Universidad de Alcalá

Escuela Politécnica Superior

Grado en Ingeniería Electrónica de Comunicaciones

Trabajo Fin de Grado

Clasificación de señales de audio con microcontroladores aplicando inteligencia artificial

Autor: Carlos García Gómez

Tutor: José Manuel Villadangos Carrizo

2023

UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior

GRADO EN INGENIERÍA ELECTRÓNICA DE COMUNICACIONES

Trabajo Fin de Grado

Clasificación de señales de audio con microcontroladores aplicando inteligencia artificial

Autor: Carlos García Gómez

Tutor: José Manuel Villadangos Carrizo

TRIBUNAL:

Presidente: Pedro Martín Sánchez

Vocal 1º: Ana Jiménez Martín

Vocal 2º: José Manuel Villadangos Carrizo

FECHA: SEPTIEMBRE 2023

ÍNDICE

Índice de Figuras

Índice de Tablas

RESUMEN

Este Trabajo Fin de Grado versa sobre como ejecutar una Red Neuronal de reconocimiento de señales de audio programada en Python en un microcontrolador de ST Microelectronics.

Comienza con el Marco Teórico explicando los conceptos esenciales para trabajar con Redes Neuronales para continuar con una descripción del hardware empleado. Se continúa explicando el código programado para diseñar la Red Neuronal en Python y se ahonda en la configuración y validación de dicho código para convertirlo en lenguaje C entendible por el microcontrolador.

Por último, se desgrana y se explica el código en C diseñado que ejecuta el microcontrolador.

ABSTRACT

This Final Degree Project focuses on how to run a Neural Network for audio signal recognition programmed in Python on an ST Microelectronics´s microcontroller.

It begins with the Theoretical Framework, explaining the essential concepts to work with Neural Networks, followed by a hardware description. It continues explaining the code programmed used to design the neural network in Python and delves into the configuration and validation of this code to make it understandable by the microcontroller in C language.

Finally, the designed C code that runs on the microcontroller is broken down and explained.

RESUMEN EXTENDIDO

Las redes neuronales están cada vez más presentes en el día a día. Son un componente esencial en *machine learning* y la inteligencia artificial. Al conseguir el aprendizaje automático se consiguen importantes avances en materias como la visión artificial, el reconocimiento de voz, toma de decisiones autónomas, etc.

A todo esto, hay que sumar la creciente tendencia al manejo de grandes cantidades de datos, con el aumento de uso de sensores y los dispositivos IoT. Para el manejo de estos datos en tiempo real y evitar latencias en el procesamiento de los mismos en la nube y la ejecución de algoritmos complejos se hace necesario la incorporación de potentes y versátiles microcontroladores como son los microcontroladores de ST Microelectronics.

Por todo esto, ante el cambio tecnológico presente en nuestros días, se crea una sinergia entre las redes neuronales y los microcontroladores que confluye en el diseño de dispositivos modernos y versátiles que hacen el día a día más sencillo a las personas, además de impulsar la industria hacia un camino más moderno y eficiente.

Ante este reto, este Trabajo Fin de Grado, trata de facilitar el contacto entre una persona que desea iniciarse en el mundo de las redes neuronales y combinarlo con los microcontroladores más utilizados en el mercado como son los de ST Microelectronics.

Para ello, se pretende facilitar este salto contando los pasos que se siguen en el desarrollo de una Red Neuronal programada en Python para la clasificación de señales de audio y su posterior programación en un microcontrolador de ST.

El Trabajo Fin de Grado comienza explicando una serie de conceptos teóricos fundamentales para comprender el funcionamiento de las Redes Neuronales y que tienen relación con el trabajo desarrollado. Los conceptos que se tratan de explicar son:

- Red neuronal.
- Funciones de activación.
- Entrenamiento de una red neuronal.
- Asignación de pesos en redes.
- Algoritmo *backpropagation*.
- Descenso del gradiente.
- *Overfitting.*
- *Underfitting.*
- Transforma de Fourier.
- Espectrograma y escala Mel.

En el siguiente capítulo se detalla el hardware empleado para ejecutar la Red Neuronal. Se proporciona información sobre el diagrama de bloques de la PCB STEVAL-STLCS01V1, así como su diseño esquemático.

Se detalla la información del microcontrolador de 32-bit STM32L476 MCU ARM Cortex-M4 y se hace hincapié en los sensores y periféricos empleados para llevar a cabo la clasificación de escenas de audio.

El audio será capturado por el micrófono disponible en la PCB STEVAL-STLCS01V1, por ello se proporciona información en detalle sobre este micrófono y se continúa ahondando en el proceso de conversión de audio PDM a PCM. Por ello se detalla el funcionamiento del periférico empleado que está dentro del microcontrolador y es el DFSDM (*Digital Filter Sigma Delta Modulator*), un periférico muy importante gracias a sus opciones de configuración y la conversión que realiza de audio PDM a PCM. Para realizar esta tarea se emplea un *buffer* circular que libera de parte de procesamiento al microcontrolador.

El Trabajo Fin de Grado continúa explicando el código realizado en Python para elaborar la Red Neuronal que hace la clasificación de las señales de audio. La elaboración de este código se apoya en Keras y TensorFlow. Keras es una biblioteca de código abierto utilizada para el diseño de redes neuronales ya que proporciona bloques modulares sobre los que se pueden desarrollar modelos complejos de aprendizaje profundo y TensorFlow permite al desarrollador abstraerse de la complejidad del diseño facilitando tareas como el entrenamiento de la red así como proporcionar modelos y algoritmos.

El script de Python diseñado pretende realizar una clasificación a través de los audios proporcionados diferenciando el estado de un ventilador. Los estados son:

- Apagado
- Encendido.
- Obstruido.

El script se va mostrando en pequeños fragmentos de código y se van ejecutando paso a paso sobre Jupyter Notebook. Los pasos del script de Python son los siguientes:

- Importación de librerías y carga de *dataset*.
- Preparación de los datos.
- Cortar datos en *frames*.
- Preprocesamiento de datos para crear los espectrogramas LogMel.
- Estandarización de características.
- Preparación de los datos de salida.
- Separación del *dataset* en un conjunto de datos de entrenamiento, validación y test.
- Salvar las características.
- Construcción del modelo.
- Compilación del modelo.
- Entrenamiento del Modelo.
- Evaluar la precisión.
- Matriz de confusión.
- Salvar el modelo.

Cada paso que realiza el código se va explicando y se trata de clarificar las operaciones que realiza con explicaciones adicionales.

Una vez ejecutado el script en Python, el Trabajo Fin de Grado continúa explicando el proceso de validación y configuración realizado con la herramienta STMCubeIde gracias a la librería X-CUBE-AI.

Antes de cargar el código en el microcontrolador hay que realizar 3 análisis que nos dan información del error cometido en la conversión de código C a código entendible por el microcontrolador y del espacio que ocupa en memoria para ver si cabe o es necesario realizar una compresión o una optimización del código en Python.

Se explica paso a paso la configuración el proceso de validación, tanto en la selección de pines del microcontrolador como el cambio que hay que hacer en el código para que funcione la validación por USB.

Una vez vista la validación se pasa a la configuración de los periféricos del microcontrolador a utilizar. Se detalla toda la selecciones de pines, la configuración del DFSDM, del reloj, de las interrupciones y los cálculos asociados para generar el código final.

Respecto al código final, se explica el árbol de carpetas del proyecto y se van comentando los ficheros más importantes que intervienen en el código. Durante la explicación se va desgranando las líneas de código para explicar las declaraciones de variables y funciones. Se va explicando poco a poco los diferentes fragmentos de

código para que el lector entienda el código realizado y su flujo de procesamiento. Se hace especial hincapié en los archivos main.c y asc_processing.c.

En el penúltimo capítulo se realiza el presupuesto del proyecto, realizando un desglose de los softwares, hardware y de ejecución final.

Para finalizar se exponen las conclusiones y las líneas futuras de trabajo.

ACRÓNIMOS

ADC: Analog to Digital Converter.

APB: Advance Peripheral Bus.

AHB: AMBA High-performance Bus.

ASC: Audio Scene Clasification.

CAN: Controller Area Network.

CDC: Communication Device Class.

CMOS: Complementary Metal–Oxide–Semiconductor.

CPU: Core Processor Unit.

DAC: Digital to Analog Conversor.

DFSDM: Digital Filter Sigma Delta Modulator.

DMA: Direct Memory Access.

ESD: Electrostatic discharge.

FAT: File Allocation Table.

FFT: Fourier Fast Transform.

GPIO: General Purpose Input/Output.

GPU: Graphics Processor Unit.

HAL: Hardware Abstraction Layer.

I2C: Inter-Integrated Circuit.

iNEMO: Inertial Modules.

LCD: Liquid-Crystal Display.

LDO: Low-Dropout Regulator.

L2R : L2 Relative Error.

MCU: Microcontroller.

MEMS: Microelectromechanical Systems.

MPU: Multiple Process Unit.

MSE: Mean Squared Error.

NVIC: Nested vector Interrupt Control.

ONEIROS: Open-Ended Neuro-Electronic Intelligent Robot Operating System.

OPAMP: Operational Amplifier.

PCM: Pulse Code Modulation.

PDM: Pulse Digital Modulation.

RAM: Random-Access Memory.

ReLu: Rectifier Linear Unit.

RTC: Real Time Clock.

SDMMC: Secure Digital/ MultiMediaCard.

SMPS: Switch Modes Power Supplies.

SPI: Serial Peripheral Interface.

SRAM: Static Random Access Memory.

SWPMI: Single Wire Protocol Master Interface.

RTOS: Real Time Operating System.

TCP/IP: Transmission Control Protocol/Internet Protocol.

UART: Universal Asynchronous Receiver-Transmitter.

USART: Universal Synchronous and Asynchronous Receiver-Transmitter.

USB: Universal Serial Bus.

USB OTG: Universal Serial Bus On The Go.

PALABRAS CLAVE

Palabras clave: red neuronal, espectrograma, escala Mel, entrenamiento y DFSDM (*Digital Filter Sigma Delta Modulator*).

1. INTRODUCCIÓN

En este Trabajo Fin de Grado se pretende explicar y documentar todo el proceso de diseño de un sistema con inteligencia artificial que realiza la clasificación de señales de audio, desde su programación en Python hasta su implementación en código C entendible por el microcontrolador. En concreto, el diseño trata de resolver el estado (apagado, encendido u obstruido) en que se encuentra un ventilador mediante una red neuronal implementada en un microcontrolador de STM32.

El objetivo es que este Trabajo Fin de Grado pueda servir como una guía para toda persona que se inicie en este campo y le sirva de apoyo para entender todo el proceso de diseño, así como completar este proyecto con nuevas funcionalidades o desarrollar otros cuya base de trabajo sea muy parecida a esta.

El Trabajo Fin de Grado se estructura de la siguiente manera:

- El segundo capítulo se dedica al Marco Teórico, en el cual, se explican los principales conceptos imprescindibles para entender el trabajo que hay detrás de una red neuronal y el trabajo desarrollado en este Trabajo Fin de Grado.
- En el tercer capítulo se hace una descripción del hardware utilizado y de los periféricos del microcontrolador empleados.
- En el cuarto capítulo se explica el script de Python donde se diseña la red neuronal para realizar la clasificación de señales de audio y que es utilizado para hacer la conversión a lenguaje C entendible por el microcontrolador.
- En el quinto capítulo se realiza la explicación del proceso de transformación y validación del código de Python en código C a través del software STM32CubeMx y también se explica la configuración de los periféricos realizada a través del STM32CubeMx.
- En el sexto capítulo se explica el código diseñado en C, el cual implementa la red neuronal diseñada en Python y hace la selección de escenas de audio, así como la comunicación con un PC a través de USB para que el usuario final vea la detección de los diferentes estados de la red.
- En el séptimo capítulo se desglosa el presupuesto del Trabajo Fin de Grado.

- En el octavo capítulo se dan a conocer las conclusiones y líneas de trabajo futuras tomando como base este Trabajo Fin de Grado.

2. MARCO TEÓRICO

2.1. Introducción

En este marco teórico se van a tratar de exponer los principales conceptos que hay que conocer para entender mejor el trabajo llevado a cabo en este Trabajo Fin de Grado.

La **inteligencia artificial** son los intentos de replicar la inteligencia humana en sistemas artificiales.

Machine learning son las técnicas de aprendizaje automático, en donde un mismo sistema aprende como encontrar una respuesta sin que alguien lo esté programando.

Deep learning es todo lo relacionado a las redes neuronales. Se llama aprendizaje profundo porque a mayor número de capas conectadas entre sí se obtiene un aprendizaje más fino.

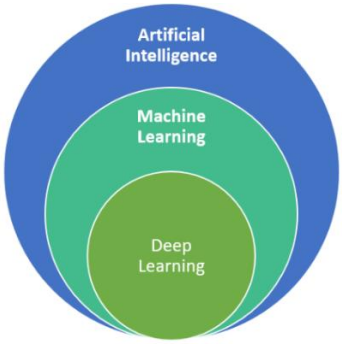


Figura 1. Inteligencia artificial [1]

Red Neuronal: es un modelo computacional basado en un conjunto de unidades neuronales simples, cada una con una regla o una condición diferente. A dicha condición se le llama "función de activación", y ésta se debe cumplir antes de propagarse la información a través del resto de unidades neuronales. Cada unidad neuronal o neurona, está conectada con otras a través de enlaces, que a su vez tienen asociados unos pesos que, activan, aumentan, disminuyen o desactivan el valor de cada neurona de la red.

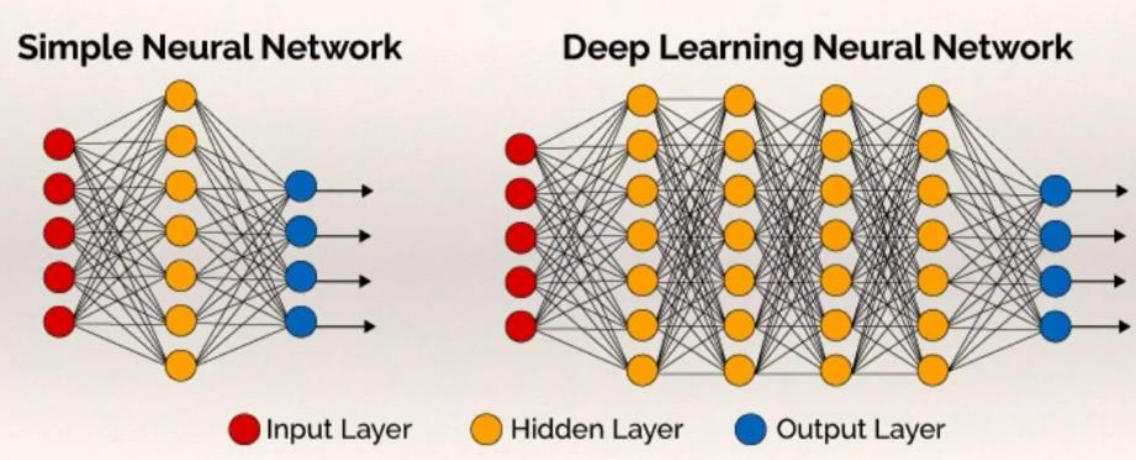


Figura 2. Red neuronal[2]

Dentro de una red neuronal, existen capas, que son conjuntos de neuronas por los cuales pasa la información que se desea clasificar o usar en una regresión. Dichas capas están siempre ocultas en cuanto a que, normalmente, no se ve el resultado del tratamiento de información en las mismas; sólo hay dos capas no ocultas, la capa de entrada de datos, y la de salida.

El propósito de las Redes Neuronales es que tienen la capacidad de aprender por si solas. Dado un conjunto de entrenamiento, del cual conocemos los resultados, se introduce en la red con un método *backward* (hacia atrás), y de esta manera, las funciones de activación y los pesos, van cambiando hasta adaptarse al conjunto conocido y después ser usado en un conjunto real. Este método se basa en la "función de pérdida", que evalúa la diferencia entre el valor de salida de la red y el valor real del conjunto de entrenamiento. Cambiando el valor de los pesos en los enlaces entre neuronas y el sesgo en cada neurona, se intenta minimizar el valor de la función pérdida.

Existen diferentes tipos de Redes Neuronales, en este Trabajo Fin de Grado, se utilizará una **red neuronal convolucional**. Este tipo de red, como su propio nombre indica, utiliza la operación de convolución como base para el procesamiento de los datos. Dentro de esta red, existen matrices llamadas filtros y cada una de ellas detecta, a su vez, distintos tipos de características dentro de los datos que se quieren procesar.

Este tipo de redes se usan principalmente para el procesamiento de imágenes, con lo cual, dichos filtros serán capaces de detectar, por ejemplo, bordes. Además, estos filtros se moverán por la imagen original según un parámetro que mide la longitud del salto *(stride*). Para reducir la dimensión de los datos a procesar, estas redes utilizan otra operación llamada *Poolling*, que tiene diferentes formas de implementación.

