, get_data_file y pad

```
def __init__(self, train_data_path, valid_data_path, test_data_path,
             seq_length):
  self.dim = 7
  self.seq_length = seq_length

  print("self.seq_length = ", self.seq_length)

  self.label2id = {"symbold": 0, "symbole": 1, "symbolf": 2, "negative": 3}
  self.train_data, self.train_label, self.train_len = self.get_data_file(
      train_data_path, "train")
  self.valid_data, self.valid_label, self.valid_len = self.get_data_file(
      valid_data_path, "valid")
  self.test_data, self.test_label, self.test_len = self.get_data_file(
      test_data_path, "test")


def get_data_file(self, data_path, data_type):
  """Get train, valid and test data from files."""
  data = []
  label = []
  with open(data_path, "r") as f:
    lines = f.readlines()
    for idx, line in enumerate(lines):  # pylint: disable=unused-variable
      dic = json.loads(line)
      data.append(dic[DATA_NAME])
      label.append(dic[LABEL_NAME])

  if data_type == "train":
    data, label = augment_data(data, label)
  length = len(label)
  print(data_type + "_data_length:" + str(length))
  return data, label, length


def pad(self, data, seq_length, dim):
  """Get neighbour padding."""
  noise_level = 20
  padded_data = []
  # Before- Neighbour padding
  tmp_data = (np.random.rand(seq_length, dim) - 0.5) * noise_level + data[0]
  tmp_data[(seq_length -
  min(len(data), seq_length)):] = data[:min(len(data), seq_length)]
  padded_data.append(tmp_data)
  # After- Neighbour padding
  tmp_data = (np.random.rand(seq_length, dim) - 0.5) * noise_level + data[-1]
  tmp_data[:min(len(data), seq_length)] = data[:min(len(data), seq_length)]
  padded_data.append(tmp_data)
  return padded_data
```

Figura 176. Extracto de código - Funciones format_support_func y format

```
def format_support_func(self, padded_num, length, data, label):
  """Support function for format.(Helps format train, valid and test.)"""
  # Add 2 padding, initialize data and label
  length *= padded_num
  features = np.zeros((length, self.seq_length, self.dim))
  labels = np.zeros(length)
  # Get padding for train, valid and test
  for idx, (data, label) in enumerate(zip(data, label)):
    padded_data = self.pad(data, self.seq_length, self.dim)
    for num in range(padded_num):
      features[padded_num * idx + num] = padded_data[num]
      labels[padded_num * idx + num] = self.label2id[label]
  # Turn into tf.data.Dataset
  dataset = tf.data.Dataset.from_tensor_slices(
      (features, labels.astype("int32")))
  return length, dataset

def format(self):
  """Format data(including padding, etc.) and get the dataset for the model."""
  padded_num = 2
  self.train_len, self.train_data = self.format_support_func(
      padded_num, self.train_len, self.train_data, self.train_label)
  self.valid_len, self.valid_data = self.format_support_func(
      padded_num, self.valid_len, self.valid_data, self.valid_label)
  self.test_len, self.test_data = self.format_support_func(
      padded_num, self.test_len, self.test_data, self.test_label)
```

5.3.2 *Script* data_augmentation.py

Sin embargo, el *script* **data_augmentation.py**, que es llamado desde el fichero **data_load.py**, sí que sufre alguna variación adicional con respecto al empleado en el capítulo anterior, siendo la optimización del modelo la razón principal por la que se han llevado a cabo los cambios que se exponen a continuación.

Así, la función `augment_data()`, expuesta en la Figura 177, es sustancialmente diferente a la expuesta en el capítulo anterior, y esto se debe, principalmente, a que en este modelo se trabaja con otro tipo de datos. Mientras que anteriormente se obtenían los datos directamente de los tres ejes del acelerómetro, ahora se trabaja adicionalmente con ángulos, correspondientes a los valores *pitch, roll* y el ángulo de flexión de los cinco sensores *flex*, es por ello que la deformación temporal (*time warping)* o la amplificación de movimiento (*movement amplification)* tal cual se habían implementado se hacen innecesarias para este modelo.

Además, cabe destacar que para la generación de nuevas muestras aleatorias se han tenido que cambiar ciertos datos; concretamente se trata del valor del factor multiplicador utilizado para obtener dichas muestras. En el caso del *pitch* y el *roll,* ese valor es el doble del valor máximo que se ha obtenido en la generación de datos, ya que es multiplicado por un valor aleatorio entre -0.5 y +0.5, mientras que en el caso de los sensores *flex* es de 50, ya que los números aleatorios varían solo de 0 a 1.

Por último, destacar que se añade ruido aleatorio para generar más muestras, tal y como se realizó en el capítulo anterior.

Figura 177. Extracto de código - Función augment_data

```
def augment_data(original_data, original_label):
  """Perform data augmentation."""
  new_data = []
  new_label = []
  for idx, (data, label) in enumerate(zip(original_data, original_label)):  # pylint: disable=unused-variable
    # Original data
    new_data.append(data)
    new_label.append(label)
    # Sequence shift
    tmp_data = [[0 for i in range(len(data[0]))] for j in range(len(data))]
    for num in range(7):
      shiftf12345 = (random.random())
      shiftpr = (random.random() - 0.5)
      for i in range(len(tmp_data)):
        for j in range(len(tmp_data[i])):
          if j > 4: #pr
            tmp_data[i][j] = data[i][j] + shiftpr * 114.57992 #max tag value = 57.28996
          else: #flex12345
            tmp_data[i][j] = data[i][j] + shiftf12345 * 50
      new_data.append(tmp_data)
      new_label.append(label)
    # Random noise
    tmp_data = [[0 for i in range(len(data[0]))] for j in range(len(data))]
    for num in range(7):
      for i in range(len(tmp_data)):
        for j in range(len(tmp_data[i])):
          tmp_data[i][j] = data[i][j] + 5 * random.random()
      new_data.append(tmp_data)
      new_label.append(label)
  return new_data, new_label
```

Para el desarrollo de este nuevo modelo se ha tomado como referencia el presentado en el capítulo previo, teniendo en cuenta las principales diferencias y cambios que deben introducirse en éste. Sin embargo, en este caso, con el fin de validar los datos de entrada que han sido capturados en pasos anteriores, se consideró la implementación de dos *scripts* capaces de representar gráficamente dichos datos.

5.3.3 *Script* data_flex_pr.py

Consta de una sola función expuesta en la Figura 178, `prepare_original_data()`, cuyo objetivo es preparar los datos obtenidos, dándoles el formato necesario para poder ser representados gráficamente. Además, se realiza la llamada a dicha función, tal y como se expone en la Figura 179.

Figura 178. Extracto de código - Función prepare_original_data

```
def prepare_original_data(folder, name, data, file_to_read, path):  # pylint: disable=redefined-outer-name
  """Read collected data from files."""
#  if folder != "negative":
    # process symbol folders
  with open(file_to_read, "r") as fr:
    with open(path, "w") as fw:
      lines = csv.reader(fr)
      for idx, line in enumerate(lines):  # pylint: disable=unused-variable,redefined-outer-name
        if len(line) == 8:
          if line[2] != "-":
            for i in range(7):
              if i < 5:
                fw.write(line[i])
                fw.write(",")
              elif i == 5:
                a0 = float(line[5])
                a1 = float(line[6])
                a2 = float(line[7])
                pitch = math.atan2((-a0), math.sqrt((a1 * a1) + (a2 * a2)));
                pitch*= 180.0 / math.pi
                fw.write(str(pitch))
                fw.write(",")
              else:
                a1 = float(line[6])
                a2 = float(line[7])
                roll = math.atan2(a1, a2);
                roll*= 180.0 / math.pi
                fw.write(str(roll))
            fw.write("\n")
```

Figura 179. Extracto de código - Main

```
if __name__ == "__main__":
  data = []  # pylint: disable=redefined-outer-name
  for idx1, folder in enumerate(folders):
    for idx2, name in enumerate(names):
      if (folder == "negative"):
        prepare_original_data(folder, name, data, "./%s/output_%s_1.txt" % (folder, folder), "./%s/output_%s_1_flex_pr.txt" % (folder, folder))
        prepare_original_data(folder, name, data, "./%s/output_%s_2.txt" % (folder, folder), "./%s/output_%s_2_flex_pr.txt" % (folder, folder))
        prepare_original_data(folder, name, data, "./%s/output_%s_3.txt" % (folder, folder), "./%s/output_%s_3_flex_pr.txt" % (folder, folder))
        prepare_original_data(folder, name, data, "./%s/output_%s_4.txt" % (folder, folder), "./%s/output_%s_4_flex_pr.txt" % (folder, folder))
        prepare_original_data(folder, name, data, "./%s/output_%s_5.txt" % (folder, folder), "./%s/output_%s_5_flex_pr.txt" % (folder, folder))
      else:
        prepare_original_data(folder, name, data, "./%s/output_%s_%s.txt" % (folder, folder, name), "./%s/output_%s_%s_flex_pr.txt" % (folder, folder, name))
```

5.3.4 *Script* data_matplotlib_pr.py

Una vez obtenidos los datos en el formato adecuado, este archivo los representa gráficamente utilizando la librería de Python **Matplotlib**.

En primer lugar, se realiza la llamada a las distintas funciones que componen el *script*, como se muestra en la Figura 180.

Figura 180. Extracto de código - Main

```
if __name__ == '__main__':
    N = 7
    theta = radar_factory(N, frame='polygon')

    data = example_data()
    spoke_labels = data.pop(0)

    fig, axes = plt.subplots(figsize=(20, 20), nrows=1, ncols=3,
                             subplot_kw=dict(projection='radar'))
    fig.subplots_adjust(wspace=0.5, hspace=0.50, top=0.95, bottom=0.05)

    # Plot data on separate axes
    for ax, (title, case_data) in zip(axes.flat, data):
        ax.set_rgrids([25, 50, 75, 100, 125, 150, 175, 200])
        ax.set_title(title, weight='bold', size='large', position=(0.5, 1.6),
                     horizontalalignment='center', verticalalignment='center')
        for d in case_data:
            ax.plot(theta, d, scalex=False, scaley=False, color='b')
#             ax.fill(theta, d, facecolor='b', alpha=0.25)
        ax.set_varlabels(spoke_labels)

    # add legend relative to top-left plot
    ax = axes[0]
    labels = ('SymbolD', 'SymbolE', 'SymbolF')

    fig.text(0.5, 0.8, 'NewModelDef_flex TRR data',
             horizontalalignment='center', color='black', weight='bold',
             size='x-large')

    plt.show()
```

En este fichero se definen diversas funciones que se encargan del diseño y aspecto gráfico de los ejes, etiquetas, leyendas, etc… de una representación de tipo *Radar chart*, tal y como se exponen en la Figura 181 y en la Figura 182.