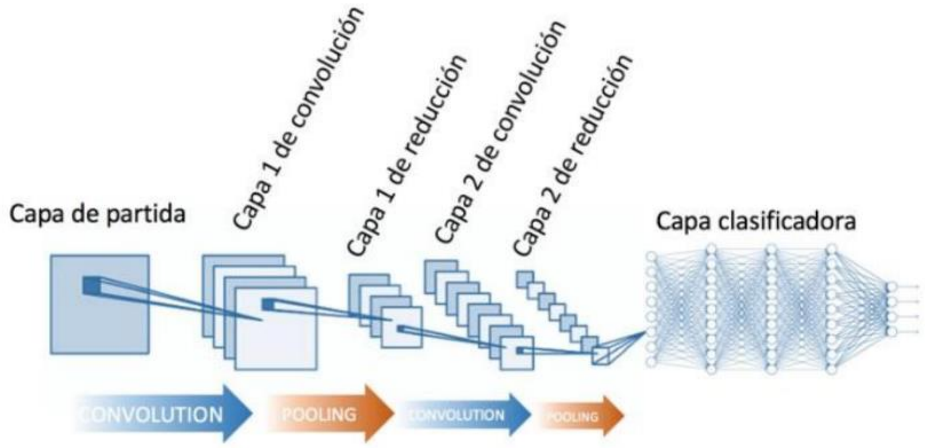


Figura 3. Red neuronal convolucional [3]

Cada neurona se encarga de recibir unos datos, de los que se conoce su comportamiento y así clasificarlos de forma binaria, creando un modelo que pueda ser utilizado para datos futuros de los que no se conoce su comportamiento. Posteriormente, las neuronas van trasladando la información entre sí.

Cada neurona o capa de la Red procesa un tipo o un conjunto diferente de atributos de los datos de entrenamiento, que, puestos en común con las demás neuronas, son capaces de clasificar los datos en sí.

El propósito es, crear un modelo a partir de datos conocidos para después utilizarlo con datos de los que se desconoce su comportamiento.

Dicho modelo cambia según los datos de entrada y puede ser modificado para ajustarse lo mejor posible a nuestro conjunto.

En una Red Neuronal, cada neurona, internamente, tiene un funcionamiento para clasificar los datos recibidos. La neurona recibe los valores de entrada multiplicados por sus respectivos pesos, provenientes de otras neuronas, que será la importancia que tendrá dicha variable en el aprendizaje de la Red Neuronal. Además, existe un término independiente, dicho término, en la neurona, viene dado como una variable más de entrada, asociada siempre a la variable 1, y se le llama Sesgo. El sesgo podremos ajustarlo a gusto modificando el peso asociado a él, para un entrenamiento más preciso de la Red Neuronal.

Se puede ver fácilmente que el efecto de concatenar muchas operaciones lineales o sumar muchas líneas rectas, equivale a solamente haber hecho una línea recta, una única operación de regresión lineal. Ese es un problema que nos limitaría una Red Neuronal, porque no serviría de mucho aplicar las diferentes capas a nuestros datos.

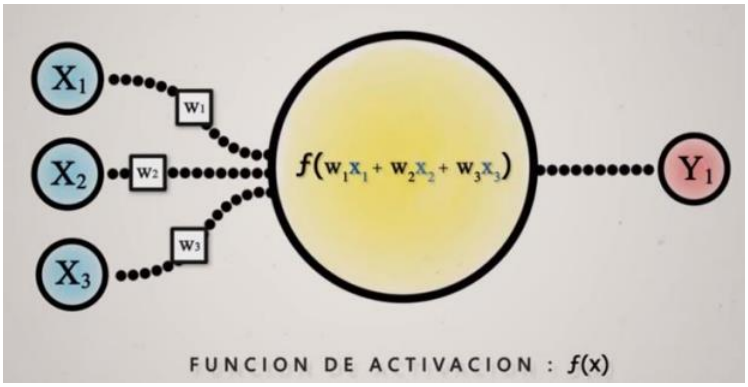


Figura 4. Funciones de activación [2]

En este momento es donde entra en juego la función de activación, que, en pocas palabras, deslinealiza la salida de cada neurona para que la concatenación de capas tenga sentido al introducir la no linealidad que nos permite que la Red pueda aproximar cualquier función.

2.2. Funciones de activación

Las funciones de activación solucionan el colapso de las linealidades de las capas de la red neuronal. En general, las funciones de activación se utilizan para dar una «no linealidad» al modelo y que la red sea capaz de resolver problemas más complejos. Si todas las funciones de activación fueran lineales, la red resultante sería equivalente a una red sin capas ocultas.

Hay muchos tipos diferentes de funciones de activación, a continuación, se muestran las principales:

Función escalón (*threshold*): Como se puede ver en la figura de abajo, en el eje x está representado el valor ya ponderado de las entradas y sus respectivos pesos, y en el eje y se tiene el valor de la función escalón. Esta función propaga un 0 si el valor de x es negativo, o un 1 si es positivo. No hay casos intermedios, y sirve para clasificar de forma muy estricta. Es la función más rígida.

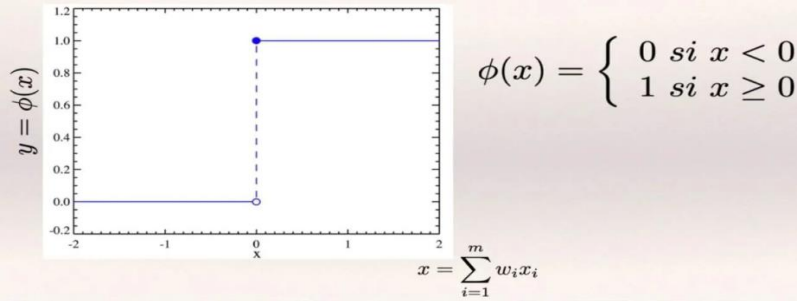


Figura 5. Función escalón [2]

Función sigmoide: Como en la función escalón, existe una división entre los valores negativos y positivos de x, pero la función sigmoide no es tan estricta, el cambio se hace de manera suave. Se usa en la regresión logística, una de las técnicas más usadas en *Machine Learning*. Como se ve la figura de abajo, la función no tiene aristas, es más suave (en términos matemáticos, es derivable). Cuanto más positivo es el valor de x más nos acercamos a 1 y por el contrario, cuanto más negativo es x más nos acercamos a 0.

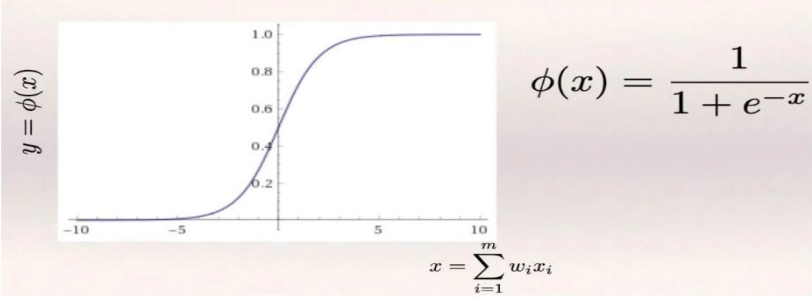


Figura 6. Función sigmoide [2]

Función softmax: Esta función es muy útil en la capa final de salida al final de la red neuronal, no solo para clasificar con valores categóricos, sino también para intentar predecir las probabilidades de pertenencia a cada categoría, donde se sabe que la probabilidad de un suceso imposible es 0 y la de un suceso seguro es 1.

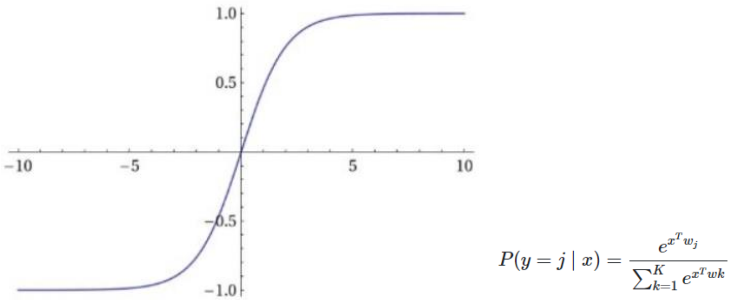


Figura 7. Función softmax [4]

Función rectificadora (ReLU): En la figura de abajo se aprecia que su valor es 0 para cualquier valor negativo de x. Si x es positivo la función retorna el propio valor de x. Por lo tanto, se obvian todas aquellas entradas que una vez ponderadas tienen un valor negativo y proporciona el valor exacto de las positivas. Se ve también que, al contrario de las anteriores, los valores no se restringen a valores entre 0 y 1.

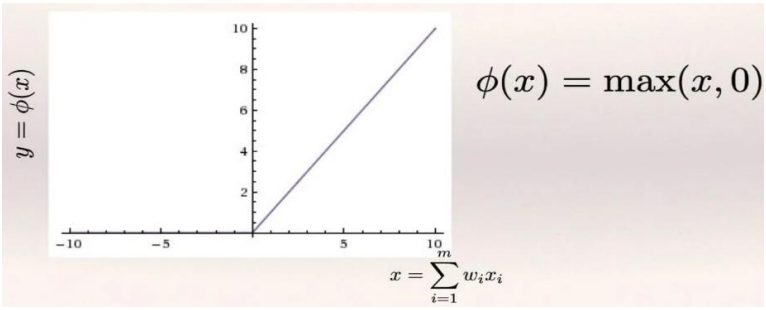


Figura 8. Función rectificadora (Relu) [2]

Función tangente hiperbólica: Esta función es muy parecida a la función sigmoidea, pero en este caso, como se ve en la figura de abajo, no empieza en 0 y termina en 1, sino que empieza en -1 y termina en 1.

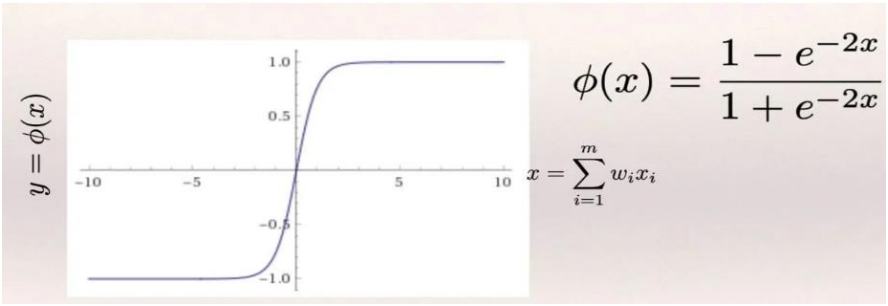


Figura 9. Función tangente hiperbólica [2]

2.3. Ejemplo de redes convolucionales

Se va a proceder a explicar con un ejemplo que ayude a comprender mejor el funcionamiento de las redes convolucionales.

Como su propio nombre indica, las redes neuronales convolucionales se caracterizan por las convoluciones. Éstas consisten en tomar "**grupos de pixeles cercanos"** de la imagen de entrada e ir operando matemáticamente (producto escalar) contra una pequeña matriz que se llama *kernel.* Se supone un *kernel* de tamaño 3×3 *pixels* que "recorre" todas las neuronas de entrada (de izquierda-derecha, de arriba-abajo) y genera una nueva matriz de salida, que en definitiva será nuestra nueva capa de neuronas ocultas.

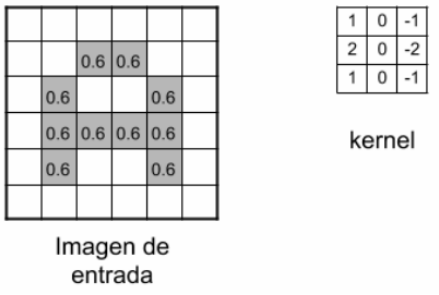


Figura 10. Imagen de entrada y kernel [5]

El *kernel* tomará inicialmente valores aleatorios y se irán ajustando mediante *backpropagation*.

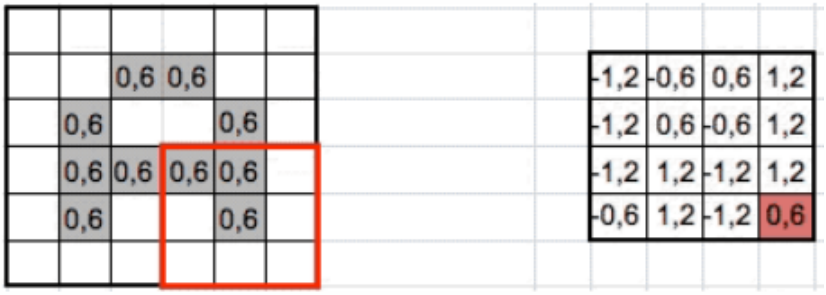


Figura 11. Producto matricial [5]

En la figura anterior se ve el *kernel* realizando el producto matricial con la imagen de entrada y desplazando de a 1 *pixel* de izquierda a derecha y de arriba-abajo y se va generando una nueva matriz que compone al mapa de características (*features*).

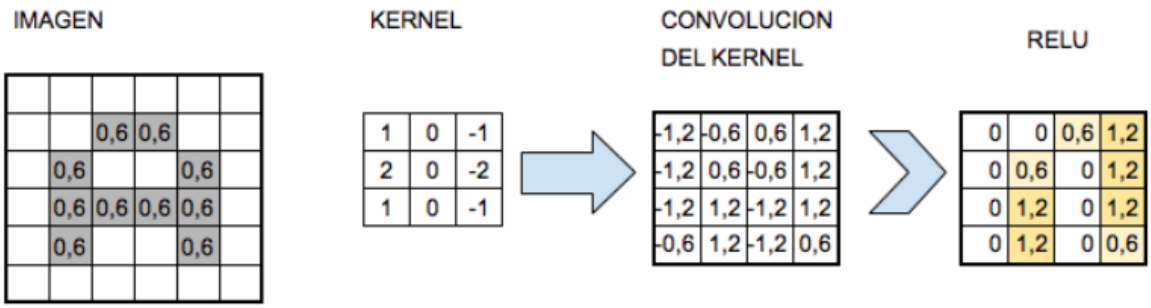


Figura 12. Convolución con kernel y función de activación ReLu [5]

La función de activación más utilizada para este tipo de redes neuronales es la llamada ReLu por *Rectifier Linear Unit* y consiste en f(x)=max(0,x).

El siguiente paso es reducir la cantidad de neuronas antes de hacer una nueva convolución para reducir la cantidad de procesamiento asociado. Por ello, para reducir el tamaño de la próxima capa de neuronas se hace un proceso de *subsampling* en el que se reduce el tamaño de las imágenes filtradas, pero en donde **deberán prevalecer las características más importantes que detectó cada filtro**. Uno de los tipos de *subsampling* más usado es el *Max-Pooling.*

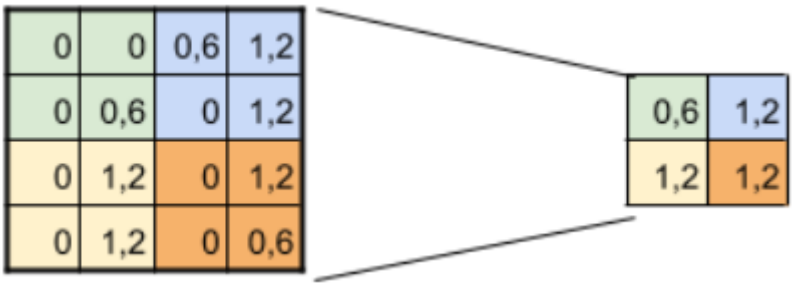


Figura 13. Subsampling con Max-Pooling [5]

Para entender mejor este concepto se explica con un ejemplo: si se supone que se hace un *Max-pooling* de tamaño 2×2. Esto quiere decir que se recorre cada una de las 32 imágenes de características obtenidas anteriormente de 28x28 píxeles de izquierda-derecha, arriba-abajo, pero en vez de tomar de tamaño un píxel, se toma de "2×2" (2 de alto por 2 de ancho = 4 píxeles) y se irá preservando el valor "más alto" de entre esos 4 píxeles (por eso lo de "Max"). En este caso, usando 2×2, la imagen resultante es reducida "a la mitad" y quedará de 14×14 píxeles. Luego de este proceso de *subsamplig* quedarán 32 imágenes de 14×14, pasando de haber tenido 25.088 neuronas a 6272, son bastantes menos y en teoría siguen almacenando la información más importante para detectar características deseadas.

Se ahonda un poco más en estos conceptos con otro ejemplo para ver el funcionamiento de una red convolucional.

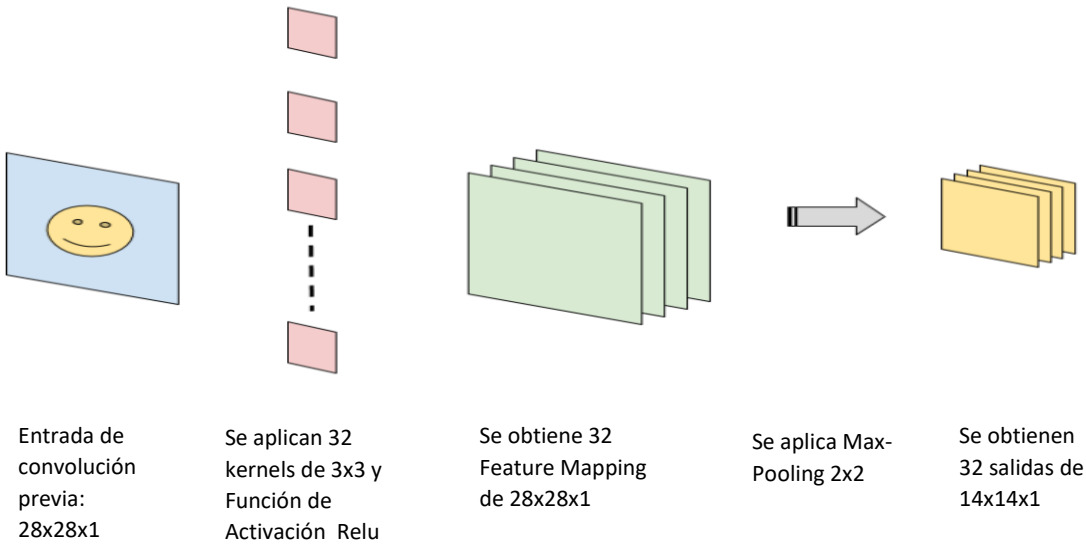


Figura 14. Primera convolución [5]

La primera convolución es capaz de detectar características primitivas como líneas o curvas. **A medida que se hagan más capas con las convoluciones, los mapas de características serán capaces de reconocer formas más complejas**, y el conjunto total de capas de convoluciones podrá "ver".

Ahora se debe hacer una segunda convolución que será:

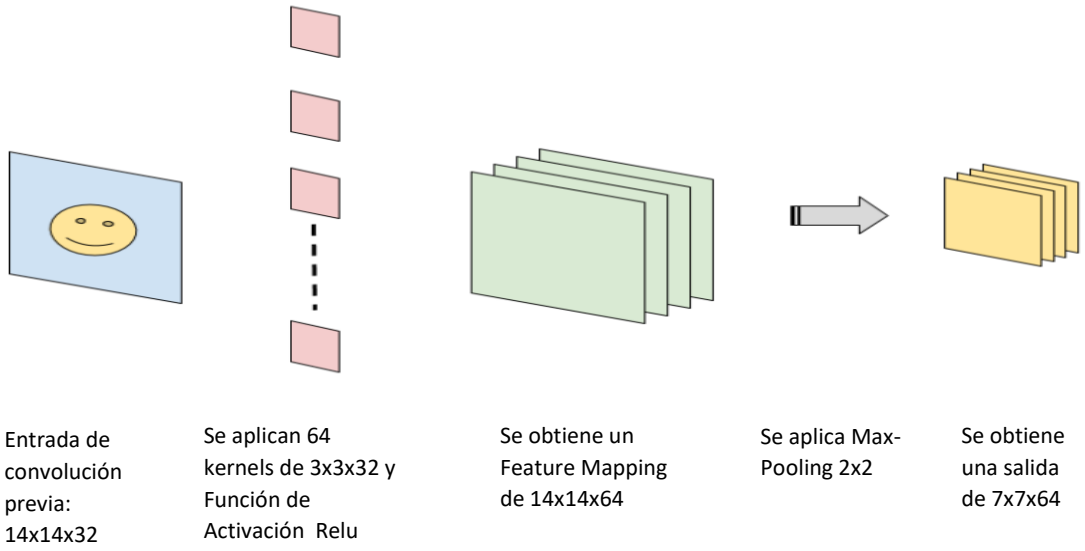


Figura 15. Segunda convolución y sucesivas [5]

La tercera convolución comenzará en tamaño 7×7 *pixels* y tras aplicar el *max-pooling* quedará en 3×3 con lo cual se podría hacer sólo una convolución más. En este ejemplo se empezó con una imagen de 28x28 *píxeles* y se hicieron 3 convoluciones. Si la imagen inicial hubiese sido mayor (de 224x224 *píxeles*) aún se hubiera podido seguir haciendo convoluciones.

1) Entrada: Imagen 7x7x64.
2) Se aplica *kernel*: 128 filtros de 3x3.
3) Se obtiene un *feature mapping* de 7x7x128.
4) Se aplica *Max-Pooling* de 2x2.
5) Se obtiene una "salida" de la convolución de 3x3x128.

Para terminar, se toma la última capa oculta a la que se hizo *subsampling*, que se dice que es "tridimensional" por tomar la forma 3x3x128 (alto, ancho, mapas) y se "aplana", esto es que deja de ser tridimensional, y pasa a ser una capa de neuronas "tradicionales".

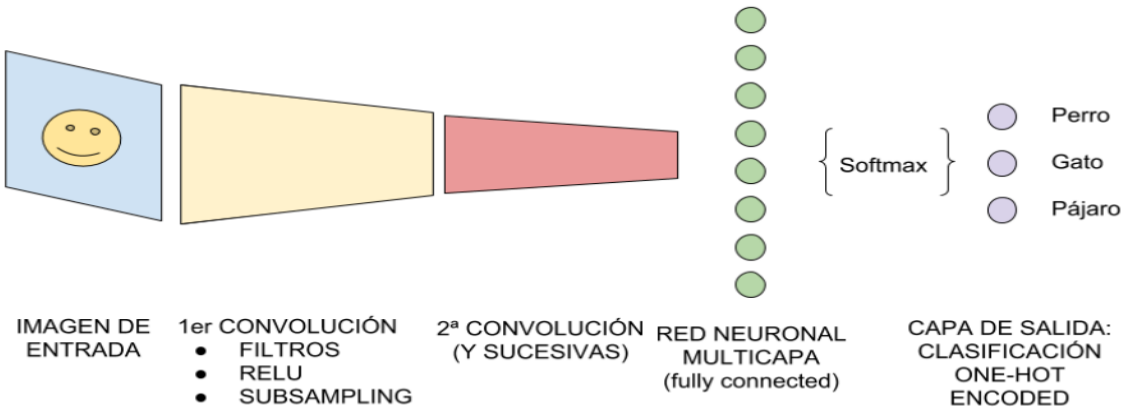


Figura 16. Arquitectura de una red neuronal convolucional [5]

Entonces, a esta nueva capa oculta "tradicional", se le aplica una función llamada *Softmax* que conecta contra la capa de salida final que tendrá la cantidad de neuronas correspondientes con las clases que se están clasificando. Si, por ejemplo, se clasifican perros y gatos, serán 2 neuronas, si se clasifican coches, aviones ó barcos serán 3, etc.

Las salidas al momento del entrenamiento tendrán el formato conocido como "*one-hot-encoding*" en el que para perros y gatos será: [1,0] y [0,1], para coches, aviones ó barcos será [1,0,0]; [0,1,0]; [0,0,1].

Y la función de *Softmax* se encarga de pasar a probabilidad (entre 0 y 1) a las neuronas de salida. Por ejemplo, una salida [0,2 0,8] indica 20% probabilidades de que sea perro y 80% de que sea gato.

2.4. Entrenamiento de una red neuronal

En el entrenamiento de una red, se dispone de los valores de entrada y se sabe cuáles son sus respectivas salidas. Lo que se pretende es ajustar los pesos para que la red neuronal aprenda, y que los valores de salida de la red se acerquen lo más posible a los valores reales conocidos.

Se alimentan en diferentes iteraciones a la red neuronal con los mismos **datos de entrenamiento** disponibles, y en cada iteración se intenta **minimizar la función de coste** (es una función que mide, en cierto modo, la **«diferencia» entre el valor de salida y el valor actual**)**.** Es decir, que la diferencia entre los valores de salida y los reales sean cada vez menores. Como los datos de entrada son los que son, sólo se puede conseguir **ajustando los pesos de las conexiones entre neuronas** al final de cada iteración.

Al final de cada iteración, para cada dato de entrada, se obtendrá un valor de salida (predicción) de la red neuronal. Este valor diferirá en una cantidad con respecto al valor real que medirá la función de coste.

Después hay que actualizar los pesos de las conexiones neuronales y retroalimentar la red neuronal con los datos iniciales del conjunto de datos. Se repite el proceso de nuevo y se van ajustando los pesos de forma que la función de coste sea cada vez menor.

Una manera de calcular la función de coste es utilizando el error cuadrático medio. Para ver su cálculo se puede ver en la figura siguiente:

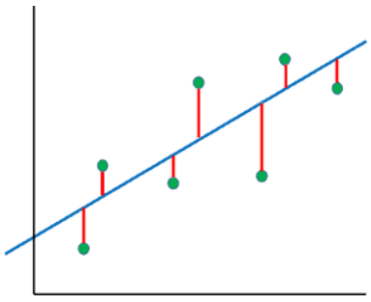


Figura 17. Error en regresión lineal [6]

En la figura se ve que se está usando una regresión lineal (en azul) para estimar los datos que se tienen (los puntos verdes). El modelo lineal tiene un error (en rojo) que se puede definir con la siguiente fórmula:

$$\text{error cuadrático} = (\text{real} - \text{estimado})^2 \qquad (1)$$

El valor estimado es el valor que nos da el modelo. En este caso, la línea azul.

Ahora que se conoce cómo calcular el error en cada punto, se puede calcular cual es el error medio. Para ello, se suman todos los errores y se dividen entre el número total de puntos. Si se llama *M* al número total de puntos nos queda la fórmula del Error Cuadrático Medio (MSE, por sus siglas en inglés, *Mean Squared Error*):

$$MSE = \frac{1}{M}\sum_{i=1}^{M}(real_i - estimado_i)^2 \quad (2)$$

Como técnica de *Machine Learning*, se usa una de la más simples, la regresión lineal.

En la figura se puede ver que se ha utilizado una regresión lineal para calcular la línea (en rojo). Esta línea obtiene el menor MSE posible para los datos que se han usado (en azul). El MSE (por sus siglas en inglés) es de 0.0332.

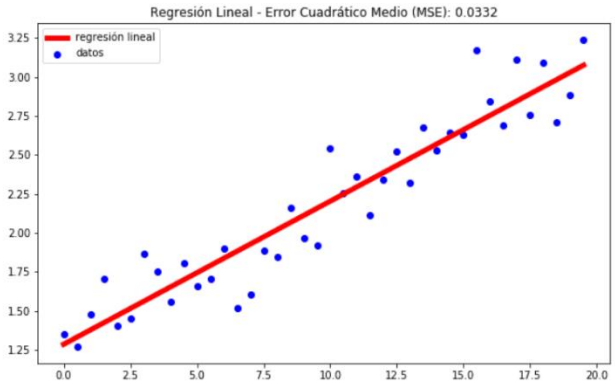


Figura 18. Error cuadrático medio (MSE) [6]

Una vez finalizado el proceso, se obtiene una red neuronal entrenada lista para predecir lo que se quiera suministrando una serie de datos de entrada. En general la red hará predicciones excelentes para datos que se han usado para entrenarla. Pero para evaluar la calidad de predicción de la red neuronal es necesario usar datos que no hayan sido usados para optimizar los pesos de las conexiones de la red.

2.5. Asignación de pesos en redes neuronales

Una vez que se dispone del diseño de la red neuronal, es necesario asignar pesos a cada conexión neuronal, de manera que la salida de la función de activación de cada neurona se vea afectada por el paso de cada conexión.

Suponiendo entonces que se tiene la red neuronal de la siguiente imagen:

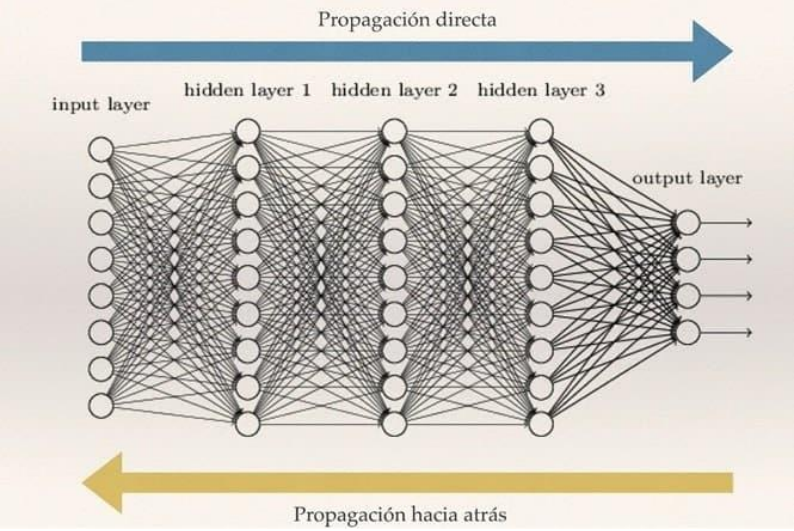


Figura 19. Red neuronal [7]

Se sabe que la información se introduce en la primera capa de neuronas por la izquierda, propagándose hacia delante atravesando las diferentes capas ocultas, hasta que llega a la capa de neuronas de salida, dando el resultado final. Este resultado se compara con los valores reales para calcular el error que se comete, y propagar este error hacia atrás a través de la red neuronal, lo que va a permitir ajustar los pesos durante el proceso de entrenamiento. Es muy importante recalcar que en la propagación hacia atrás **todos los pesos se ajustan de manera simultánea**. Existen varios algoritmos de asignación de pesos, como el *back propagation* o el descenso del gradiente.

2.6. Algoritmo de backpropagation

El objetivo del algoritmo *backpropagation* es **minimizar los errores en el proceso de aprendizaje y mejorar la precisión de las predicciones** ajustando los pesos y sesgos de cada nodo de la red. Lo que hace *backpropagation* es detectar cuánta responsabilidad tiene cada uno de los nodos de la red en los posibles errores y corregir la configuración de los parámetros de esos nodos, que luego influirán en otros nodos y capas posteriores.

El método emplea un ciclo de propagación–adaptación de dos fases, en resumen, permite que la información del costo fluya hacia atrás a través de la red para calcular el gradiente. Una vez que se ha aplicado un patrón a la entrada de la red como estímulo, este se propaga desde la primera capa a través de las capas siguientes de la red, hasta generar una salida. La señal de salida se compara con la salida deseada y se calcula una señal de error para cada una de las salidas. Las salidas de error entonces se propagan hacia atrás, partiendo de la capa de salida, hacia todas las neuronas de la capa oculta que contribuyen directamente a la salida. Sin embargo, las neuronas de la capa oculta solo reciben una fracción de la señal total del error, basándose aproximadamente en la contribución relativa que haya aportado cada neurona a la salida original. Este proceso se repite, capa por capa, hasta que todas las neuronas de la red hayan recibido una señal de error que describa su contribución relativa al error total.

La importancia de este proceso consiste en que, a medida que se entrena la red, las neuronas de las capas intermedias se organizan a sí mismas de tal modo que las distintas neuronas aprenden a reconocer distintas características del espacio total de entrada. Después del entrenamiento, cuando se les presente un patrón arbitrario de entrada que contenga ruido o que esté incompleto, las neuronas de la capa oculta de la red responderán con una salida activa si la nueva entrada contiene un patrón que se asemeje a aquella característica que las neuronas individuales hayan aprendido a reconocer durante su entrenamiento.

A continuación, se detallan los pasos necesarios para implementar este algoritmo:

1. Asignar a cada conexión neuronal un peso con un valor pequeño, pero no nulo.
2. Introducir la primera observación del conjunto de entrenamiento por la capa inicial de la red neuronal.
3. La información se propaga de izquierda a derecha, activando cada neurona que ahora es afectada por el peso de cada conexión, hasta llegar a la capa de neuronas de salida, obteniendo el resultado final para esa observación en concreto.
4. Se mide el error que se ha cometido para esa observación.
5. Comienza la propagación hacia atrás de derecha a izquierda, actualizando los pesos de cada conexión neuronal, dependiendo de la responsabilidad del peso actualizado en el error cometido.
6. Se repiten los pasos desde el paso 2, actualizando todos los pesos para cada observación o conjunto de observaciones de nuestro conjunto de entrenamiento.
7. Cuando todas las observaciones del conjunto de entrenamiento han pasado por la red neuronal, hemos completado lo que se denomina un *Epoch*. Podemos realizar tantos *Epochs* como creamos convenientes hasta que el resultado final sea el deseado.

2.7. Descenso del gradiente

El descenso de gradiente es un algoritmo de optimización que se usa comúnmente para entrenar modelos de *Machine Learning* y Redes neuronales. Los datos de entrenamiento ayudan a que estos modelos aprendan con el tiempo, y la función de coste dentro del descenso de gradiente actúa específicamente como un barómetro, midiendo su precisión con cada iteración de actualizaciones de parámetros. Hasta que la función sea cercana o igual a cero, el modelo continuará ajustando sus parámetros para producir la menor cantidad de errores posible.

El punto de partida es solo un punto arbitrario para que se pueda evaluar el rendimiento. Desde ese punto de partida, se encontrará la derivada (o pendiente), y desde allí, se puede usar una recta tangente para observar la inclinación de la pendiente. La pendiente informará de las actualizaciones de los parámetros, es decir, los pesos y el sesgo. La pendiente en el punto de partida será más pronunciada, pero a medida que se generen nuevos parámetros, la pendiente debe reducirse gradualmente hasta alcanzar el punto más bajo de la curva, conocido como punto de convergencia.