Figura 181. Extracto de código - Funciones encargadas del diseño y aspecto gráfico

```
def radar_factory(num_vars, frame='polygon'):
    """Create a radar chart with `num_vars` axes.

    This function creates a RadarAxes projection and registers it.

    Parameters
    ----------
    num_vars : int
        Number of variables for radar chart.
    frame : {'circle' | 'polygon'}
        Shape of frame surrounding axes.

    """
    # calculate evenly-spaced axis angles
    theta = np.linspace(0, 2*np.pi, num_vars, endpoint=False)

    class RadarAxes(PolarAxes):

        name = 'radar'
        # use 1 line segment to connect specified points
        RESOLUTION = 1

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # rotate plot such that the first axis is at the top
        self.set_theta_zero_location('N')

    def fill(self, *args, closed=True, **kwargs):
        """Override fill so that line is closed by default"""
        return super().fill(closed=closed, *args, **kwargs)

    def plot(self, *args, **kwargs):
        """Override plot so that line is closed by default"""
        lines = super().plot(*args, **kwargs)
        for line in lines:
            self._close_line(line)

    def _close_line(self, line):
        x, y = line.get_data()
        # FIXME: markers at x[0], y[0] get doubled-up
        if x[0] != x[-1]:
            x = np.concatenate((x, [x[0]]))
            y = np.concatenate((y, [y[0]]))
            line.set_data(x, y)

    def set_varlabels(self, labels):
        self.set_thetagrids(np.degrees(theta), labels)
```

Figura 182. Extracto de código - Funciones encargadas del diseño y aspecto gráfico (II)

```
    def _gen_axes_patch(self):
        # The Axes patch must be centered at (0.5, 0.5) and of radius 0.5
        # in axes coordinates.
        if frame == 'circle':
            return Circle((0.5, 0.5), 0.5)
        elif frame == 'polygon':
            return RegularPolygon((0.5, 0.5), num_vars,
                                  radius=.5, edgecolor="k")
        else:
            raise ValueError("unknown value for 'frame': %s" % frame)

    def _gen_axes_spines(self):
        if frame == 'circle':
            return super()._gen_axes_spines()
        elif frame == 'polygon':
            # spine_type must be 'left'/'right'/'top'/'bottom'/'circle'.
            spine = Spine(axes=self,
                          spine_type='circle',
                          path=Path.unit_regular_polygon(num_vars))
            # unit_regular_polygon gives a polygon of radius 1 centered at
            # (0, 0) but we want a polygon of radius 0.5 centered at (0.5,
            # 0.5) in axes coordinates.
            spine.set_transform(Affine2D().scale(.5).translate(.5, .5)
                                + self.transAxes)
            return {'polar': spine}
        else:
            raise ValueError("unknown value for 'frame': %s" % frame)

register_projection(RadarAxes)
return theta
```

Posteriormente, se empiezan a representar gráficamente los datos sobre los ejes definidos mediante la función expuesta en la Figura 183 y en la Figura 184.

Figura 183. Extracto de código - Función example_data

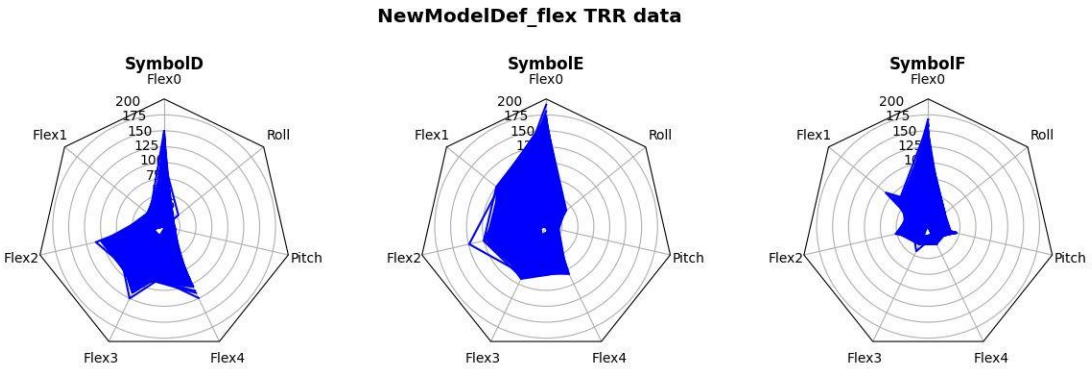
```
def example_data():
    g_data_d = np.genfromtxt("./symbold/output_symbold_trr_flex_pr.txt", dtype=np.float, delimiter=",")
    g_data_e = np.genfromtxt("./symbole/output_symbole_trr_flex_pr.txt", dtype=np.float, delimiter=",")
    g_data_f = np.genfromtxt("./symbolf/output_symbolf_trr_flex_pr.txt", dtype=np.float, delimiter=",")
    print("Flex sensor data d: \n" + str(g_data_d))
    print("Flex sensor data e: \n" + str(g_data_e))
    print("Flex sensor data f: \n" + str(g_data_f))

    # determine max values for g_data_d, g_data_e, g_data_f
    print("- MAX value g_data_d:")
    max_d0 = g_data_d[:,0].max()
    max_d1 = g_data_d[:,1].max()
    max_d2 = g_data_d[:,2].max()
    max_d3 = g_data_d[:,3].max()
    max_d4 = g_data_d[:,4].max()
    max_d5 = g_data_d[:,5].max()
    max_d6 = g_data_d[:,6].max()
    print("  %.2f %.2f %.2f %.2f %.2f - %.2f - %.2f" % (max_d0, max_d1, max_d2, max_d3, max_d4, max_d5, max_d6));

    print("- MAX value g_data_e:")
    max_e0 = g_data_e[:,0].max()
    max_e1 = g_data_e[:,1].max()
    max_e2 = g_data_e[:,2].max()
    max_e3 = g_data_e[:,3].max()
    max_e4 = g_data_e[:,4].max()
    max_e5 = g_data_e[:,5].max()
    max_e6 = g_data_e[:,6].max()
    print("  %.2f %.2f %.2f %.2f %.2f - %.2f - %.2f" % (max_e0, max_e1, max_e2, max_e3, max_e4, max_e5, max_e6));

    print("- MAX value g_data_f:")
    max_f0 = g_data_f[:,0].max()
    max_f1 = g_data_f[:,1].max()
    max_f2 = g_data_f[:,2].max()
    max_f3 = g_data_f[:,3].max()
    max_f4 = g_data_f[:,4].max()
    max_f5 = g_data_f[:,5].max()
    max_f6 = g_data_f[:,6].max()
    print("  %.2f %.2f %.2f %.2f %.2f - %.2f - %.2f" % (max_f0, max_f1, max_f2, max_f3, max_f4, max_f5, max_f6));
```

Figura 184. Extracto de código - Generación de datos

```
#generate data
data = [
    ['Flex0', 'Flex1', 'Flex2', 'Flex3', 'Flex4', 'Pitch', 'Roll'],
    ('SymbolD', g_data_d),
    ('SymbolE', g_data_e),
    ('SymbolF', g_data_f)
]
return data
```

Al ejecutar dichos ficheros sobre el conjunto de valores correspondientes a las muestras que constituyen los datos de entrada para cada uno de los gestos a reconocer, se obtienen gráficos como los que se muestran en la Figura 185.

Figura 185. Representación gráfica de los valores de los datos de entrada



Como se comentó anteriormente, se han tomado como referencia las letras D, E y F del alfabeto dactilológico de la lengua española de signos, representando, en este caso, los datos que corresponden a cada uno de los cinco sensores *flex,* junto con el *pitch* y *roll*. De esta manera, se facilita la validación visual de los datos de entrada, siendo cada símbolo fácilmente diferenciable y observándose una confusión reducida entre uno y otro.

5.3.5 *Script* train.py

A continuación, se crea el modelo a través del fichero **train.py**, aunque en esta ocasión, dicho modelo es el mismo que el generado en el capítulo anterior modificando una serie de parámetros. Para ello, se ha decidido seguir un proceso experimental con el fin de experimentar la influencia que tiene la variación de determinados parámetros en el modelo; concretamente, se han estudiado cinco casos distintos con el fin de elegir los parámetros adecuados que garantizan un modelo válido. Cabe destacar que este proceso se ha realizado, inicialmente, sin contar con los datos del *pitch* y *roll*, incluidos más adelante.

Dichas variaciones se concentran en cinco parámetros fundamentales del modelo, tales como el número de `epochs`, `batch_size`, `steps_per_epoch` y el número de filtros y tamaño del Kernel en las capas convolucionales.

Los primeros parámetros por modificar son `epochs` y `batch_size`, siendo ambos fundamentales en la generación del modelo. El primero de ellos, como se ha explicado previamente, indica el número de veces que los datos de entrenamiento han pasado por la red neuronal en el proceso de entrenamiento. Es fundamental ajustar dicho valor, ya que un número alto de *epochs* provocaría que el modelo se ajustase en exceso a los datos, pudiendo ocasionar problemas de generalización en el conjunto de datos de prueba y validación. El segundo de estos hiperparámetros determina el tamaño de los lotes en una iteración del entrenamiento para actualizar el gradiente. El tamaño óptimo depende de muchos factores, entre ellos de la capacidad de memoria del computador. Es por ello que se hace fundamental modificar estos parámetros con el fin de encontrar el compromiso adecuado entre pérdidas, precisión y tamaño del modelo.

Así, el parámetro `steps_per_epoch` se ha tratado de dos maneras diferentes, por un lado, se le ha dado un valor fijo, y por otro, se ha utilizado una de las propuestas usadas comunmente para calcularlo, que no es más que la división del número de muestras del conjunto de datos entre el tamaño de los lotes.

Por último, el número de filtros y tamaño del Kernel afectan, en gran medida, al tamaño del modelo, así como a la precisión y las pérdidas que presenta. De forma general, cuanto mayor sea el *grid*, mayor será la reducción del tamaño del modelo, además de que las pérdidas pueden aumentar y la precisión disminuir. Sin embargo, si esta ventana es más pequeña, es probable que la precisión y las pérdidas mejoren, pero el tamaño de dicho modelo aumente. Modificando estos valores se puede encontrar un compromiso entre estos parámetros.

A continuación, se exponen los cinco casos estudiados, además de analizar su influencia.

- **Caso 1**

En la Tabla 7 se muestran los valores establecidos, en este caso, para cada uno de los parámetros de la generación del modelo TF.

Tabla 7. Caso 1

Epochs	**50**
Batch_size	**64**
Filtros (1ª capa Conv2D)	**8**
Tamaño del Kernel (1ª capa Conv2D)	**2x5**
Filtros (2ª capa Conv2D)	**16**
Tamaño del Kernel (2ª capa Conv2D)	**4x1**
Steps per epoch	**1000**

Observando los resultados obtenidos para este primer caso, una vez finalizado el proceso de entrenamiento, los primeros que indican el comportamiento del modelo son la matriz de confusión, expuesta en la Figura 186, y las pérdidas y precisión de éste.

Figura 186. Matriz de confusión - Caso 1

```
tf.Tensor(
[[12  0  0  0]
 [ 0 12  0  0]
 [ 0  0 11  1]
 [ 0  0  0 14]], shape=(4, 4), dtype=int32)
Loss 1.1409385204315186, Accuracy 0.9800000190734863
```

Así, en este caso, la precisión del modelo ronda el 98%, valor muy cercano al 100%, mientras que las pérdidas son de 1.14 aproximadamente, siendo éste, a priori, un valor alto.

Además, otro registro a considerar es el tamaño del modelo, que se puede observar en la Figura 187, tanto en su versión básica como en su versión optimizada (cuantificada).

Figura 187. Tamaño - Caso 1

```
Basic model is 38320 bytes
Quantized model is 13984 bytes
Difference is 24336 bytes
```

En este caso, se obtiene un modelo básico de 38.32 KB y un modelo optimizado de 13.984 KB, valores que, comparándolos con los demás casos estudiados, ayudarán a determinar si el modelo es válido o existen mejores soluciones.

- **Caso 2**

En la Tabla 8 se muestran los valores establecidos, en este caso, para cada uno de los parámetros de la generación del modelo TF.

Tabla 8. Caso 2

Epochs	**50**
Batch_size	**64**
Filtros (1ª capa Conv2D)	**4**
Tamaño del Kernel (1ª capa Conv2D)	**2x5**
Filtros (2ª capa Conv2D)	**8**
Tamaño del Kernel (2ª capa Conv2D)	**4x1**
Steps per epoch	**1000**

En la Figura 188 se expone la matriz de confusión resultante para este caso en concreto.

Figura 188. Matriz de confusión - Caso 2

```
tf.Tensor(
[[11  1  0  0]
 [ 0 12  0  0]
 [ 0  0 11  1]
 [ 0  2  0 12]], shape=(4, 4), dtype=int32)
Loss 2.4917025566101074, Accuracy 0.9200000166893005
```

Para este segundo caso, se puede observar un aumento importante de las pérdidas, así como una disminución de la precisión, claros indicadores de que este modelo no parece el más adecuado.

Sin embargo, la parte positiva es que se reduce prácticamente en un 50% el tamaño del modelo en su versión básica, mientras que en su versión optimizada también disminuye hasta los 8.144 KB, tal y como se puede observar en la Figura 189.

Figura 189. Tamaño - Caso 2

```
Basic model is 20192 bytes
Quantized model is 8144 bytes
Difference is 12048 bytes
```

- **Caso 3**

En la Tabla 9 se muestran los valores establecidos, en este caso, para cada uno de los parámetros de la generación del modelo TF.

Tabla 9. Caso 3

Epochs	**50**
Batch_size	**32**
Filtros (1ª capa Conv2D)	**8**
Tamaño del Kernel (1ª capa Conv2D)	**2x5**
Filtros (2ª capa Conv2D)	**16**
Tamaño del Kernel (2ª capa Conv2D)	**4x1**
Steps per epoch	**Valor variable**

En la Figura 190 se expone la matriz de confusión resultante para este caso en concreto.

Figura 190. Matriz de confusión - Caso 3

```
tf.Tensor(
[[12  0  0  0]
 [ 0 12  0  0]
 [ 0  0 11  1]
 [ 0  0  0 14]], shape=(4, 4), dtype=int32)
Loss 0.6459611654281616, Accuracy 0.9800000190734863
```

En esta ocasión, se observa una evolución favorable de las pérdidas y la precisión, siendo éstas de 0.645 y del 98% respectivamente, siendo, en este aspecto, el mejor modelo de los tres estudiados hasta el momento.

En cuanto al tamaño, que se expone en la Figura 191, los valores son idénticos al primer modelo expuesto, por lo que se podría concluir que aquel modelo queda descartado, ya que se ha encontrado una mejor opción.

Figura 191. Tamaño - Caso 3

```
Basic model is 38320 bytes
Quantized model is 13984 bytes
Difference is 24336 bytes
```

- **Caso 4**

En la Tabla 10 se muestran los valores establecidos, en este caso, para cada uno de los parámetros de la generación del modelo TF.

Tabla 10. Caso 4

Epochs	**50**
Batch_size	**32**
Filtros (1ª capa Conv2D)	**6**
Tamaño del Kernel (1ª capa Conv2D)	**2x5**
Filtros (2ª capa Conv2D)	**12**
Tamaño del Kernel (2ª capa Conv2D)	**2x2**
Steps per epoch	**Valor variable**

En la Figura 192 se expone la matriz de confusión resultante para este caso en concreto.

Figura 192. Matriz de confusión - Caso 4

```
tf.Tensor(
[[12  0  0  0]
 [ 0 12  0  0]
 [ 0  0 11  1]
 [ 0  0  0 14]], shape=(4, 4), dtype=int32)
Loss 0.3645670413970947, Accuracy 0.9800000190734863
```

Como se puede observar, la evolución sigue siendo favorable, ya que la precisión del modelo se mantiene, pero las pérdidas vuelven a reducirse prácticamente en un 50% con respecto al modelo anterior.

Además, en este caso, el tamaño de las dos versiones del modelo también se reduce, como se observa en la Figura 193, por lo tanto, se puede determinar que este es el modelo más adecuado de los casos considerados hasta el momento.

Figura 193. Tamaño - Caso 4

```
Basic model is 29128 bytes
Quantized model is 10928 bytes
Difference is 18200 bytes
```

- **Caso 5**

En la Tabla 11 se muestran los valores establecidos, en este caso, para cada uno de los parámetros de la generación del modelo TF.

Tabla 11. Caso 5

Epochs	**50**
Batch_size	**16**
Filtros (1ª capa Conv2D)	**6**
Tamaño del Kernel (1ª capa Conv2D)	**2x5**
Filtros (2ª capa Conv2D)	**12**
Tamaño del Kernel (2ª capa Conv2D)	**2x2**
Steps per epoch	**Valor variable**

En la Figura 194 se expone la matriz de confusión resultante para este caso en concreto.

Figura 194. Matriz de confusión - Caso 5

```
tf.Tensor(
[[12  0  0  0]
 [ 0 12  0  0]
 [ 0  0 11  1]
 [ 0  0  0 14]], shape=(4, 4), dtype=int32)
Loss 0.2741382420063019, Accuracy 0.9800000190734863
```

En el último caso estudiado se vuelve a observar una disminución de las pérdidas, así como una precisión similar a las obtenidas en los modelos anteriores, siendo ésta muy cercana al 100%.

Asimismo, el tamaño de ambas versiones es idéntico al del modelo anterior, como se expone en la Figura 195, por lo tanto, todos los modelos anteriores quedan descartados, ya que se ha encontrado una opción donde el compromiso entre pérdidas, precisión y tamaño del modelo se puede considerar la mejor de las opciones analizadas.

Figura 195. Tamaño - Caso 5

```
Basic model is 29128 bytes
Quantized model is 10928 bytes
Difference is 18200 bytes
```

Es probable que, modificando aún más los valores y estudiando más casos se encuentre un modelo óptimo, pero los resultados de este último se han considerado lo suficientemente válidos para la tarea a realizar en el presente TFG. Sin embargo, esto se podrá comprobar más adelante, cuando se termine por completo el desarrollo del modelo y se observen las prestaciones definitivas.

Como se explicó previamente, el modelo es, básicamente, el mismo que el generado en el capítulo anterior, modificando los parámetros mencionados a los valores indicados e incluyendo los cinco sensores *flex* y los valores de *pitch* y *roll,* tal y como se expone en el extracto de código que se observa en la Figura 196.

Figura 196. Extracto de código - Capas del modelo

```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(
        8, (2, 7),
        padding="same",
        activation="relu",
        input_shape=(seq_length, 7, 1)),
    tf.keras.layers.MaxPool2D((2, 2)),  # (batch, 42, 1, 8)

    tf.keras.layers.Dropout(0.1),  # (batch, 42, 1, 8)
    tf.keras.layers.Conv2D(16, (2, 2), padding="same",
                           activation="relu"),  # (batch, 42, 1, 16)
    tf.keras.layers.MaxPool2D((2, 2), padding="same"),  # (batch, 14, 1, 16)

    tf.keras.layers.Dropout(0.1),  # (batch, 14, 1, 16)
    tf.keras.layers.Flatten(),  # (batch, 224)
    tf.keras.layers.Dense(16, activation="relu"),  # (batch, 16)
    tf.keras.layers.Dropout(0.1),  # (batch, 16)
    tf.keras.layers.Dense(4, activation="softmax")  # (batch, 4)
```

Está formado por las mismas capas que las descritas en el capítulo anterior, cuyas funciones han sido expuestas.