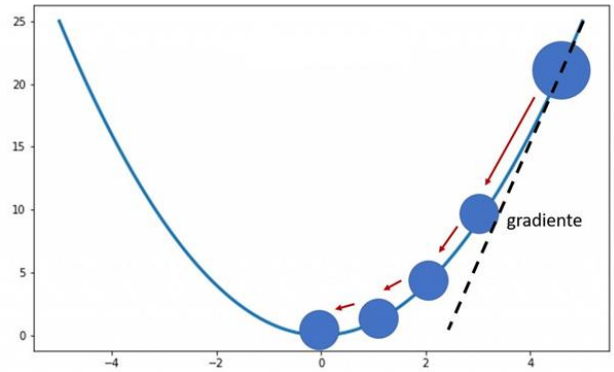


Figura 20. Descenso del Gradiente [8]

El objetivo del descenso de gradiente es minimizar la función de coste, o el error entre $y_{pronosticada}$ e y_{real}. Para hacer esto, se necesitan dos puntos de datos: una dirección y una tasa de aprendizaje.

Estos factores determinan los cálculos de derivadas parciales de iteraciones futuras, lo que le permite llegar gradualmente al mínimo local o global (es decir, punto de convergencia).

- **Tasa de aprendizaje** es el tamaño de los pasos que se dan para alcanzar el mínimo. Suele ser un valor pequeño y se evalúa y actualiza en función del comportamiento de la función de coste (en la figura anterior serían los pasos que va dando la bola en su trayectoria descendiente). Las tasas de aprendizaje dan como resultados pasos más grandes, pero se corre el riesgo de sobrepasar el mínimo. Por el contrario, una tasa de aprendizaje baja tiene tamaños de paso pequeños. Si bien tiene la ventaja de una mayor precisión, el número de iteraciones compromete la eficiencia general, ya que esto requiere más tiempo y cálculos para alcanzar el mínimo.
- **La función de coste (o pérdida)** mide la diferencia, o error, entre la ´y_{real}´ y la ´$y_{pronosticada}$´ en su posición actual. Esto mejora la eficacia del modelo de *Machine Learning* , proporcionando *feedback* al modelo para que pueda ajustar los parámetros para minimizar el error y encontrar el mínimo local o global. Repite continuamente, moviéndose a lo largo de la dirección de descenso más pronunciado (o el gradiente negativo) hasta que la función de costo valga cero o esté cerca de cero. En este punto, el modelo dejará de aprender. Además, si bien los términos función de coste y función de pérdida se consideran sinónimos, existe una ligera diferencia entre ellos. Vale la pena señalar que una función de pérdida se refiere al error de un ejemplo de entrenamiento, mientras que una función de coste calcula el error promedio en todo un conjunto de entrenamiento.

2.8. Overfitting

Cuando se desarrolla un modelo de ***Machine Learning***, se intenta enseñar cómo lograr un objetivo, hacer una detección de algo o hacer una clasificación, etc. Para ello se parte de una base de datos que será utilizada para entrenar el modelo, es decir, para aprender a utilizarlo para lograr el objetivo deseado. Sin embargo, si no se hacen correctamente, es posible que el modelo considere como validado, solo los datos que se han usado para entrenar el modelo, sin reconocer ningún otro dato que sea un poco diferente a la base de datos inicial. Este fenómeno se llama ***overfitting* en *Machine Learning***.

Se dice que un **modelo estadístico está sobreajustado** cuando se entrena con muchos datos. Cuando un modelo se entrena con tantos datos, comienza a aprender del ruido y de las entradas de datos inexactas en el conjunto de datos empleado. Entonces, el modelo no categoriza los datos correctamente, debido a demasiados detalles y ruido.

Las causas del **sobreajuste** son los métodos no paramétricos y no lineales porque estos tipos de algoritmos de aprendizaje automático tienen más libertad para construir el modelo basado en el conjunto de datos y, por lo tanto, realmente pueden construir modelos poco realistas. Para evitar el *overfitting* se puede hacer lo siguiente:

- Dividir los datos en entrenamiento, validación y prueba.
- Obtener un mayor número de datos.
- Ajustar los parámetros de los modelos.
- Utilizar modelos más simples.

- Que los datos vengan de distintas distribuciones.
- Bajar el número de iteraciones en los algoritmos iterativos.

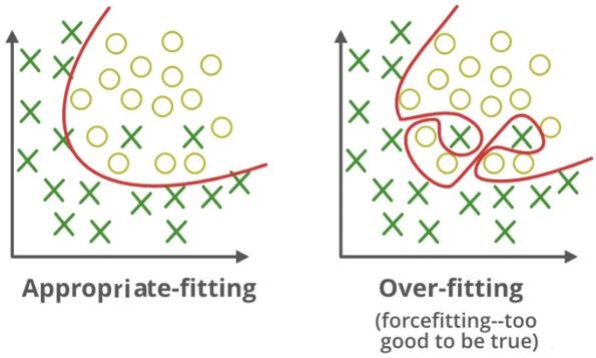


Figura 21. Overfitting [9]

Para complementar el concepto de *overfitting*, es necesario comprender una serie de conceptos clave.

- *Sweet spot*: es el punto medio que se debe encontrar en el aprendizaje del modelo en el que se asegure que no se está incurriendo en underfitting u overfitting, lo cual a veces puede resultar una complicada tarea.
- *Bias*: se trata de la **diferencia entre los valores predichos y los valores reales** o verdaderos en el modelo. No siempre es fácil para el modelo aprender de señales bastante complejas. De manera muy simple, un alto sesgo o bias indican que el modelo sufre de *underfitting* o subajuste.
- Varianza: se refiere a la sensibilidad del modelo a conjuntos específicos en los datos de entrenamiento. Un **algoritmo de alta varianza** producirá un modelo extraño que es drásticamente diferente del conjunto de entrenamiento.
- *Accuracy*: es la relación entre el número de predicciones correctas y el número total de muestras de entrada.

2.9. Underfitting

Se refiere a los casos en los que un modelo de datos no es capaz de identificar patrones, es decir, no es capaz de capturar de forma precisa la relación entre las variables de entrada y salida, de modo que se genera un alto índice de errores en el conjunto de entrenamiento y en los datos no vistos. Ocurre cuando un modelo es demasiado simple, lo que puede deberse a que necesita más tiempo de entrenamiento, más funciones de entrada o menos regularización. Cuando un modelo está subajustado, no puede establecer la tendencia dominante dentro de los datos, lo que provoca errores de entrenamiento y un rendimiento deficiente del modelo. Si un modelo no puede generalizar a nuevos datos, entonces no se puede utilizar para tareas de clasificación o predicción. La generalización de un modelo a nuevos datos es, en última instancia, lo que permite utilizar algoritmos de *Machine Learning* cada día para hacer predicciones y clasificar datos.

Un sesgo alto y una varianza baja son indicadores de subajuste. Dado que este comportamiento se puede ver al utilizar el conjunto de datos de entrenamiento, los modelos subajustados suelen ser más fáciles de identificar que los sobreajustados.

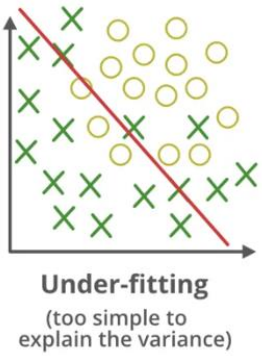


Figura 22. Underfitting [9]

Formas de prevenir el *underfitting*:

- Tratar los datos correctamente, eliminando los datos anómalos de un conjunto de datos (*outliers*) y variables innecesarias.
- Utilizar modelos más complejos.
- Ajustar los parámetros de los modelos.
- Aumentar las iteraciones en los algoritmos iterativos.

2.10. Transformada de Fourier

Una señal es una variación de una determinada cantidad a lo largo del tiempo.

Para el audio, la cantidad que varía es la presión del aire. ¿Cómo se captura esta información digitalmente? Se puede tomar muestras de la presión del aire a lo largo del tiempo. La velocidad a la que se muestrea los datos puede variar, pero normalmente es de 44,1 kHz o 44100 muestras por segundo. Lo que se ha capturado es una forma de onda para la señal, y esto puede ser interpretado, modificado y analizado con software de un ordenador.

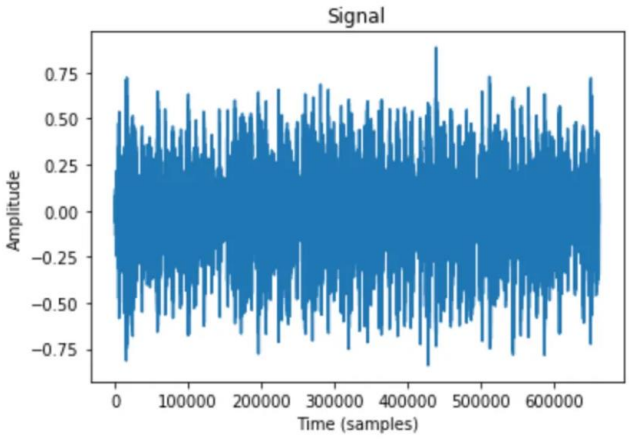


Figura 23. Señal de audio[10]

Una señal de audio se compone de varias ondas sonoras de una sola frecuencia. Al tomar muestras de la señal a lo largo del tiempo, solo se capturan las amplitudes resultantes. La **transformada de Fourier** es una fórmula

matemática que nos permite descomponer una señal en sus frecuencias individuales y la amplitud de la frecuencia. En otras palabras, convierte la señal del dominio del tiempo al dominio de la frecuencia. El resultado se llama **espectro** .

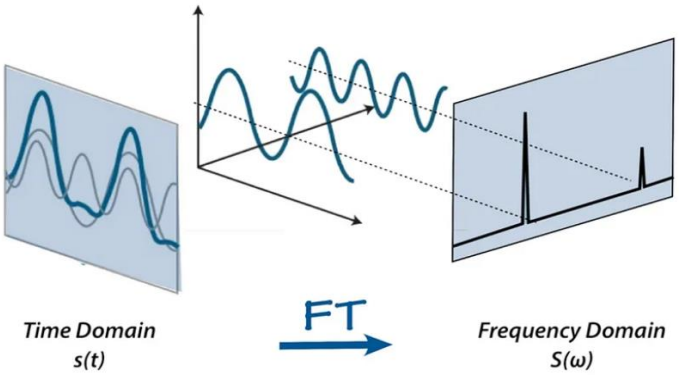


Figura 24. Transformada de Fourier[10]

Esto es posible porque cada señal se puede descomponer en un conjunto de ondas seno y coseno que se suman a la señal original. Este es un teorema notable conocido como **el teorema de Fourier.**

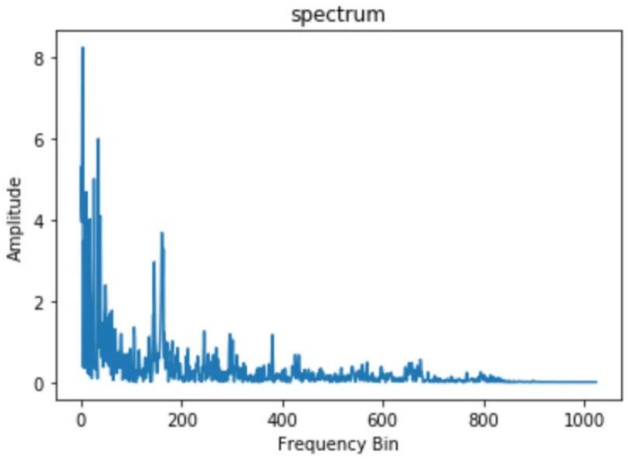


Figura 25. Espectro de la Transformada de Fourier[10]

2.11. El espectrograma

La transformada rápida de Fourier es una herramienta poderosa que nos permite analizar el contenido de frecuencia de una señal. En el caso que ocupa a este Trabajo fin de Grado, son señales de audio que varían con el tiempo, es decir, son señales no periódicas. Para representar el espectro de estas señales a medida que varíen con el tiempo se emplea la Transformada rápida de Fourier (FFT). La FFT se calcula en segmentos de ventana superpuestos de la señal, y se obtiene un **espectrograma.**

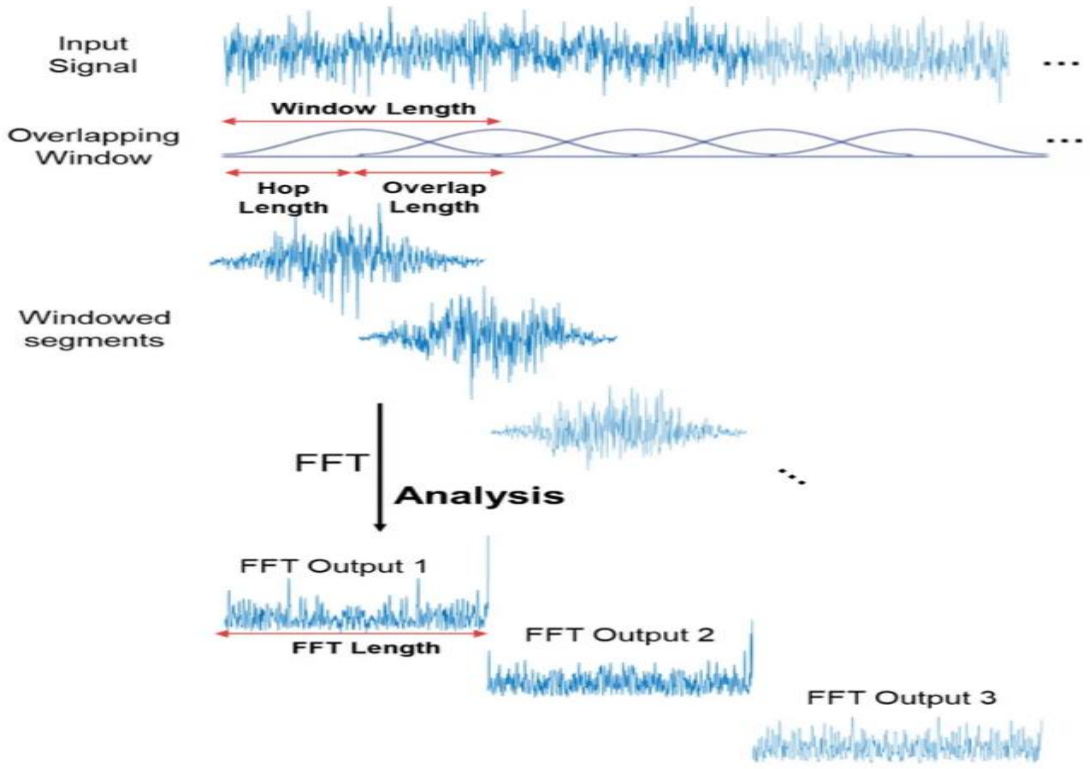
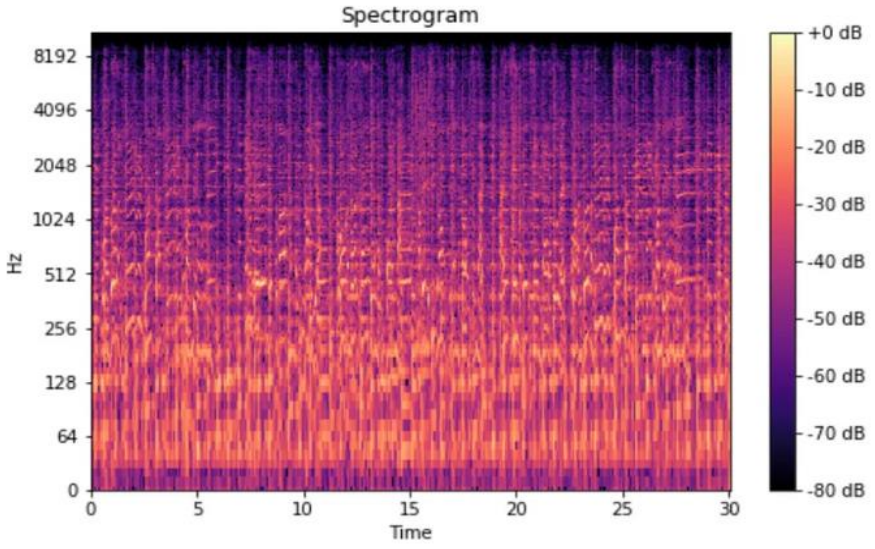


Figura 26. Descomposición de una señal de audio en Transformadas de Fourier[10]

Un espectrograma se puede considerar como un montón de FFT apiladas una encima de la otra. Es una forma de representar visualmente el volumen o la amplitud de una señal, ya que varía con el tiempo a diferentes frecuencias. El eje y se convierte a una escala logarítmica y la dimensión del color se convierte a decibelios. Esto se debe a que los humanos solo pueden percibir un rango muy pequeño y concentrado de frecuencias y amplitudes.



7

Figura 27. Espectrograma resultante[10]

2.12. La escala Mel

Los estudios han demostrado que los humanos no perciben frecuencias en una escala lineal. Somos mejores detectando diferencias en frecuencias más bajas que en frecuencias más altas. Por ejemplo, se puede distinguir fácilmente la diferencia entre 500 y 1000 Hz, pero difícilmente se puede distinguir entre 10 000 y 10 500 Hz, aunque la distancia entre los dos pares sea la misma.