El único cambio respecto al modelo anterior, donde se incluyen el *pitch* y *roll*, viene dado en la función `reshape_function()`, expuesta en la Figura 197, donde se indican los datos de entrada que se utilizan para el entrenamiento, siendo en este caso de 7, los cinco sensores *flex*, el *pitch* y el *roll.*

Figura 197. Extracto de código - Función reshape

```
def reshape_function(data, label):
  reshaped_data = tf.reshape(data, [-1, 7, 1])

  return reshaped_data, label
```

5.4 Resultados de la generación del modelo

Es importante resaltar que al introducir los cinco sensores *flex* y los valores del *pitch* y el *roll,* se han tenido que modificar algunos parámetros de nuevo, ya que esto ha conllevado a un incremento de los problemas de *overfitting* del modelo. Para ello se ha seguido un estudio similar al explicado anteriormente, donde se han determinado los valores expuestos en la Figura 198 para los parámetros `epochs` y `batch_size`.

Figura 198. Extracto de código - Valores de epoch y batch_size

```
epochs = 100 #50
batch_size = 16 #16
```

Tras ese proceso, y tal y como se puede observar, se ha duplicado el número de `epochs` respecto al valor original, mientras que el valor del parámetro `batch_size` no varía.

A continuación, durante la fase de entrenamiento, se pueden analizar diversos registros que proporcionan información sobre el modelo que se está generando, como se puede observar en la Figura 199. Así, en primer lugar, se observan las capas y sus parámetros.

Figura 199. Capas y parámetros del modelo

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 128, 7, 8)         120
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 64, 3, 8)          0
_________________________________________________________________
dropout (Dropout)            (None, 64, 3, 8)          0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 64, 3, 16)         528
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 32, 2, 16)         0
_________________________________________________________________
dropout_1 (Dropout)          (None, 32, 2, 16)         0
_________________________________________________________________
flatten (Flatten)            (None, 1024)              0
_________________________________________________________________
dense (Dense)                (None, 16)                16400
_________________________________________________________________
dropout_2 (Dropout)          (None, 16)                0
_________________________________________________________________
dense_1 (Dense)              (None, 4)                 68
=================================================================
Total params: 17,116
Trainable params: 17,116
Non-trainable params: 0
```

Como se explicó previamente, consta de las mismas capas que el modelo anterior, en este caso con un total de 17116 parámetros.

Además, se puede observar en la Figura 200 el tamaño del modelo antes de ser entrenado. Éste se incrementará al entrenarse, aunque se verá reducido en su versión optimizada.

Figura 200. Tamaño del modelo previo al entrenamiento

```
Model size: 66.859375 KB
```

En las primeras métricas del entrenamiento se puede observar cómo a medida que avanza el proceso, disminuyen las pérdidas y aumenta la precisión, tal y como se expone en la Figura 201.

Figura 201. Métricas iniciales

```
Epoch 2/100
141/141 [==============================] - 2s 15ms/step - loss: 1.2324 - accuracy: 0.4200 - val_loss: 1.0682 - val_accuracy: 0.4400
Epoch 3/100
141/141 [==============================] - 2s 15ms/step - loss: 1.1747 - accuracy: 0.3773 - val_loss: 1.0502 - val_accuracy: 0.5000
Epoch 4/100
141/141 [==============================] - 2s 14ms/step - loss: 1.0927 - accuracy: 0.4449 - val_loss: 1.0434 - val_accuracy: 0.4800
Epoch 5/100
141/141 [==============================] - 2s 14ms/step - loss: 1.0740 - accuracy: 0.4036 - val_loss: 1.0046 - val_accuracy: 0.5000
Epoch 6/100
141/141 [==============================] - 2s 16ms/step - loss: 1.0284 - accuracy: 0.4827 - val_loss: 1.0703 - val_accuracy: 0.5200
Epoch 7/100
141/141 [==============================] - 2s 13ms/step - loss: 1.0608 - accuracy: 0.4680 - val_loss: 1.0827 - val_accuracy: 0.5000
Epoch 8/100
141/141 [==============================] - 2s 13ms/step - loss: 1.0263 - accuracy: 0.5067 - val_loss: 1.0538 - val_accuracy: 0.7000
Epoch 9/100
141/141 [==============================] - 2s 12ms/step - loss: 0.9135 - accuracy: 0.5907 - val_loss: 0.7981 - val_accuracy: 0.9200
Epoch 10/100
141/141 [==============================] - 2s 12ms/step - loss: 0.7526 - accuracy: 0.7453 - val_loss: 0.9545 - val_accuracy: 0.8400
```

Al finalizar el entrenamiento se puede observar, en la Figura 202, una gran diferencia respecto a las métricas previas.

Figura 202. Métricas al finalizar el entrenamiento

```
Epoch 92/100
141/141 [==============================] - 2s 14ms/step - loss: 0.4242 - accuracy: 0.7351 - val_loss: 0.2808 - val_accuracy: 0.7600
Epoch 93/100
141/141 [==============================] - 2s 12ms/step - loss: 0.2898 - accuracy: 0.7533 - val_loss: 0.2271 - val_accuracy: 0.7600
Epoch 94/100
141/141 [==============================] - 2s 16ms/step - loss: 0.2468 - accuracy: 0.7680 - val_loss: 0.2021 - val_accuracy: 1.0000
Epoch 95/100
141/141 [==============================] - 2s 13ms/step - loss: 0.2274 - accuracy: 0.9711 - val_loss: 0.1830 - val_accuracy: 1.0000
Epoch 96/100
141/141 [==============================] - 2s 14ms/step - loss: 0.2136 - accuracy: 0.9711 - val_loss: 0.1668 - val_accuracy: 1.0000
Epoch 97/100
141/141 [==============================] - 3s 18ms/step - loss: 0.1885 - accuracy: 0.9764 - val_loss: 0.1528 - val_accuracy: 1.0000
Epoch 98/100
141/141 [==============================] - 2s 15ms/step - loss: 0.1857 - accuracy: 0.9720 - val_loss: 0.1388 - val_accuracy: 1.0000
Epoch 99/100
141/141 [==============================] - 2s 14ms/step - loss: 0.1867 - accuracy: 0.9667 - val_loss: 0.1290 - val_accuracy: 1.0000
Epoch 100/100
141/141 [==============================] - 2s 13ms/step - loss: 0.1610 - accuracy: 0.9747 - val_loss: 0.1187 - val_accuracy: 1.0000
```

Por un lado, las pérdidas son cercanas a 0, y por otro, la precisión sobrepasa el 95%. Además, las pérdidas de valoración son, de forma general, inferiores a las pérdidas de entrenamiento, clave que indica que, previsiblemente, no se generan problemas de *overfitting*.

Asimismo, se puede observar en la Figura 203 la matriz de confusión, así como el valor final de las pérdidas y precisión del modelo.

Figura 203. Matriz de confusión, pérdidas y precisión

```
tf.Tensor(
[[12  0  0  0]
 [ 0 12  0  0]
 [ 0  0 11  1]
 [ 0  1  0 13]], shape=(4, 4), dtype=int32)
Loss 1.1182987689971924, Accuracy 0.9599999785423279
```

La precisión consigue sobrepasar el 95%, mientras que las pérdidas se incrementan a 1.1, valor que se puede considerar alto, pero teniendo en cuenta la precisión del modelo, más que aceptable para el objetivo de este TFG.

Por último, en la Figura 204 se expone el tamaño obtenido para este modelo.

Figura 204. Tamaños del modelo

```
Basic model is 71216 bytes
Quantized model is 22304 bytes
Difference is 48912 bytes
```

En consecuencia, el modelo básico tiene un tamaño de 71.2 KB y el modelo optimizado 22.3 KB. Como bien se ha expuesto, al realizar la optimización del modelo se pierde cierta información en favor de la reducción de tamaño, sin embargo, ambas versiones son válidas y pueden usarse perfectamente en el dispositivo Argon, es por ello que se ha decidido elegir el modelo básico, al no comprometer los recursos de almacenamiento disponibles en dicho dispositivo.

5.4.1 Conversión a un archivo de C

El proceso para convertir a un archivo C es exactamente el mismo que el explicado en el capítulo anterior, obteniendo las dos versiones del modelo, básico y optimizado, como se puede observar en la Figura 205.

Figura 205. Versiones .cc y .tflite del modelo

model.cc	01/02/2021 15:30	Archivo CC	429 KB
model.tflite	12/05/2021 11:06	Archivo TFLITE	70 KB
model_quantized.cc	01/02/2021 15:30	Archivo CC	135 KB
model_quantized.tflite	12/05/2021 11:06	Archivo TFLITE	22 KB

5.5 Implementación de la aplicación

5.5.1 Estructura de la aplicación

La estructura de la aplicación es similar a la utilizada en el capítulo anterior, tal y como se observa a continuación. Ésta se compone de seis elementos fundamentales.

- ***Main Loop***
- ***Flex handler***
- ***TF Lite Interpreter***
- **Modelo**
- ***Gesture predictor***
- ***Output handler***

Además, los ficheros que componen dicha aplicación son los que se exponen seguidamente.

- ***constants.h, constants.cpp***
- ***main.cpp***
- ***tf_flex_pr.cpp*** (main del programa)
- ***main_functions.h***
- ***output_handler.h, particle_output_handler.cpp***
- ***flex_handler.h, particle_flex_handler.cpp***
- ***gesture_predictor.h, gesture predictor.cpp***
- ***flex_model_data.h, flex_model_data.cpp***

De los cuatro ficheros fundamentales de la aplicación, dos de ellos son prácticamente idénticos a los utilizados en el capítulo previo, correspondientes a **gesture_predictor** y **output_handler**. El único cambio viene dado en el primero de ellos, en el que el valor umbral o probabilidad para reconocer un gesto se ha aumentado de un 0,8 a un 0,9, y esto se debe a un proceso experimental con el fin de comprobar que aumentando dicho valor, los gestos se siguen reconociendo adecuadamente. Esto significa que para que un gesto sea reconocido debe identificarse el número de veces establecido (4) a partir de las inferencias identificadas por el modelo TF.

Sin embargo, los otros dos ficheros, correspondientes a **flex_handler** y la función *main* del programa (**tf_flex_pr**) sí que sufren cambios respecto al capítulo anterior y se exponen a continuación.

***Flex handler* (Gestor de los *flex*)**

En este caso lleva implícito el gestor del acelerómetro, que se encarga de leer los datos del acelerómetro, y convertirlos en los correspondientes valores de *pitch* y *roll.* Además, captura los datos de los sensores *flex* y los escribe en el *buffer* de entrada del modelo.

Para ello, consta de una sola función cuyo código, que se expone en la Figura 206, es muy similar al *sketch* utilizado para la captura de datos.

Figura 206. Extracto de código - Función ReadFlex