En 1937, Stevens, Volkmann y Newmann propusieron una unidad de tono tal que distancias iguales en el tono sonaran igualmente distantes para el oyente. Esto se llama escala de Mel**.** Se realiza una operación matemática sobre las frecuencias para convertirlas a la escala Mel.

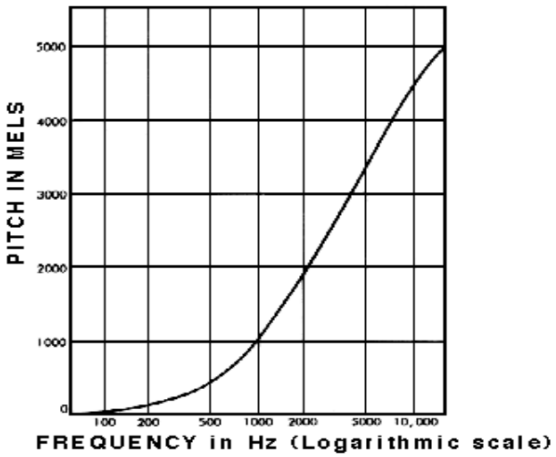


Figura 28. Escala Mel [10]

Un **espectrograma Mel** es un espectrograma en el que las frecuencias se convierten a la escala de Mel.

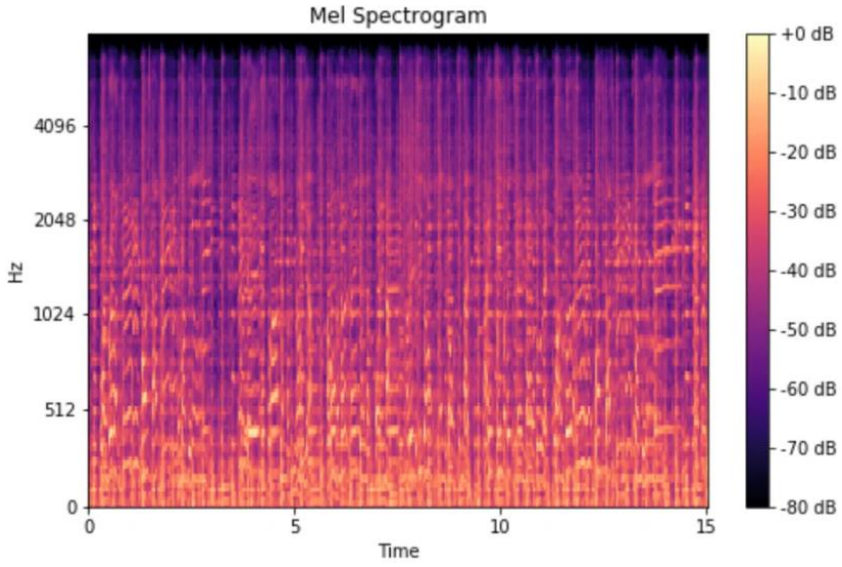


Figura 29. Espectrograma Mel [10]

Para conseguir este espectrograma, en resumen, se siguen estos pasos:

- Se toman muestras de la presión del aire a lo largo del tiempo para representar digitalmente una señal de audio.

- Se mapea la señal de audio desde el dominio del tiempo al dominio de la frecuencia utilizando la transformada rápida de Fourier, y se realiza en segmentos de ventana superpuestos de la señal de audio.
- Se convierte el eje y (frecuencia) a una escala logarítmica y la dimensión de color (amplitud) a decibelios para formar el espectrograma.
- Se mapea el eje y (frecuencia) en la escala de Mel para formar el espectrograma de Mel.

3. HARDWARE DEL SISTEMA

Para desarrollar este Trabajo Fin de Grado se va a emplear el kit de desarrollo compuesto por la PCB STEVAL-STLCS01V1 y la PCB STLCR01V1, al conjunto se llama Sensor Tile.

La diminuta placa base del sistema SensorTile (13,5 x 13,5 mm) incorpora sensores inerciales de muy baja potencia y alta precisión, un sensor de presión barométrica y un micrófono digital. La MCU integrada de 80 MHz cuenta con un interfaz de micrófono y un consumo muy bajo. Además, proporciona Bluetooth Smart maximizando el rendimiento en radio frecuencia.

El circuito integrado a utilizar se compone de dos PCB, la primera es la STEVAL-STLCS01V1 la cual se puede ver en la siguiente figura:

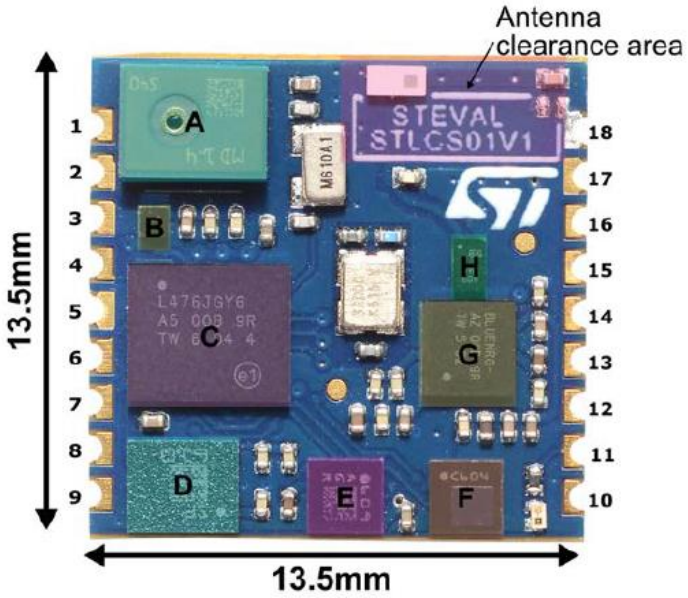


Figura 30. PCB STLCS01V1[11]

Los circuitos integrados que se encuentran en la PCB STEVAL-STLCS01V1 son:

(A)Micrófono digital MP34DT05-A MEMS.
(B)LDO de 1.8V y 150mA de bajo ruido LD39115J18R.
(C)Microcontrolador de 32-bit STM32L476 MCU ARM Cortex-M4.
(D)Módulo inercial iNEMO LSM6DSM con acelerómetro 3D y giróscopo en 3D.
(E)Módulo eCompass y magnetómetro LSM303AGR.
(F)Sensor de presión digital LPS22HB (260-1260 hPa).
(G)Procesador bluetooth BlueNRG-MS.
(H)Filtro de harmónicos BALF-NRG-02D3 50 Ω.

La PCB STLCS01V1 tiene el siguiente diagrama de bloques:

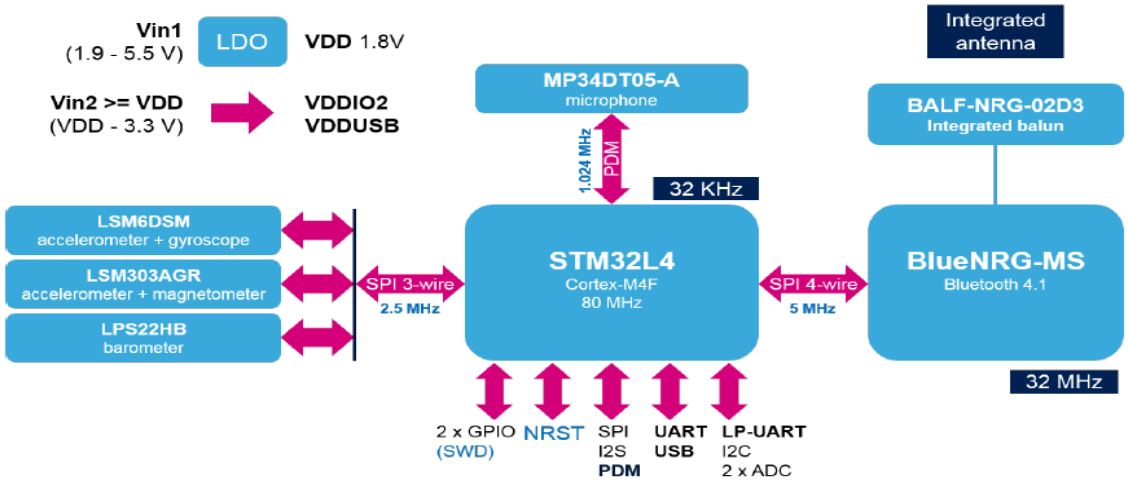


Figura 31. Diagrama de bloques de la PCB STLCS01V1[11]

A continuación, se puede ver el diseño esquemático de STEVAL-STLCS01V1, comenzando con el microcontrolador DSP STM32L476xx, el cristal y el led.

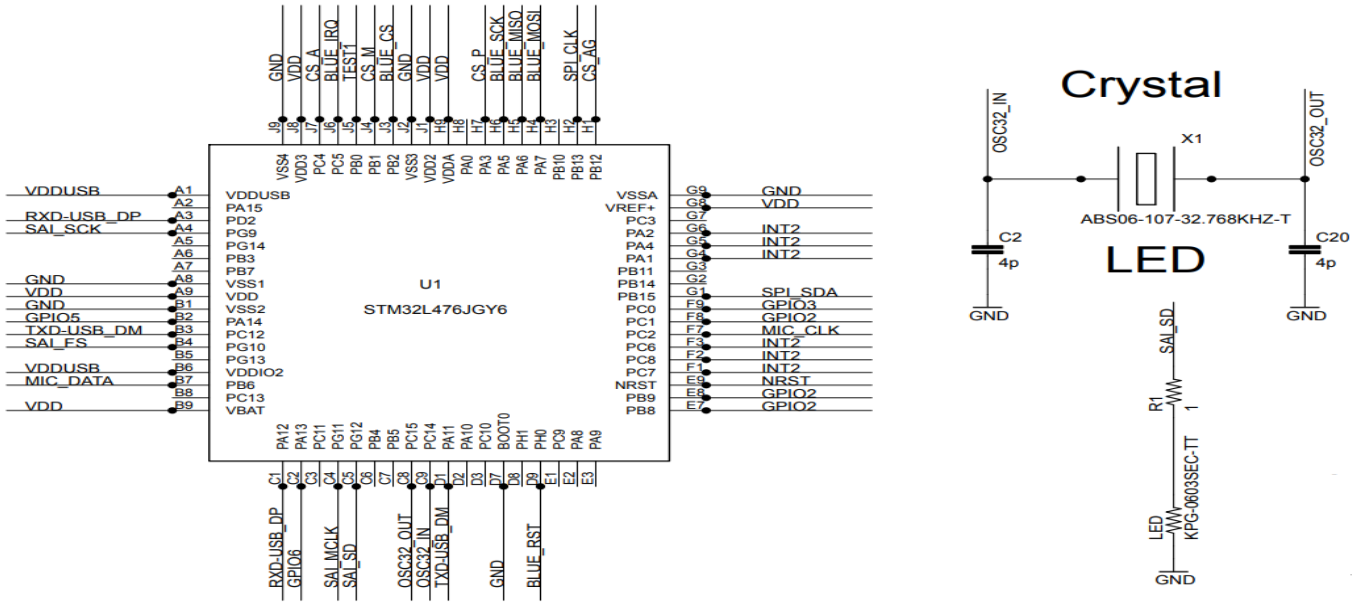


Figura 32. Microcontrolador DSP STM32L476xx , cristal y led. [11]

En la figura de abajo el *moon pin out* se refiere al *pinout* del módulo STEVAL-STLCS01V1. También dispone de 7 condensadores de desacoplo.

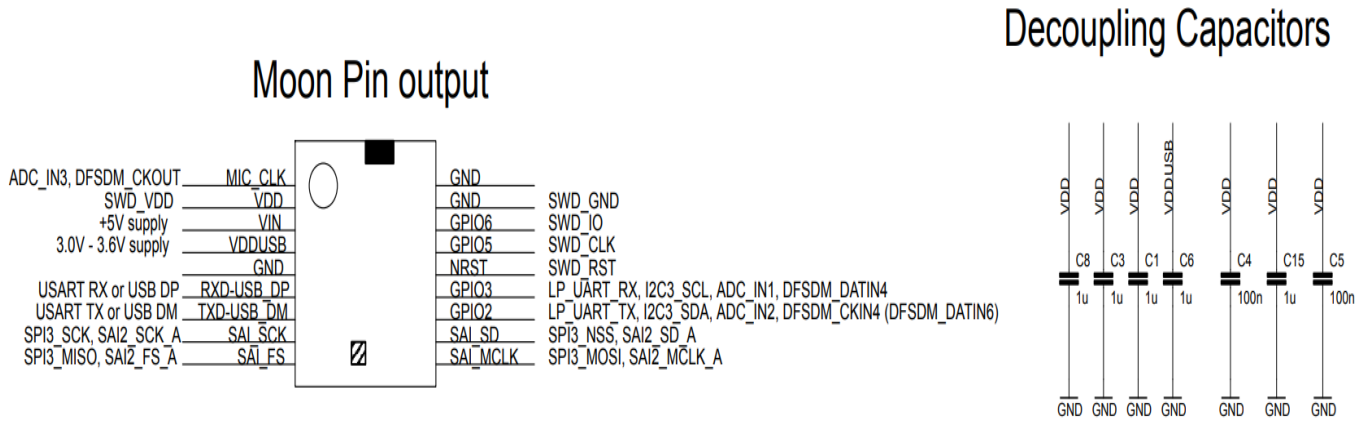


Figura 33. Pinout y condensadores de desacoplo. [11]

La PCB STEVAL-STLCS01V1 dispone de un regulador de tensión y de un conector Hirose para conectar a un conector hembra como se ve en la figura bajo estas líneas. Recalcar que para trabajar con la PCB STEVAL-STLCS01V1 hay que soldarlo a la PCB STLCR01V1.

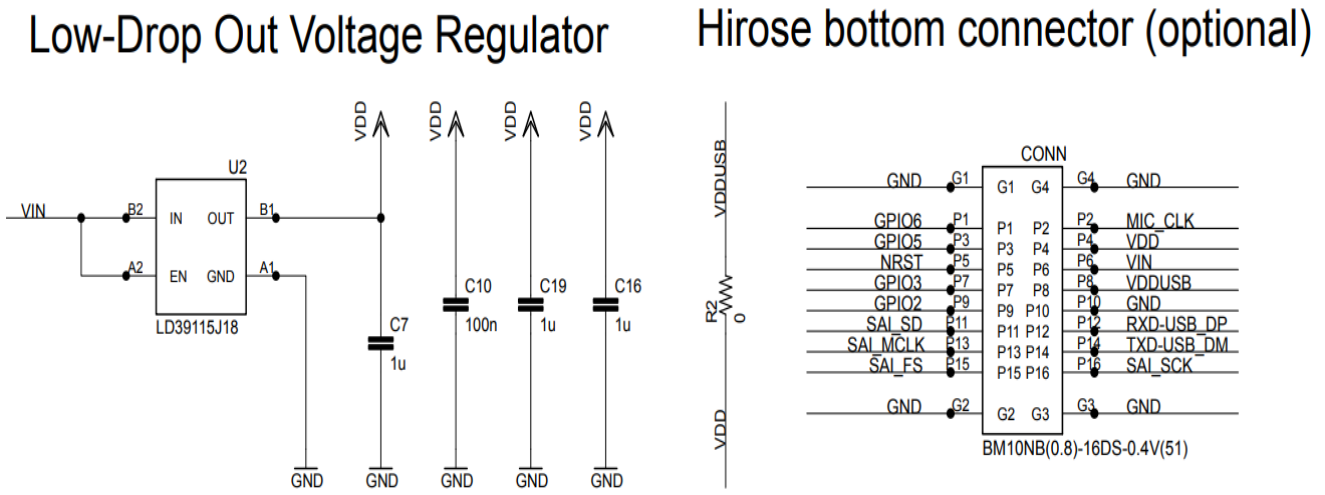


Figura 34. Regulador y conector Hirose [11]

También tiene un chip de bluetooth, un acelerómetro y magnetómetro.

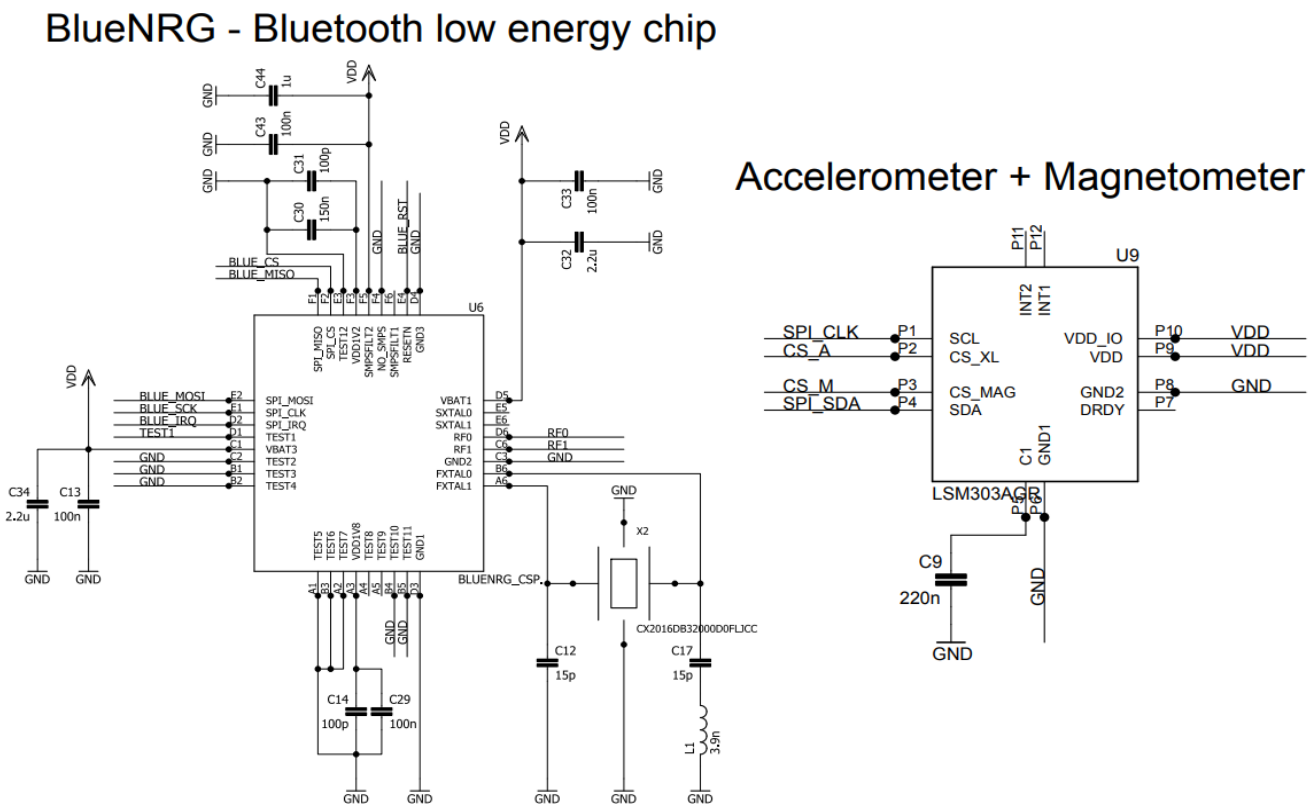


Figura 35. Bluetooth, acelerómetro y magnetómetro[11]

Por último, se puede apreciar el diseño esquemático del acelerómetro y giróscopo, el chip de antena para el bluetooth, el sensor de presión y el micrófono.

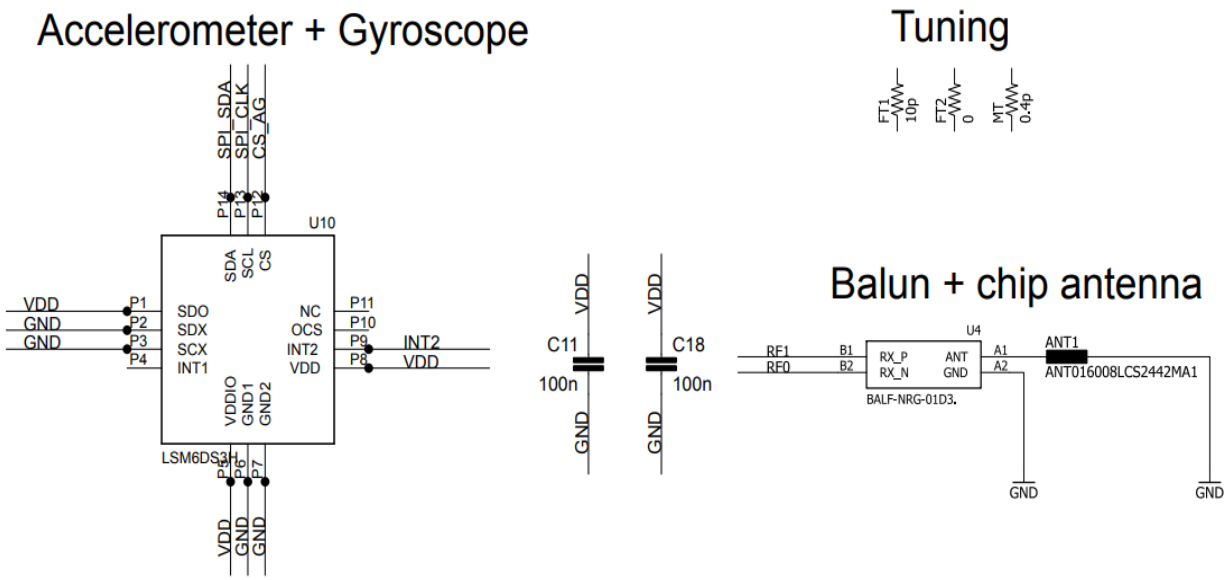


Figura 36. Acelerómetro, giróscopo y chip de antena. [11]

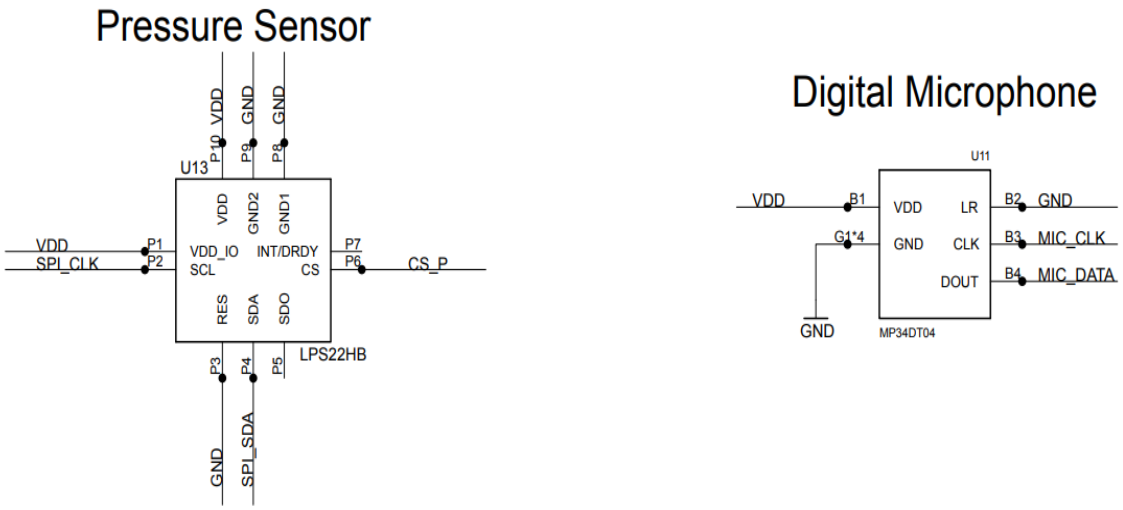


Figura 37. Sensor de presión y micrófono digital. [11]

La PCB STEVAL-STLCS01V1 hay que soldarlo a la PCB STLCR01V1, la cual se puede ver en la siguiente figura.

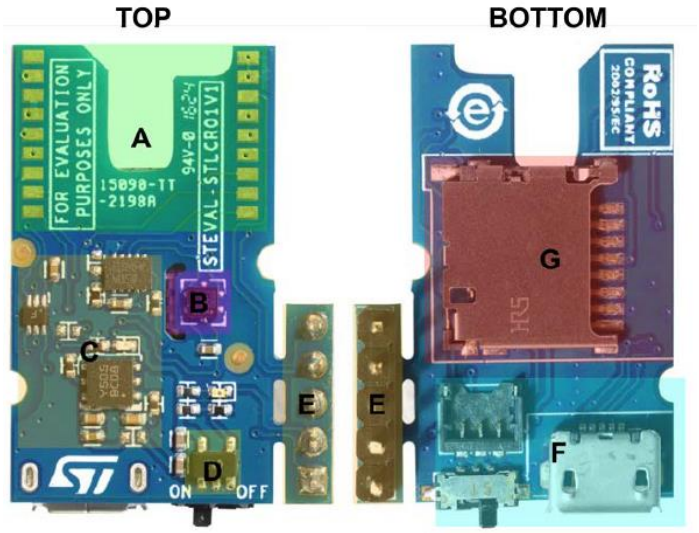


Figura 38. PCB STLCR01V1. [11]

La tarjeta STLCR01V1 está compuesta por:

- (A) Huella de la tarjeta STEVAL-STLCS01V1 para soldarla.
- (B) HTS221 es un sensor digital capacitivo de humedad y temperatura.
- (C) STBC08PMR es un cargador lineal de 800 mA con regulador térmico para batería de iones de litio. El STC3115 es un medidor de carga de batería. El LDK120M-R es un LDO de muy bajo ruido y corriente de 200 mA. El USBLC6-2P6 es un protector de ESD de baja capacitancia.
- (D) Interruptor de encendido on/off.
- (E) Conector SWD de 5 pines para programación y depuración.
- (F) Conector micro USB para carga de batería y puerto de comunicación. Conector de batería de 3 pines para conectar la batería de iones de lítio.
- (G) Slot para tarjeta microSD.

A continuación se puede ver el diseño esquemático empleado en el diseño de la tarjeta STLCR01V1.

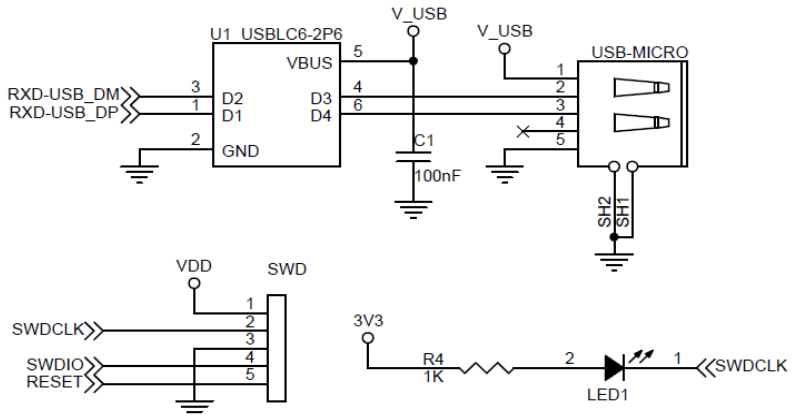


Figura 39 Micro USB, ESD, pines y led. [11]

En la figura 39 se muestra del conector micro USB, el protector de ESD USBLC6-2P6, una tira de pines de paso 2,54´´ para conectar el programador y un led.

En la figura siguiente se aprecia la huella donde hay que soldar la PCB STEVAL-STLCS01V1.

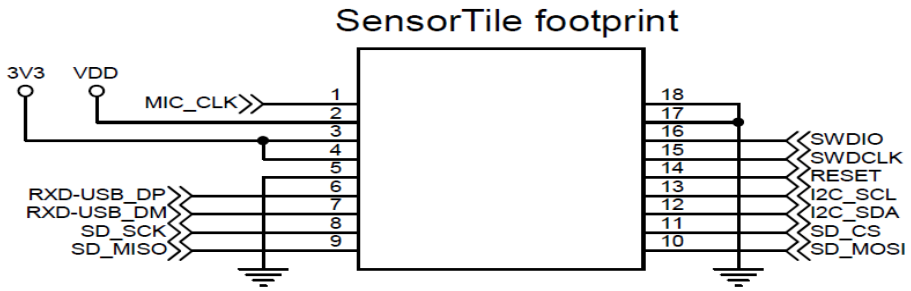


Figura 40 Huella del STEVAL-STLCS01V1. [11]

En la figura siguiente se puede apreciar el diseño del slot para tarjeta microSD y el sensor de temperatura y humedad HTS221.

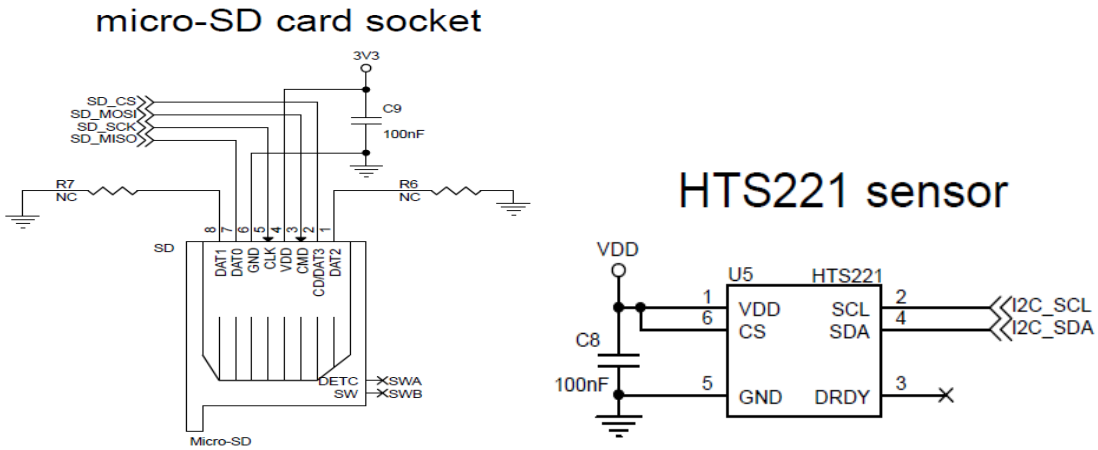


Figura 41 Micro SD y sensor de temperatura y humedad. [11]

Por último, se ve el diagrama de conexión del LDO LDK120M-R, el cargador lineal STBC08PMR, el conector de batería, el led que indica el estado de carga de la batería y el medidor de carga de la batería STC3115.

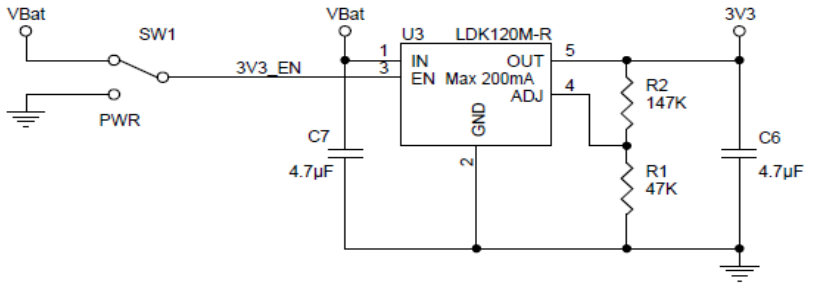


Figura 42 LDO. [11]

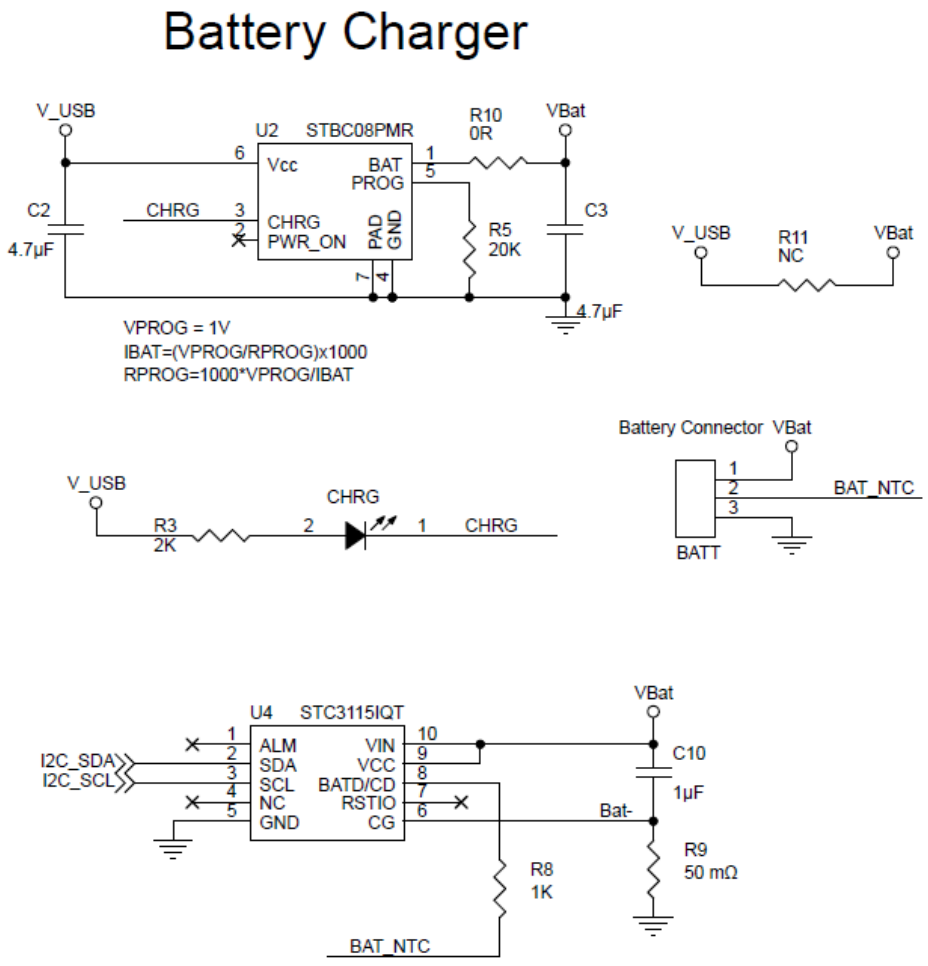


Figura 43 Cargador STBC08PMR, led, conector y medidor de batería STC3115IQT. [11]

En adelante, para cuando se nombre Sensor Tile durante este Trabajo Fin de Grado se refiere al conjunto formado por la PCB STEVAL-STLCS01V1 y STLCR01V1.