```
bool ReadFlex(tflite::ErrorReporter *error_reporter, float *input,
              int length, bool reset_buffer)
{
  // Clear the buffer if required, e.g. after a successful prediction
  if (reset_buffer)
  {
    memset(save_data, 0, 1400 * sizeof(float));
    begin_index = 0;
    pending_initial_data = true;
  }
  // Keep track of whether we stored any new data
  bool new_data = false;
  // Loop through new samples and add to buffer
  while (IMU.accelerationAvailable())
  {
    float x, y, z;
    // Read each sample, removing it from the device's FIFO buffer
    if (!IMU.readAcceleration(x, y, z))
    {
      error_reporter->Report("Failed to read data");
      break;
    }
    // Throw away this sample unless it's the nth
    if (sample_skip_counter != sample_every_n)
    {
      sample_skip_counter += 1;
      continue;
    }
```

En esta primera parte del código, se inicializa el *buffer* si es necesario, además de iniciar la función `loop`, donde se van añadiendo nuevos datos al *buffer*.

Con los datos obtenidos se calcula el valor del voltaje y la resistencia correspondiente a cada uno de los cinco sensores *flex* para, posteriormente, estimar el ángulo de doblez de cada uno de ellos, como se observa en la Figura 207.

Figura 207. Extracto de código - Cálculo del valor del voltaje y resistencia

```
int flexADC0 = analogRead(FLEX_PIN0);
float flexV0 = flexADC0 * VCC / 4095.0;
float flexR0 = R_DIV*(VCC/flexV0-1.0);

int flexADC1 = analogRead(FLEX_PIN1);
float flexV1 = flexADC1 * VCC / 4095.0;
float flexR1 = R_DIV*(VCC/flexV1-1.0);

int flexADC2 = analogRead(FLEX_PIN2);
float flexV2 = flexADC2 * VCC / 4095.0;
float flexR2 = R_DIV*(VCC/flexV2-1.0);

int flexADC3 = analogRead(FLEX_PIN3);
float flexV3 = flexADC3 * VCC / 4095.0;
float flexR3 = R_DIV*(VCC/flexV3-1.0);

int flexADC4 = analogRead(FLEX_PIN4);
float flexV4 = flexADC4 * VCC / 4095.0;
float flexR4 = R_DIV*(VCC/flexV4-1.0);

// Use the calculated resistance to estimate the sensor's bend angle
float value[NUMBER_SENSORS];
float value_acc[NUMBER_ACC_VALUES];

value[0] = map(flexR0, STRAIGHT_RESISTANCE0, BEND_RESISTANCE0, 0.0, 90.0);
value[1] = map(flexR1, STRAIGHT_RESISTANCE1, BEND_RESISTANCE1, 0.0, 90.0);
value[2] = map(flexR2, STRAIGHT_RESISTANCE2, BEND_RESISTANCE2, 0.0, 90.0);
value[3] = map(flexR3, STRAIGHT_RESISTANCE3, BEND_RESISTANCE3, 0.0, 90.0);
value[4] = map(flexR4, STRAIGHT_RESISTANCE4, BEND_RESISTANCE4, 0.0, 90.0);
```

A continuación, se calcula el valor del *pitch* y *roll* a partir de los valores correspondientes a las coordenadas x,y,z proporcionadas por el acelerómetro y se escriben las muestras en el *buffer*, como se observa en el extracto de código de la Figura 208.

Figura 208. Extracto de código - Cálculo del pitch y roll

```
// Calculate Pitch and Roll
value_acc[0] = atan2(-x, sqrt(y * y + z * z));
value_acc[1] = atan2(y, z);
   // Convert from radians to degrees:
value_acc[0] *= 180.0 / M_PI;
value_acc[1] *= 180.0 / M_PI;

// Write samples to our buffer
save_data[begin_index++] = value[0];
save_data[begin_index++] = value[1];
save_data[begin_index++] = value[2];
save_data[begin_index++] = value[3];
save_data[begin_index++] = value[4];
save_data[begin_index++] = value_acc[0];
save_data[begin_index++] = value_acc[1];
```

Por último, se realizan diversas acciones relacionadas con la inicialización de contadores, espera, retornos, etc…, tal y como se expone en la Figura 209, ya explicado en el capítulo anterior.

Figura 209. Extracto de código - Acciones de inicialización

```
  // Since we took a sample, reset the skip counter
  sample_skip_counter = 1;
  // If we reached the end of the circle buffer, reset
  if (begin_index >= 1000)
  {
    begin_index = 0;
  }
  new_data = true;
}

// Skip this round if data is not ready yet
if (!new_data)
{
  return false;
}

// Check if we are ready for prediction or still pending more initial data
if (pending_initial_data && begin_index >= 200)
{
  pending_initial_data = false;
}

// Return if we don't have enough data
if (pending_initial_data)
{
  return false;
}

// Copy the requested number of bytes to the provided input tensor
for (int i = 0; i < length; ++i)
{
  int ring_array_index = begin_index + i - length;
  if (ring_array_index < 0)
  {
    ring_array_index += 1400;
  }
  input[i] = save_data[ring_array_index];
}

return true;
```

Función principal (*main)*

Estos tres componentes de los que se ha hablado previamente se unen en el fichero **tf_flex_pr.cpp**, en el que se encuentra la función principal de la aplicación (*main).* El archivo es muy similar al utilizado en la aplicación del capítulo anterior.

En primer lugar, se configuran diversas variables, así como algunos extras, como se expone en la Figura 210.

Figura 210. Extracto de código - Configuración de variables

```
namespace micro
{
TfLiteRegistration *Register_DEPTHWISE_CONV_2D();
TfLiteRegistration *Register_MAX_POOL_2D();
TfLiteRegistration *Register_CONV_2D();
TfLiteRegistration *Register_FULLY_CONNECTED();
TfLiteRegistration *Register_SOFTMAX();
TfLiteRegistration *Register_RESHAPE();

} // namespace micro
} // namespace ops
} // namespace tflite

// Globals, used for compatibility with Arduino-style sketches.
namespace
{
tflite::ErrorReporter *error_reporter = nullptr;
const tflite::Model *model = nullptr;
tflite::MicroInterpreter *interpreter = nullptr;
TfLiteTensor *model_input = nullptr;
int input_length;

// Create an area of memory to use for input, output, and intermediate arrays.
// The size of this will depend on the model you're using, and may need to be
// determined by experimentation.
// If you're getting Red SOS Heap fragmentation errors on Argon or Boron devices,
// reduce the size of the tensor arena product from 60 to 50 or 40.
constexpr int kTensorArenaSize = 40 * 1024;
uint8_t tensor_arena[kTensorArenaSize];

// Whether we should clear the buffer next time we fetch data
bool should_clear_buffer = false;
} // namespace
// Globals, used for compatibility with Arduino-style sketches.
namespace
{
tflite::ErrorReporter *error_reporter = nullptr;
const tflite::Model *model = nullptr;
tflite::MicroInterpreter *interpreter = nullptr;
TfLiteTensor *model_input = nullptr;
int input_length;

// Create an area of memory to use for input, output, and intermediate arrays.
// The size of this will depend on the model you're using, and may need to be
// determined by experimentation.
// If you're getting Red SOS Heap fragmentation errors on Argon or Boron devices,
// reduce the size of the tensor arena product from 60 to 50 or 40.
constexpr int kTensorArenaSize = 60 * 1024;
uint8_t tensor_arena[kTensorArenaSize];

// Whether we should clear the buffer next time we fetch data
bool should_clear_buffer = false;
} // namespace
```

La variable `input_length` almacena la longitud del tensor de entrada del modelo, mientras que la variable `should_clear_buffer` es un *flag* que indica si se debe

inicializar el *buffer* en la próxima ejecución, lo cual se hace después de reconocer adecuadamente un gesto.

La función `setup()` realiza todos los preparativos necesarios para poder ejecutar inferencias, tal y como se observa en la Figura 211 y en la Figura 212.

Figura 211. Extracto de código - Función SetUp

```
// The name of this function is important for Arduino compatibility.
void setup()
{
  // Set up logging. Google style is to avoid globals or statics because of
  // lifetime uncertainty, but since this has a trivial destructor it's okay.
  static tflite::MicroErrorReporter micro_error_reporter; // NOLINT
  error_reporter = &micro_error_reporter;

  // Map the model into a usable data structure. This doesn't involve any
  // copying or parsing, it's a very lightweight operation.
  model = tflite::GetModel(g_magic_wand_model_data);
  if (model->version() != TFLITE_SCHEMA_VERSION)
  {
    error_reporter->Report(
        "Model provided is schema version %d not equal "
        "to supported version %d.",
        model->version(), TFLITE_SCHEMA_VERSION);
    return;
  }

  // Pull in only the operation implementations we need.
  // This relies on a complete list of all the ops needed by this graph.
  // An easier approach is to just use the AllOpsResolver, but this will
  // incur some penalty in code space for op implementations that are not
  // needed by this graph.
  static tflite::MicroMutableOpResolver micro_mutable_op_resolver; // NOLINT
  micro_mutable_op_resolver.AddBuiltin(
      tflite::BuiltinOperator_DEPTHWISE_CONV_2D,
      tflite::ops::micro::Register_DEPTHWISE_CONV_2D());
```

Figura 212. Extracto de código - Función SetUp (II)