En la siguiente figura se pude ver como conectar la batería y cómo conectar el cable de programación del Sensor Tile.

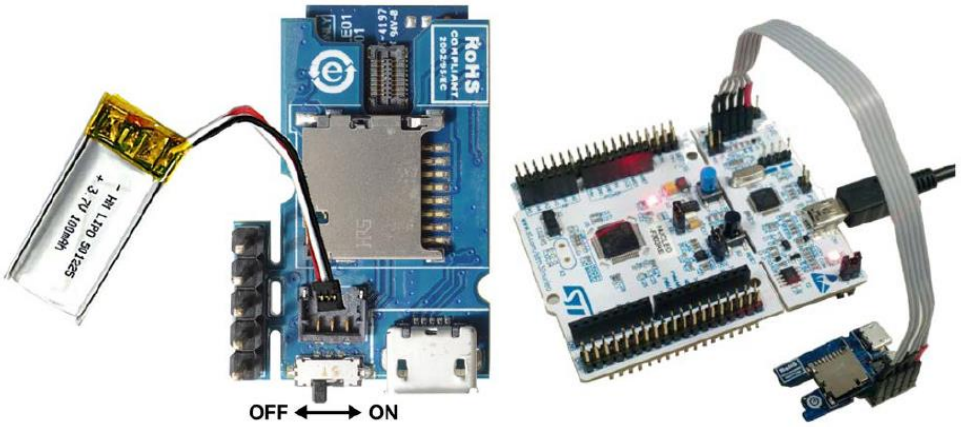


Figura 44 Conexión de batería y cable de programación. [11]

Una vez soldado se puede meter en su carcasa para su utilización como se muestra en la siguiente figura.

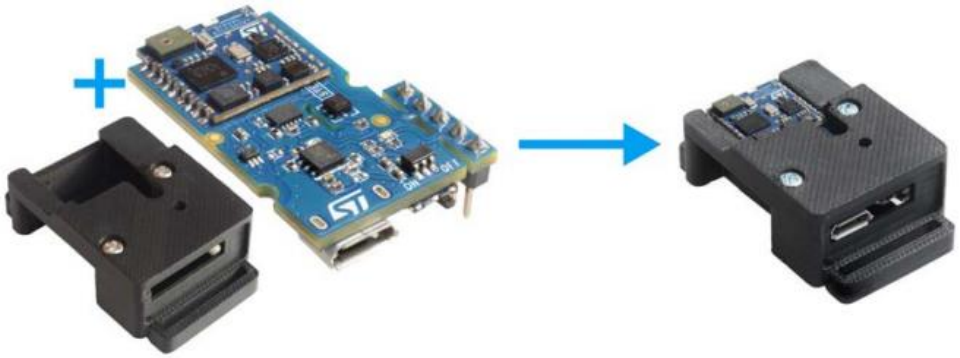


Figura 45 Carcasa. [11]

A continuación, se va explicar un poco más en detalle el hardware empleado en el Trabajo Fin de Grado que es el microcontrolador, el micrófono y el periférico DFSDM (*Digital Filter for Sigma-Delta Modulator*). Éste último periférico, el DFSDM, se emplea para realizar la conversión de la señal PDM proporcionada por el micrófono digital a PCM.

3.1. El microcontrolador

El Sensor Tile monta un microcontrolador STM32L476JGY6TR. Esta familia de microcontroladores es de muy bajo consumo y están basados en el núcleo RISC de 32 bits Arm® Cortex®-M4 de alto rendimiento que funciona a una frecuencia de hasta 80 MHz. El núcleo Cortex-M4 presenta una unidad de precisión de coma flotante (FPU) que admite todas las instrucciones de procesamiento de datos y tipos de datos para Arm®.

También implementa un conjunto completo de instrucciones DSP y una unidad de protección de memoria (MPU) que mejora las aplicaciones de seguridad.

Los microcontroladores STM32L476xx incorporan memorias de alta velocidad (memoria Flash de hasta 1 Mbyte y SRAM de hasta a 128 Kbyte), un controlador flexible de memoria externa (FSMC) para memorias estáticas, una interfaz de memorias flash Quad SPI y una amplia gama de E/S y periféricos mejorados conectados a dos buses APB, dos buses AHB y una matriz de bus multi-AHB de 32 bits.

Los microcontroladores STM32L476xx incorporan varios mecanismos de protección para memoria Flash y SRAM: protección de lectura, protección contra escritura, protección contra lectura de salida de código propietario y firewall.

Esta familia de microcontroladores ofrece hasta tres rápidos ADC de 12 bits (5 Mbps), dos comparadores, dos amplificadores operacionales, dos canales DAC, un búfer de referencia de voltaje interno, un RTC de baja potencia, dos temporizadores de 32 bits de propósito general, dos temporizadores PWM de 16 bits dedicados al control de motores, siete temporizadores de propósito general de 16 bits y dos temporizadores de bajo consumo de 16 bits. Admite cuatro filtros digitales para moduladores sigma delta externos (DFSDM).

Además, hay disponibles hasta 24 canales de detección capacitivos. El microcontrolador también incorpora un controlador LCD integrado de 8x40 o 4x44 con un convertidor *step-up* interno.

Cuenta con interfaces de comunicación estándar y avanzados:

- Tres I2C.
- Tres SPI.
- Tres USART, dos UART y un UART de bajo consumo.
- Dos SAI (interfaces de audio en serie).
- Un SDMMC.
- Un CAN.
- Un USB OTG de alta velocidad.
- Una SWPMI (interfaz maestra de protocolo de cable único).

El STM32L476xx funciona en el rango de temperatura de -40 a +85 °C, -40 a +105 °C y -40 a +125 °C. El rango de alimentación va de 1,71 a 3.6 V Vdd cuando se utiliza un regulador LDO interno y cuando se utiliza un SMPS externo de 1,05 a 1,32 V. Permite el diseño de aplicaciones de bajo consumo por el conjunto de modos de ahorro de energía que tiene.

Admite algunas fuentes de alimentación independientes: entrada de alimentación independiente analógica para ADC, DAC, OPAMP y comparadores, entrada de alimentación dedicada de 3,3 V para USB y hasta 14 I/Os que se pueden alimentar de forma independiente hasta 1,08 V. Una entrada VBAT permite hacer un *backup* del RTC y de los registros. Se pueden usar fuentes de alimentación VDD12 dedicadas para omitir el regulador LDO cuando se conecta a un SMPS externo.

3.2. El micrófono

El micrófono digital MEMS es un sensor que convierte las ondas de presión acústica en una señal digital. Los MCU y MPU de STM32 adquieren datos digitales de los micrófonos a través de sus periféricos para ser procesados y transformados en audio estándar. Después, el microcontrolador maneja los datos de audio de acuerdo con la aplicación de audio de destino.

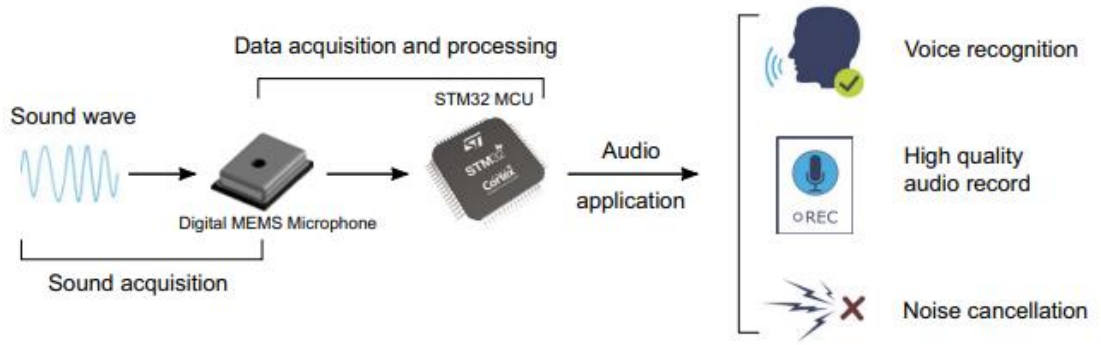


Figura 46 Esquema típico de funcionamiento de un micrófono con un microntrolador.[12]

Las partes principales de un micrófono digital son el transductor, el amplificador, el modulador PDM.

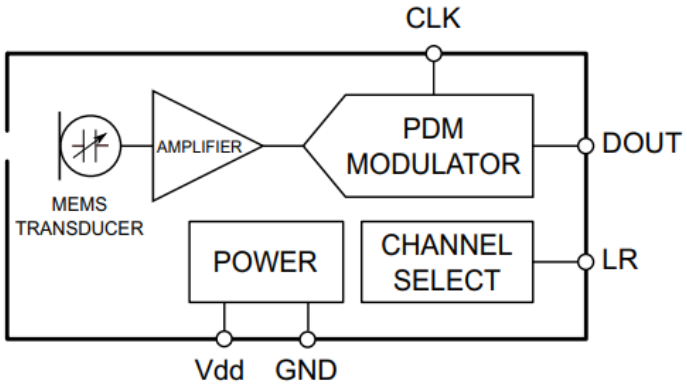


Figura 47 Partes principales de un micrófono. [12]

Transductor MEMS: El transductor MEMS es una capacitancia variable que convierte el cambio de la presión del aire causado por el sonido a un voltaje.

Amplificador: El amplificador amortigua el voltaje proporcionado por el transductor MEMS y proporciona una señal suficientemente fuerte al modulador PDM.

Modulador PDM: El modulador convierte la señal analógica en una señal modulada en PDM. La entrada de reloj (CLK) se utiliza para controlar el modulador PDM. El rango de la frecuencia del reloj de los micrófonos digitales ST es de 1 MHz a 3,25 MHz. Esta frecuencia define la frecuencia de muestreo a la que se muestrea la señal de salida analógica del amplificador para producir una representación en tiempo discreto (flujo de bits PDM).

Selección de canal: La salida del micrófono es llevada al nivel adecuado en el flanco de reloj elegido y luego entra en un estado de alta impedancia durante la otra mitad del ciclo de reloj. La selección de canal define el flanco de reloj en el que el micrófono digital emite datos válidos. El pin LR debe conectarse a Vdd o GND.

Señal PDM: PDM es una forma de modulación utilizada para representar una señal analógica en el dominio digital. Es un flujo de alta frecuencia de muestras digitales de 1 bit. En una señal PDM, la densidad relativa de los pulsos corresponde a la amplitud de la señal analógica. Un grupo grande de unos corresponde a un valor de amplitud alto (positivo), cuando un grupo grande de ceros correspondería a un valor bajo de amplitud (negativo), y la alternancia de 1 y 0 correspondería a una amplitud de valor cero.

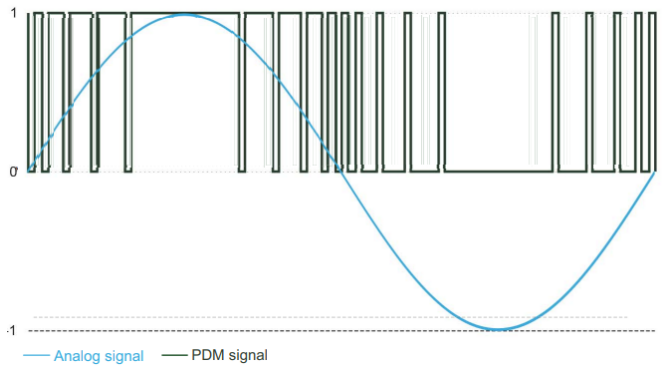


Figura 48 Señal PDM y analógica. [12]

Señal PCM: En una señal PCM, los valores de amplitud son codificados en pulsos. Cuando se realiza una transmisión codificada en PCM hay que tener en cuenta dos propiedades básicas que determinan la fidelidad de la señal analógica:

- La tasa de muestreo.
- La profundidad de bits.

La frecuencia de muestreo es el número de muestras de una señal que se toman por segundo para representarlo digitalmente. La profundidad de bits determina el número de bits de información en cada muestra.

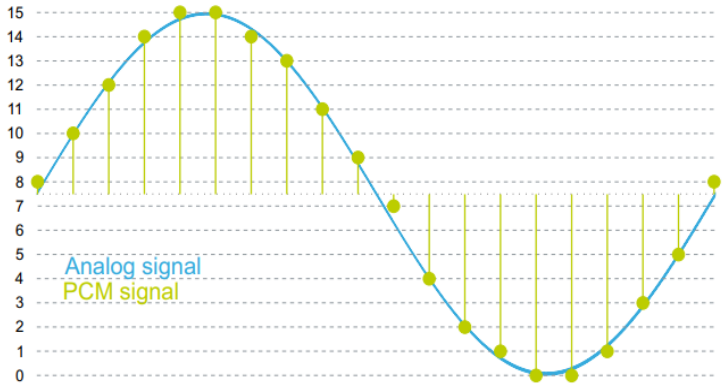


Figura 49 Señal PCM y analógica. [12]

Conversión de PDM a PCM: Para que el microcontrolador pueda trabajar con la señal proporcionada por el micrófono, hay que hacer una conversión de formato, hay que pasarla de PDM a PCM, para ello las señales PDM deben filtrarse y diezmarse.

En la etapa de diezmado, la frecuencia de muestreo de la señal PDM se reduce a la frecuencia objetivo de muestreo de audio (16 kHz, por ejemplo). Al seleccionar una de cada M muestras, la tasa de muestreo se reduce por un factor de M. Por lo tanto, la frecuencia de datos PDM (la cual es la frecuencia del reloj del micrófono) es M veces la frecuencia objetivo de muestreo de audio necesaria en una aplicación, donde M es el factor de diezmado.

$$\text{Frecuencia PDM} = \text{frecuencia de muestreo de audio} \times \text{factor de diezmado} \qquad (3)$$

El factor de diezmado está generalmente en el rango de 48 a 128.

La etapa de diezmado está precedida por un filtro de paso bajo para evitar la distorsión de *aliasing*.

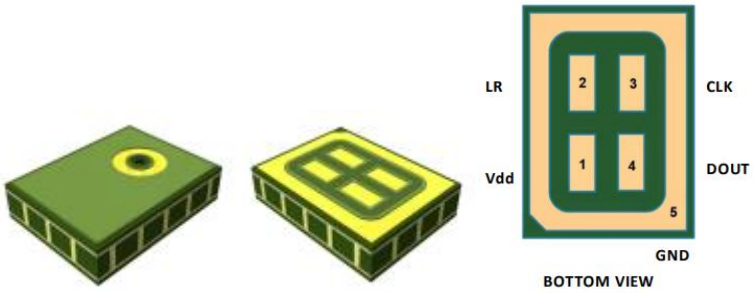
3.3. El micrófono MP34DT005-A

El micrófono empleado por el Sensor Tile es el MP34DT005-A. Es un micrófono MEMS digital omnidireccional, ultra compacto y de baja potencia construido con un elemento de detección capacitivo, una interfaz IC y proporciona a su salida una señal en protocolo PDM.

El elemento de detección, capaz de detectar ondas acústicas, se fabrica utilizando un proceso de micro mecanizado de silicio especializado dedicado a producir sensores de audio.

La interfaz IC se fabrica utilizando un proceso CMOS que permite diseñar un circuito dedicado capaz de proporcionar una señal digital externamente en formato PDM.

El MP34DT05-A es un micrófono digital de baja distorsión con una relación señal/ruido de 64 dB y una sensibilidad de –26 dBFS ±3 dB.



Pin #	Pin name	Function
1	Vdd	Power supply
2	LR	Left/Right channel selection
3	CLK	Synchronization input clock
4	DOUT	Left/Right PDM data output
5 (ground ring)	GND	Ground

Figura 50 Micrófono MP34DT005-A. [13]

Se utilizará una configuración del micrófono en modo monofónica como se muestra en la figura.

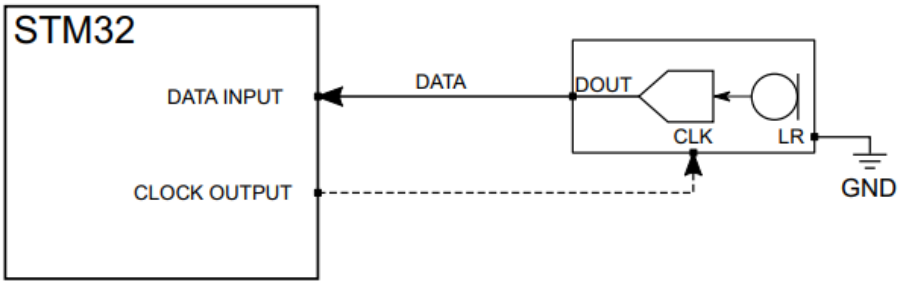


Figura 51 Configuración monofónica del micrófono. [13]

3.4. El DFSDM (Digital Filter Sigma Delta Modulator)

Los micrófonos MEMS digitales que proporcionan señal PDM se pueden conectar directamente al DFSDM. El DFSDM se encarga de muestrear, filtrar y diezmar la señal PDM proporcionada por el micrófono digital conectado.