```
micro_mutable_op_resolver.AddBuiltin(
    tflite::BuiltinOperator_MAX_POOL_2D,
    tflite::ops::micro::Register_MAX_POOL_2D());
micro_mutable_op_resolver.AddBuiltin(tflite::BuiltinOperator_CONV_2D,
                                     tflite::ops::micro::Register_CONV_2D());
micro_mutable_op_resolver.AddBuiltin(
    tflite::BuiltinOperator_FULLY_CONNECTED,
    tflite::ops::micro::Register_FULLY_CONNECTED());
micro_mutable_op_resolver.AddBuiltin(tflite::BuiltinOperator_SOFTMAX,
                                     tflite::ops::micro::Register_SOFTMAX());



micro_mutable_op_resolver.AddBuiltin(tflite::BuiltinOperator_RESHAPE,
                                     tflite::ops::micro::Register_RESHAPE());



// Build an interpreter to run the model with
static tflite::MicroInterpreter static_interpreter(
    model, micro_mutable_op_resolver, tensor_arena, kTensorArenaSize,
    error_reporter);
interpreter = &static_interpreter;

// Allocate memory from the tensor_arena for the model's tensors
interpreter->AllocateTensors();

// Obtain pointer to the model's input tensor
model_input = interpreter->input(0);
if ((model_input->dims->size != 4) || (model_input->dims->data[0] != 1) ||
    (model_input->dims->data[1] != 128) ||
    (model_input->dims->data[2] != kChannelNumber) ||
    (model_input->type != kTfLiteFloat32))
{
  error_reporter->Report("Bad input tensor parameters in model");
  return;
}

input_length = model_input->bytes / sizeof(float);

TfLiteStatus setup_status = SetupAccelerometer(error_reporter);
if (setup_status != kTfLiteOk)
{
  error_reporter->Report("Set up failed\n");
}
```

Por último, en la función `loop`, expuesta en la Figura 213Figura 214, se leen los valores de los sensores para, posteriormente, ajustar la variable `should_clear_buffer` a *false*, asegurando así que no se siga intentando inicializar dicho *buffer* más de una vez.

Si no se obtienen nuevos datos, la función `ReadFlex()` retorna *false,* y se retornará del *loop* e intentará de nuevo la próxima vez que se llame a la función.

Figura 213. Extracto de código - Función loop

```
void loop()
{
  // Attempt to read new data
  bool got_data = ReadFlex(error_reporter, model_input->data.f,
                           input_length, should_clear_buffer);

  //Serial.println("Data: %d", got_data);

  // Don't try to clear the buffer again
  should_clear_buffer = false;
  // If there was no new data, wait until next time
  if (!got_data)
    return;
  // Run inference, and report any error
  TfLiteStatus invoke_status = interpreter->Invoke();
  if (invoke_status != kTfLiteOk)
  {
    error_reporter->Report("Invoke failed on index: %d\n", begin_index);
    return;
  }
  // Analyze the results to obtain a prediction


    int pred_symbolD = (int)(100 * interpreter->output(0)->data.f[0]);
    int pred_symbolE = (int)(100 * interpreter->output(0)->data.f[1]);
    int pred_symbolF = (int)(100 * interpreter->output(0)->data.f[2]);


    error_reporter->Report("D: %d%%    E: %d%%    F: %d%%", pred_symbolD, pred_symbolE, pred_symbolF);

  int gesture_index = PredictGesture(interpreter->output(0)->data.f);
  // Clear the buffer next time we read data
  should_clear_buffer = gesture_index < 3;
  // Produce an output
  HandleOutput(error_reporter, gesture_index);
}
```

Si el valor que retorna `ReadFlex()` es *true*, se ejecuta la inferencia en el tensor de entrada y se pasa el resultado a la función `PredictGesture()`, que proporciona el índice del gesto que ha sido detectado. Si dicho índice es menor a 3, el gesto es válido, reajustándose el valor del *flag* `should_clear_buffer` para inicializar el *buffer* la próxima vez que se llame a la función `ReadFlex()`. Por último, se llama a la función `HandleOutput()` para reportar los resultados al usuario.

Además, en el archivo **particle_main.cpp** se llama automáticamente a las funciones `setup()` y `loop()`, por lo tanto no es necesario utilizar su propia rutina para ello, y con ello se consigue ahorrar en recursos.

5.5.2 Validación experimental

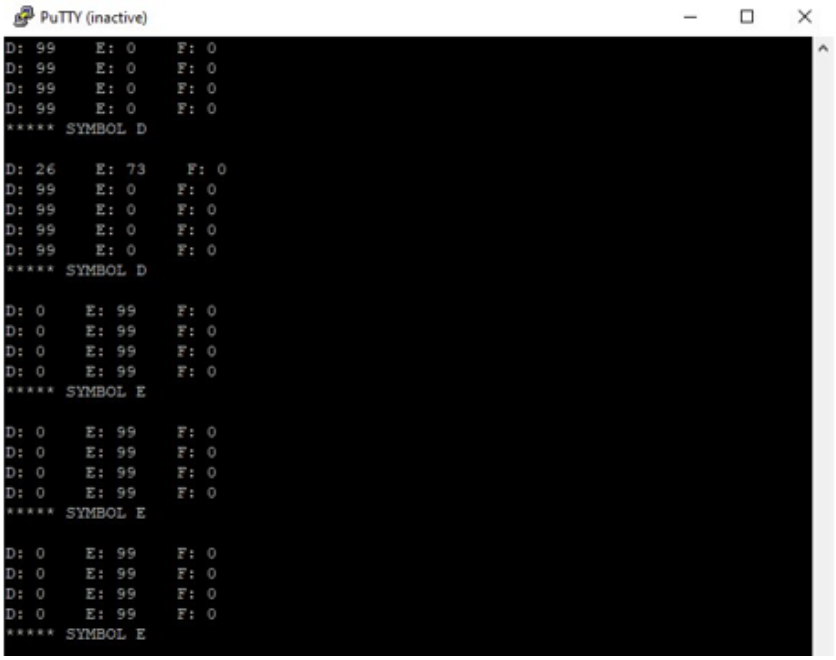
El último paso es validar los resultados obtenidos o comprobar el correcto funcionamiento del modelo desarrollado, satisfaciendo las necesidades iniciales y cumpliendo los objetivos establecidos.

Los gestos a realizar son las letras D, E y F del alfabeto dactilológico que se exponen en la Figura 214. Para una buena interpretación de los gestos se debe tener en cuenta el movimiento de los dedos, así como la orientación de la mano.

Figura 214. Letras D, E y F del alfabeto dactilológico de la lengua de signos española

Tras ejecutar la aplicación, se lee el puerto serie y se obtienen resultados como los que se exponen en la Figura 215.

Figura 215. Resultados de la lectura del puerto-serie

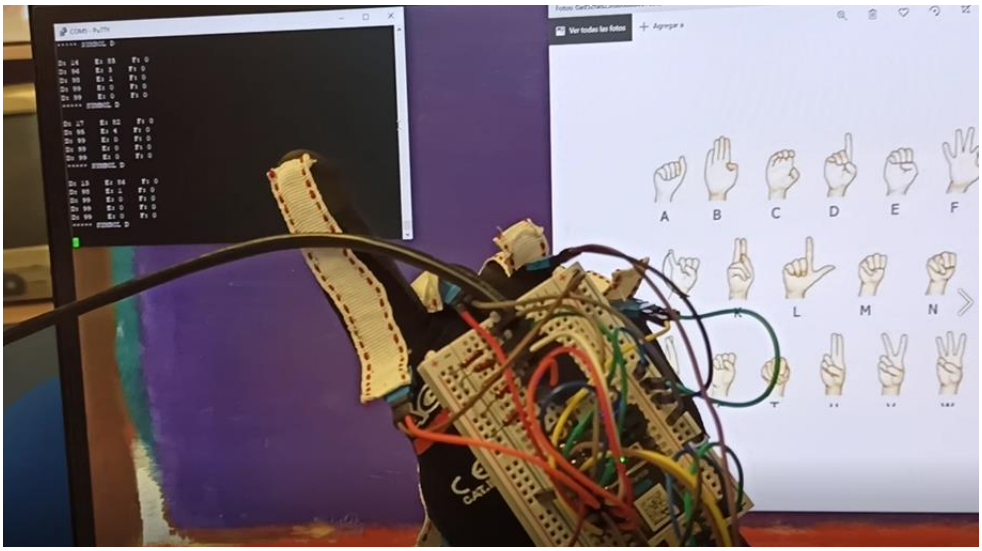


Como se puede observar, cuando el gesto realizado es identificado a partir de las inferencias identificadas por el modelo, superando el valor umbral (90%), éste es reconocido correctamente. Además, tras varias pruebas se ha comprobado que se reconocen a la perfección las tres letras seleccionadas.

Por último, tal y como se hizo en el capítulo anterior, el LED se encenderá de un color u otro en función del símbolo reconocido. El color verde está asociado a la letra D, el azul a la letra E, y el rojo a la letra F.

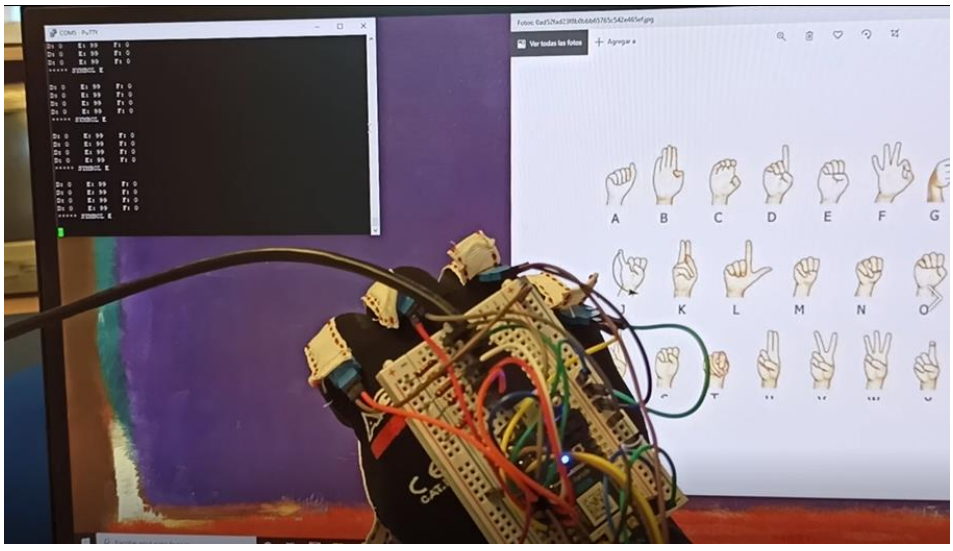
En la Figura 216 se puede observar la realización del primer gesto, la letra D, así como el correcto reconocimiento de dicho signo.

Figura 216. Gesto de la letra D



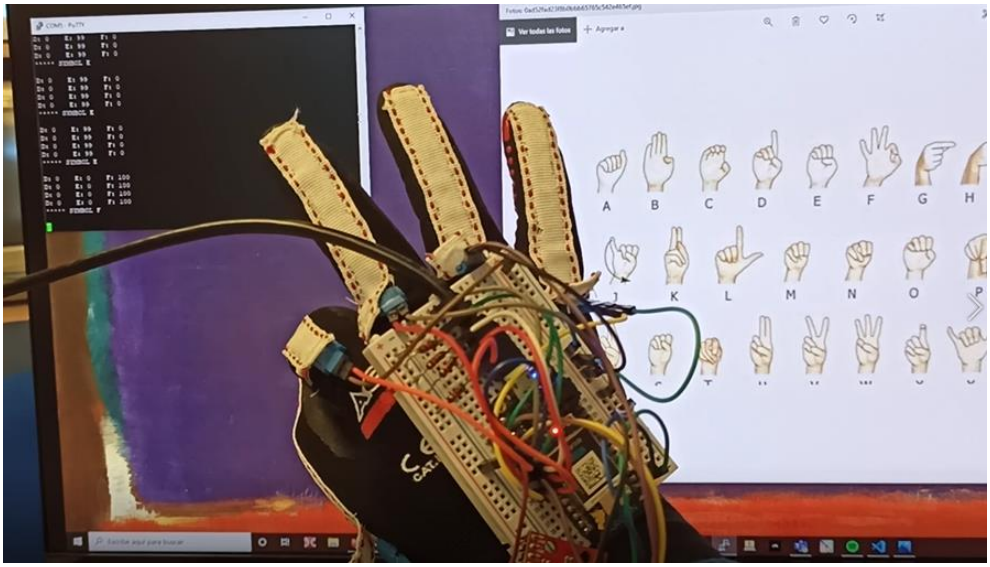
En la Figura 217 se puede observar la letra E correctamente reconocida.

Figura 217. Gesto de la letra E



Por último, se expone en la Figura 218 es el gesto asociado a la letra F y su reconocimiento experimental.

Figura 218. Gesto de la letra F



De esta manera se ha validado el correcto desarrollo del modelo generado, capaz de reconocer las letras D, E y F del alfabeto dactilológico de la lengua de signos española, así como su aplicación asociada, capaz de leer, interpretar y representar los resultados obtenidos.

Capítulo 6. Conclusiones

Llegados a este punto, se puede afirmar que los objetivos propuestos inicialmente en el desarrollo del presente TFG se han cumplido con éxito. De esta manera, se ha conseguido desarrollar un modelo TF para una aplicación específica, siendo en este caso, un modelo capaz de reconocer letras del alfabeto dactilológico de la lengua de signos española. Además, se ha desarrollado la aplicación que ha permitido implementar y validar el modelo sobre el dispositivo IoT Argon de *Particle*.

Para ello, en primer lugar, se ha realizado un estudio teórico para, por un lado, comprender el entorno de programación y funcionalidades de la plataforma Argon de *Particle*, y por otro, facilitar la comprensión del algoritmo de inteligencia artificial *Tensor Flow* y sus distintas funcionalidades, así como ciertas aplicaciones de referencia.

A continuación, se ha desarrollado un modelo TF inicial, capaz de reconocer tres gestos, mediante los datos de entrada capturados por el acelerómetro LSM9DS1 de la empresa *SparkFun Electronics*. Además, se ha implementado la aplicación con la que se ha conseguido validar dicho modelo implementándolo, a su vez, en la plataforma Argon de *Particle*. De esta manera, se han conseguido afianzar los conocimientos teóricos adquiridos previamente, facilitando el desarrollo del modelo final.

Por último, se ha desarrollado el modelo TF, capaz de reconocer letras del alfabeto dactilológico de la lengua de signos española, concretamente tres letras, ya que se han considerado suficientes para el propósito del presente TFG. Para ello, se han integrado, junto con el acelerómetro LSM9DS1, los cinco sensores *flex* correspondientes a los cinco dedos de la mano junto al acelerómetro y la plataforma Argon de *Particle* en un guante, facilitando la realización de los diferentes gestos. Además, tal y como se hizo en el paso anterior, se ha implementado una aplicación con el fin de validar dicho modelo.

Una vez se ha conseguido obtener la información necesaria para analizar el algoritmo de IA utilizado, a través de los distintos procesos detallados previamente, se ha llegado a la conclusión de que no es el idóneo para este tipo de aplicaciones. Es importante destacar que con el algoritmo *Tensor Flow*, concretamente *Tensor Flow Lite*, aplicado al reconocimiento de gestos se obtienen resultados válidos, pero ampliamente mejorables en comparación con otros algoritmos estudiados por el grupo de investigación. Esto se debe a dos razones fundamentales, por un lado, la **complejidad** del algoritmo aplicado a este campo, ya que durante el desarrollo del presente TFG han sido múltiples los problemas que se han tenido que solventar por falta de información o poca claridad al respecto. Por otro lado, cabe destacar que el **tamaño** del modelo generado es considerablemente elevado teniendo en cuenta que solo se reconocen tres letras del alfabeto dactilológico de la lengua de signos española, y es muy probable que si se hubieran incluido todas las letras no hubiera sido posible implementarlo en la plataforma, al menos en una versión no optimizada.

Con todo esto, se da por concluido el presente Trabajo de Fin de Grado, cumpliendo con todos los objetivos propuestos inicialmente.

BIBLIOGRAFÍA

[1] S. Greengard, *The Internet of Things (MIT Press Essential Knowledge series)*. Massachusetts, 2015.

[2] J. A. Pascual, "Inteligencia artificial: qué es, cómo funciona y para qué se está utilizando," *ComputerHoy.com*, 2019. [Online]. Available: https://computerhoy.com/reportajes/tecnologia/inteligencia-artificial-469917. [Accessed: 02-Feb-2020].

[3] A. M. Augusto, T. María Antonia, G. J. Luis, P. Adrián, and J. Matías, "Inteligencia Artificial de las Cosas," San Salvador de Jujuy, 2019.

[4] Tensorflow, "TensorFlow Lite | TensorFlow," *tensorflow.org*, 2019. [Online]. Available: https://www.tensorflow.org/lite/guide. [Accessed: 02-Feb-2020].

[5] Particle, "Particle Reference Documentation | Device OS API." [Online]. Available: https://docs.particle.io/reference/device-os/firmware/argon/. [Accessed: 03-Feb-2020].

[6] Particle, "Argon Datasheet," *docs.particle.io*. [Online]. Available: https://docs.particle.io/datasheets/wi-fi/argon-datasheet/. [Accessed: 18-Jul-2020].

[7] Sparkfun, "Acelerómetro LSM9DS1," *sparkfun.com*. [Online]. Available: https://www.sparkfun.com/products/13284.

[8] GitHub, "I2C," *github.com*. [Online]. Available: https://github.com/rambo/I2C.

[9] Sparkfun, "Sensor flexible," *sparkfun.com*. [Online]. Available: https://www.sparkfun.com/products/10264.

[10] Sparkfun, "Flex Sensor Hook Up Guide." [Online]. Available: https://learn.sparkfun.com/tutorials/flex-sensor-hookup-guide.

[11] Visual Studio, "Visual Studio Code," *visualstudio.com*. [Online]. Available: https://code.visualstudio.com/.

[12] Tensor Flow, "TensorFlow," *www.tensorflow.org*. [Online]. Available: https://www.tensorflow.org.

[13] Keras, "Keras," *keras.io*. [Online]. Available: https://keras.io/api/.

[14] TensorFlow, "TensorFlow Lite para microcontroladores," *tensorflow.org*. .

[15] C. Nicholson, "Convolutional Network," 2020. [Online]. Available: https://wiki.pathmind.com/convolutional-network.

[16] B. Dickens, "What are convolutional neural networks ?," *TechTalks*, 2020. [Online]. Available: https://bdtechtalks.com/2020/01/06/convolutional-neural-networks-cnn-convnets/.

[17] GitHub, "Librería NumPy Python," *github.com*. [Online]. Available: https://github.com/numpy/numpy.

[18] J. Torres, *Python Deep Learning. Introducción práctica con Keras y TensorFlow 2*, 1º Edición. 2020.

[19] P. Warden and D. Situnayake, *TinyML*, 1st Editio. O'Reilly, 2020.

[20] Arduino SA, "LSM9DS1 Libraries." [Online]. Available: https://github.com/sparkfun/LSM9DS1_Breakout/tree/master/Libraries.

[21] machinelearninguru.com, "Convolution Layer," 2020. [Online]. Available: https://www.machinelearninguru.com//computer_vision/basics/convolution/convolution_layer.html.

[22] Confederacion Estatal de Personas Sordas, "Lengua de Signos Española," *cnse.es*. [Online]. Available: https://www.cnse.es/index.php/lengua-de-signos.

[23] E. Cedeno, "Alfabeto dactilológico," *paperblog.es*, 2020. [Online]. Available: https://www.pinterest.es/pin/800233427522130580/.

[24] C. Rivero, "Plataforma para la interpretación del alfabeto dactilológico de la lengua de signos española basada en dispositivos IoT," Universidad de Las Palmas de Gran Canaria, 2018.

Pliego de condiciones

El Pliego de Condiciones expone las condiciones bajo las que se ha desarrollado el presente Trabajo Fin de Grado. Se exponen el conjunto de herramientas *hardware*, *software* y *firmware* empleadas durante su realización.

- **Condiciones Hardware**

En la Tabla 12 se recogen los dispositivos *hardware* utilizados.

Tabla 12. Condiciones hardware

Equipo	Modelo	Fabricante/Comerciante
Ordenador portátil	CPU Intel Core i7 8GB RAM	ASUS
Particle Argon	Argon	Particle
Acelerómetro LSM9DS1	SparkFun 9DoF IMU Breakout	Sparkfun Electronics
Flex sensor	Flex sensor 2.2”	Spectral Symbol

- **Condiciones Software**

Asimismo, se exponen en la Tabla 13 las herramientas *software* utilizadas, especificando su versión.

Tabla 13. Condiciones software

Aplicación	Versión
Sistema Operativo Windows	Microsoft Windows 10
Microsoft Office	Office 365
Microsoft Visio	2013
Microsoft Visual Studio Code	1.56.2
Notepad++	7.8.7
Putty	0.74.0.0
Plataforma Tensor Flow	2.2.0
Particle Workbench	1.13.8
Python	3.8.1

- **Condiciones Firmware**

Por último, en la Tabla 14 se puede ver el *firmware* utilizado, así como su versión.

Tabla 14. Condiciones firmware

Aplicación	Versión
Firmware del dispositivo Argon	1.5.2
Librería Arduino_LSM9DS1	1.0.0
Librería TensorFlowLite	0.1.1

Presupuesto

En este capítulo se recogen los gastos generados en la realización del presente Trabajo Fin de Grado con el objetivo de obtener un presupuesto. Éste se en las siguientes partes:

- Trabajo tarifado por tiempo empleado.
- Amortización del inmovilizado material, dividida en:
    - Amortización del material *hardware.*
    - Amortización del material *software.*
- Redacción de la documentación.
- Derechos de visado del COITT (Colegio Oficial de Ingenieros Técnicos de Telecomunicación).
- Gastos de tramitación y envío.

Por último, se aplicarán los impuestos vigentes para obtener el coste total del TFG.

- **Trabajo tarifado por tiempo empleado**

Este concepto contabiliza los gastos correspondientes a la mano de obra, en función del salario correspondiente a la hora de trabajo de un Ingeniero Técnico de Telecomunicación.

Para calcular el total, se ha tenido en cuenta la tabla retributiva de personal contratado en proyectos de investigación, elaborada por la ULPGC en el año 2019. Para una dedicación de 20 horas semanales, el salario asciende a 711,90 €**.** Además, es importante saber que la duración aproximada del TFG es de 4 meses, por ello el cálculo del coste total se realiza por tiempo empleado.

$$711{,}90 \times 4 = 2847{,}60$$

Por lo tanto, el trabajo tarifado por tiempo empleado asciende a la cantidad de *dos mil ochocientos cuarenta y siete euros con sesenta céntimos.*

- **Amortización del inmovilizado material**

En el inmovilizado material se consideran tanto los recursos *hardware* como los recursos *software* empleados en la realización del presente TFG.

Para estipular el coste de amortización en un periodo de 3 años se utiliza un sistema de amortización lineal, donde el inmovilizado material se desprecia de forma constante a lo largo de su vida útil. Por otro lado, la cuota de amortización anual se calcula con la Fórmula 2 que se expone a continuación.

Fórmula 2. Cuota anual

$$Cuota\ anual = \frac{Valor\ de\ adquisición - Valor\ residual}{Número\ de\ años\ de\ vida\ útil}$$

donde el valor residual se corresponde con el valor teórico que se supone que tendrá el elemento después de su vida útil.

- Amortización del material *hardware*

Tal y como se comentó previamente, la duración efectiva de este proyecto ha sido de 4 meses, cifra considerablemente inferior a los 3 años estipulados para el coste de amortización, es por ello que dichos costes se calcularán en base a los derivados de los primeros 4 meses.

En la Tabla 15 se especifica el *hardware* amortizable necesario para la realización del trabajo, indicando su valor de adquisición y su amortización, teniendo en cuenta un tiempo de uso de 4 meses.

Tabla 15. Hardware amortizable

Elemento	Valor de adquisición	Amortización
Ordenador portátil Asus	1120,00 €	123,00€
Argon	22,87 €	22,87 €
Acelerómetro LSM9DS1	12,75 €	12,75 €
Flex sensor 2.2” (x5)	34,12 €	34,12 €
Total	**1189,74 €**	**192,74 €**

Tal y como se puede observar, excepto en el primer elemento, el valor de adquisición coincide con su amortización debido al bajo coste de éstos, obteniendo un coste total del material hardware empleado de *ciento noventa y dos euros con setenta y cuatro céntimos.*

- Amortización del material *software*

Para el cálculo de los costes de amortización del material *software* se considerarán, al igual que con el material *hardware*, los costes derivados de los primeros 4 meses.

En la Tabla 16 se exponen los elementos *software* necesarios, así como su valor de adquisición y amortización.

Tabla 16. Elementos software

Elemento	Valor de adquisición	Amortización
Sistema operativo Windows	0,00 **€**	0,00 €
Microsoft Office	0,00 € (*)	0,00 € (*)
Microsoft Visio	0,00 € (*)	0,00 € (*)
NotePad++	0,00 € (**)	0,00 € (**)
Putty	0,00 € (**)	0,00 € (**)
Particle Workbench	0,00 € (**)	0,00 € (**)
Visual Studio Code	0,00 € (**)	0,00 € (**)
Plataforma Tensor Flow	0,00 € (**)	0,00 € (**)
Python	0,00 € (**)	0,00 € (**)
Total	**0,00 €**	**0,00 €**

(*) Licencia de uso proporcionada por la ULPGC.
(**) Software libre.

Por tanto, el coste total del material *software* es de *cero euros*.

- **Redacción del trabajo**

Para determinar el coste asociado a la redacción de la memoria del presente Trabajo Fin de Grado se ha utilizado la Fórmula 3.

Fórmula 3. Coste de la redacción

$$R = 0{,}07 \times P \times C_n,$$

donde:

- **R** son los honorarios por la redacción.
- **P** es el presupuesto.
- $\boldsymbol{C_n}$ es el coeficiente de ponderación en función del presupuesto.

Asimismo, el valor del presupuesto P es la suma de los costes del trabajo tarifado por el tiempo empleado, así como de la amortización del inmovilizado material, tal y como se muestra en la Tabla 17.

Tabla 17. Valor del presupuesto (P)

Concepto	Coste
Trabajo tarifado por tiempo empleado	2847,60
Amortización del inmovilizado material	192,74 + 0,00
Total	**3040,34 €**

Además, se ha de tener en cuenta que el coeficiente de ponderación $\boldsymbol{C_n}$ para presupuestos menores de 30.050,00 € viene definido por el COITT con un valor de 1.00. Por ello, el coste derivado de la reacción del TFG es:

$$R = 0{,}07 \times P \times 1 = 0{,}07 \times 3040{,}34 \times 1 = 212{,}82\ €$$

De esta forma, el oste de la redacción del trabajo asciende a *doscientos doce euros con ochenta y dos céntimos.*

- **Derechos de visado del COITT**

El COITT establece que, para proyectos técnicos de carácter general, los derechos de visado para 2016 se calculan en base a la Fórmula 4.

Fórmula 4. Coste del derecho de visado

$$V = 0{,}006 \times P_1 \times C_1 + 0{,}003 \times P_2 \times C_2,$$

donde:

- V es el coste de visado del trabajo.
- P_1 es el presupuesto del proyecto.
- C_1 es el coeficiente reductor en función del presupuesto.
- P_2 es el presupuesto de ejecución material correspondiente a la obra civil.
- C_2 es el coeficiente reductor en función a P_2.

El presupuesto P_1 se calcula sumando los costes de las secciones correspondientes al trabajo tarifado por tiempo empleado, a la amortización del inmovilizado material y a la redacción del documento, tal y como se muestra en la Tabla 18. Además, el coeficiente C_1 para proyectos de presupuesto inferior a 30.050,00 € es de 1,00, y el valor de P_2 es de 0,00 € ya que no se realiza ninguna obra.

Tabla 18. Valor del presupuesto (P1)

Concepto	Coste
Trabajo tarifado por tiempo empleado	2847,60 €
Amortización del material hardware	192,74 €
Amortización del material software	0,00 €
Redacción del trabajo	212,82 €
Total	**3253,16 €**

De esta manera, aplicando a la fórmula expuesta los datos de la tabla y el coeficiente especificado se obtiene:

$$V = 0{,}006 \times P_1 \times C_1 + 0{,}003 \times P_2 \times C_2 = 0{,}006 \times 3253{,}16 \times 1 = 19{,}52€$$

Los costes por derechos de visado del presupuesto ascienden a *diecinueve euros con cincuenta y dos céntimos.*

- **Gastos de tramitación y envío**

Los gastos de tramitación y envío están estipulados en 6,00 € por cada documento visado de forma telemática.

- **Material fungible**

Además de los recursos *hardware* y *software*, se han empleado otros materiales considerados fungibles, como se muestra en la Tabla 19.

Tabla 19. Material fungible

Concepto	Coste
Folios	10,00 €
Tóner de la impresora	30,00 €
Guante	5,00 €
Total	**45,00 €**

- **Aplicación de impuestos y coste total**

La realización del presente TFG está gravada por el Impuesto General Indirecto Canario (IGIC) en un 7%. En la Tabla 20 se muestra el presupuesto final con dichos impuestos aplicados.

Tabla 20. Coste total

Concepto	Coste
Trabajo tarifado por tiempo empleado	2847,60 €
Amortización del material hardware	192,74 €
Amortización del material software	0,00 €
Redacción del trabajo	212,82 €
Derechos de visado del COITT	19,52 €
Gastos de tramitación y envíos	6,00 €
Costes de material fungible	45,00 €
Total (Sin IGIC)	3323,68 €
IGIC	232,65 €
Total	**3556,33 €**

El presupuesto total del presente Trabajo Fin de Grado "**Desarrollo de una plataforma para reconocimiento de gestos basada en Tensor Flow Lite sobre el dispositivo IoT Argon de Particle"** es de *tres mil quinientos cincuenta y seis euros con treinta y tres céntimos.*

En Las Palmas de Gran Canaria, a 6 de septiembre de 2021.

Fdo. D. Tomás Real Rueda