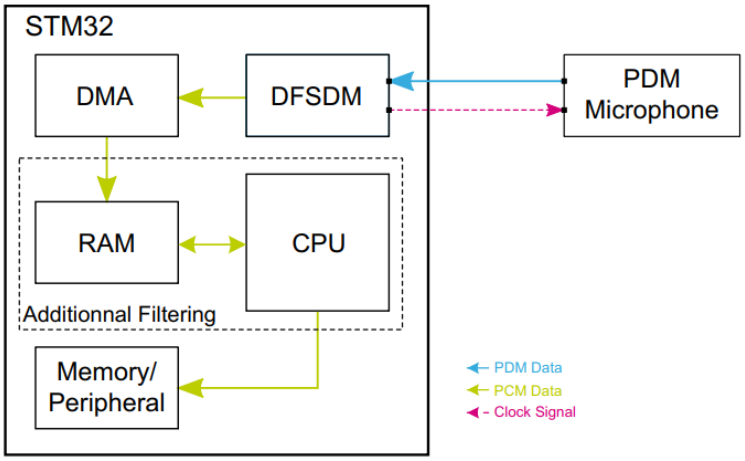


Figura 52 Diagrama de bloques básico de un microcontrolador, micrófono y DFSDM. [12]

El DFSDM es un periférico dentro de las MCU y MPU de los microcontroladores STM32. Se comporta como un ADC estándar con una velocidad/resolución escalable y un *front-end* analógico externo. Este periférico permite la conversión de señales analógicas a digitales con alta precisión y reducción de ruido.

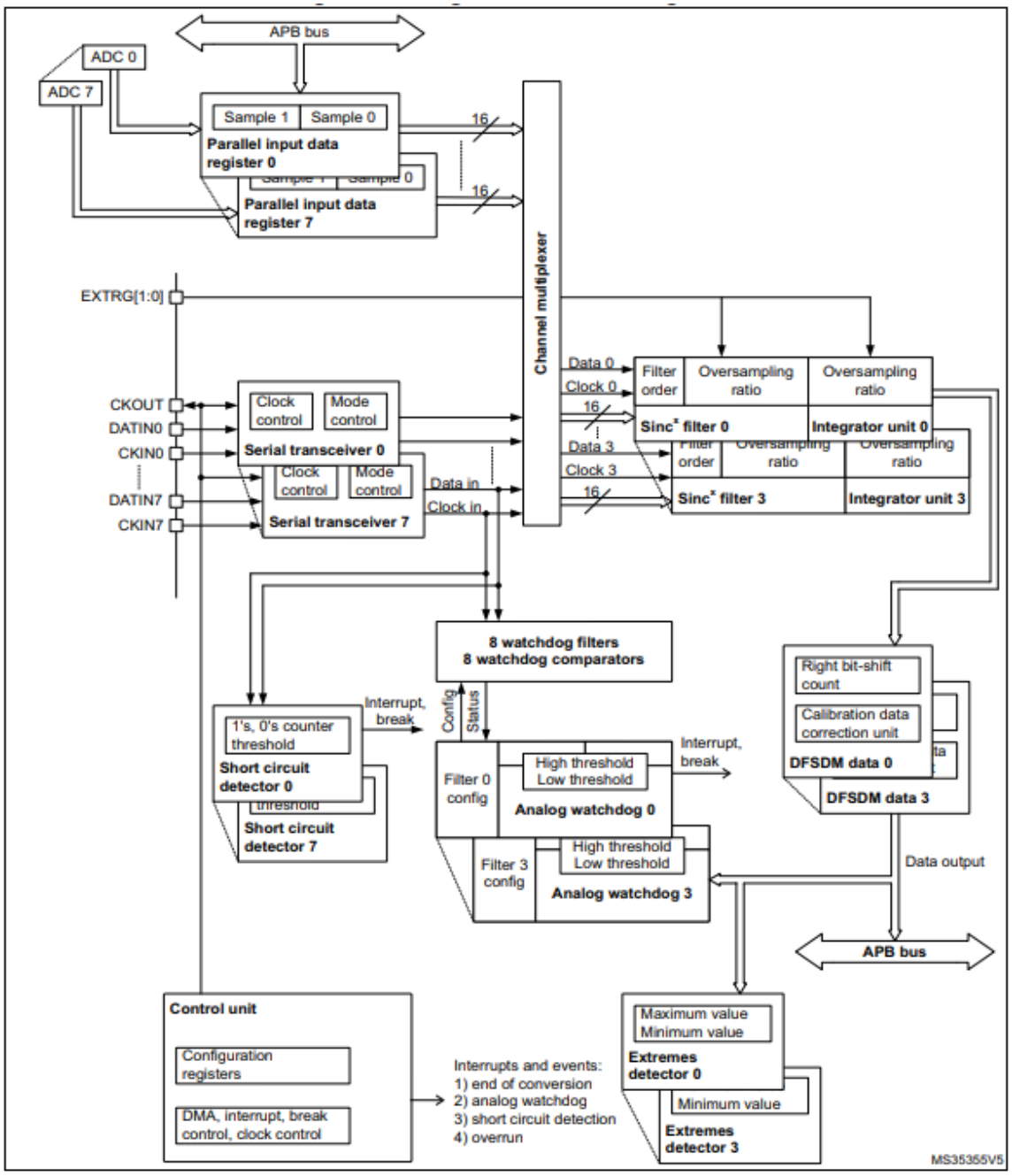
Para realizar la conversión de una señal analógica a digital, se utiliza un modulador sigma-delta el cual, sobre muestrea la señal analógica para convertir la señal en una secuencia de bits.

El DFSDM se caracteriza por tener varios canales de entrada de señales analógicas y filtros digitales independientes.

La resolución del DFSDM de conversión es de hasta 24 bits, que otorga una gran precisión en la digitalización de las señales analógicas. Además, el módulo DFSDM dispone de filtros digitales programables que permiten mejorar las características de la señal de entrada con parámetros como la ganancia, el tipo de filtro o la frecuencia de corte y además también permite el ajuste de la velocidad de conversión y su resolución.

Una de las grandes ventajas de utilizar el DFSDM es la transferencia directa al módulo de memoria DMA. De esta manera no interviene el procesador principal y se liberan así recursos, dando más tiempo al microcontrolador para ejecutar la lista de tareas que tenga que realizar.

A continuación, se muestra el diagrama de bloques del DFSDM.



1. This example shows 4 DFSDM filters and 8 input channels (max. configuration). Availability of the input from internal ADC

Figura 53 Diagrama de bloques del DFSDM.[14]

En el diseño que nos ocupa se utilizará una configuración monofónica del micrófono y se empleará el canal 5 del DFSDM porque el rutado del Sensor Tile lo lleva a ese canal.

El DFSDM cuenta con una señal de salida de reloj (DFSDM_CKOUT) para conectar al micrófono. El DFSDM dispone de un reloj especial para aplicaciones de audio, será el que utilice el micrófono para tomar las muestras de audio y para la comunicación con el DFSDM. Este reloj será el que se seleccionará para el diseño y se hará en el *clock management* del STM32CubeIde, seleccionando el PLLSAI1P (se verá en el apartado de *Validate on Target*).

La conexión con el micrófono será a través de PDM (*pulse density modulation*) y será el DFSDM la que la convierta a PCM.

El DFSDM permite dos tipos de conversiones de audio, la inyectada y la regular, también denominada "FAST". Las conversiones regulares tienen una prioridad más baja y pueden ser interrumpidas por una conversión inyectada. Si la conversión regular es interrumpida por una conversión inyectada, se reinicia una vez que la conversión inyectada haya terminado, y esta interrupción se indica como una bandera para esta conversión regular retrasada. El caso que ocupa a este Trabajo Fin de Grado no hay conversiones inyectadas.

Las conversiones regulares solo pueden ser iniciadas por software, pueden ejecutarse en modo continuo, en el cual no hay cambio de canales, y pueden realizarse en modo rápido sin rellenado de filtro.

Las conversiones regulares también se utilizan típicamente para conversiones continuas de un solo canal, por ejemplo, en aplicaciones de audio como es el caso que ocupa a este Trabajo Fin de Grado, además hay que recordar que la configuración del micrófono es tipo mono, por tanto, de un solo canal.

El DFSDM permite realizar ajustes durante el procesamiento digital de la señal ya que contiene una implementación de filtro digital de tipo Sincx. Este filtro Sincx realiza un filtrado de una secuencia de datos digitales de entrada, lo que resulta en la disminución de la tasa de datos de salida (decimación) y el aumento de la resolución de los datos de salida. El filtro digital Sincx (hay disponibles cinco) es configurable para alcanzar las tasas de datos de salida y la resolución de datos de salida requeridas (desde 1 a 1024). La configuración en el caso que nos ocupa se realizará a través del parámetro Fosr que va desde 1 a 215 y proporciona un filtro Sinc de orden 4.

La función de transferencia del filtro se calcula así:

$$H(z) = \left(\frac{1-z^{-FOSR}}{1-z^{-1}}\right)^2 (1 + z^{-(2\,FOSR)}) \qquad (4)$$

Otros aspectos a tener en cuenta en el DFSDM es el integrador, que no será utilizado. El integrador realiza una decimación adicional y un aumento en la resolución de los datos provenientes del filtro digital. El integrador simplemente realiza la suma de los datos del filtro digital durante un número determinado de muestras de datos del filtro.

El parámetro de la razón de sobre muestreo del integrador define cuántos conteos de datos se sumarán para generar una salida de datos desde el integrador. La razón de sobre muestreo del integrador (Iosr, por sus siglas en inglés) se puede configurar en el rango de 1 a 256. Será configurado a valor 1 para puentearlo.

Durante la adquisición de datos se va a utilizar un *buffer* circular. El funcionamiento del *buffer* circular es el siguiente, se utiliza un *buffer* del doble de tamaño que la muestra, el DMA irá llenando este *buffer* empezando desde el principio. La interrupción se activará, y se hará cuando llegue a la mitad del *buffer* o al final del *buffer*. Cuando se active la interrupción de mitad de *buffer*, se procesará el *frame* 1 y se continúa escribiendo el *frame* 2.

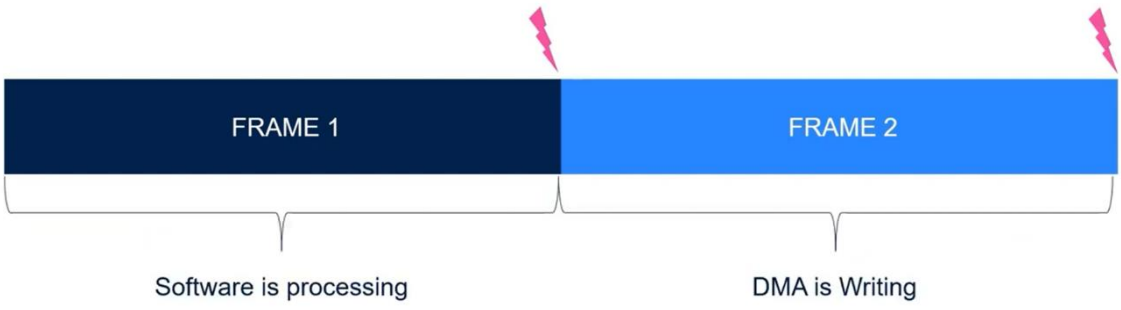


Figura 54 Buffer circular 1.[15]

Cuando el segundo *buffer* se llene, se activará la interrupción de llenado y el DMA procesará el *frame* 2 y continúa escribiendo sobre el *frame* 1. Resaltar que con las conversiones continuas que se van a utilizar el *flag* de interrupción se limpia de manera automática.

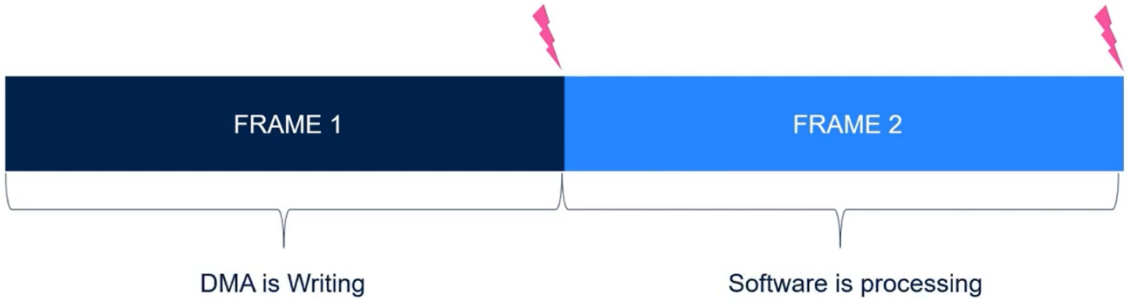


Figura 55 Buffer circular 2. [15]

Por tanto, el uso de un *buffer* circular en una adquisición por DMA permite una transferencia continua de datos, ya que el DMA puede seguir almacenando datos en el *buffer* mientras el *firmware* procesa los datos previamente adquiridos.

De esta manera y de forma continua se va procesando el audio que se adquiere a través del micrófono.

4. DISEÑO DE LA RED NEURONAL

A continuación, se empieza a describir todo el proceso de diseño de la red neuronal. Esta red neuronal será capaz de hacer una selección de escenas de audio. La red neuronal trabajará sobre el estado de un ventilador, distinguiendo entre los estados apagado, encendido y obstruido.

Los pasos que componen todo el proceso de diseño de la red neuronal que haga la selección de escenas de audio es el que se ve en la figura siguiente.

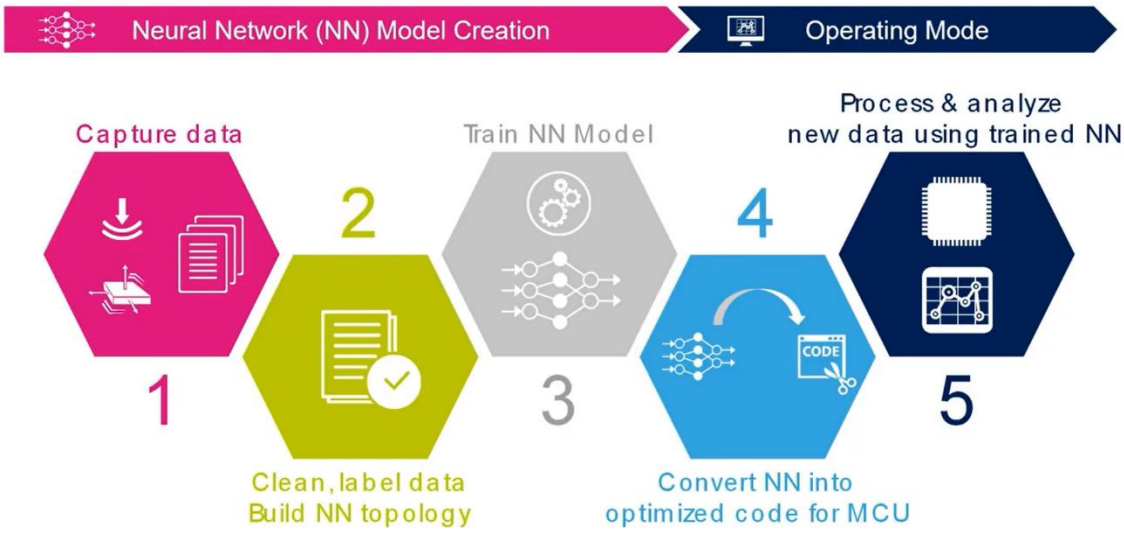


Figura 56 Proceso completo de diseño de una red neuronal en un microcontrolador STM32[16]

Por tanto los pasos son:

1. Hacer varias capturas de audio.
2. Hacer una limpieza de los datos capturados y construir la red neuronal.
3. Entrenar la red neuronal.
4. Convertir el código hecho en Python a código C entendible por el microcontrolador.
5. Procesar los datos nuevos utilizando la red entrenada.

El paso 1 se realiza a través del código DataLog, que es un código que proporciona el paquete de *firmware* de ST STSW-STLKT01_V2.5.0. Este código permite grabar en la tarjeta microSD del Sensor Tile una vez que se le da un "doble *tap*" a su carcasa.

A través de este *firmware* se graban 3 audios de duración aproximada 35 segundos:

1. Con el ventilador apagado.
2. Con el ventilador encendido.
3. Con el ventilador obstruido.

El audio grabado tiene una frecuencia de 16 KHz y es mono. Es muy importante recalcar que las características del audio de grabación son las que se han de tener en la configuración del Sensor Tile a la hora de escuchar las situaciones en las que se encuentra el ventilador, ya que la red está entrenada con las características de los audios grabados con el código DataLog.

A modo de resumen gráfico, todo el proceso de la red neuronal se resume en esta imagen:

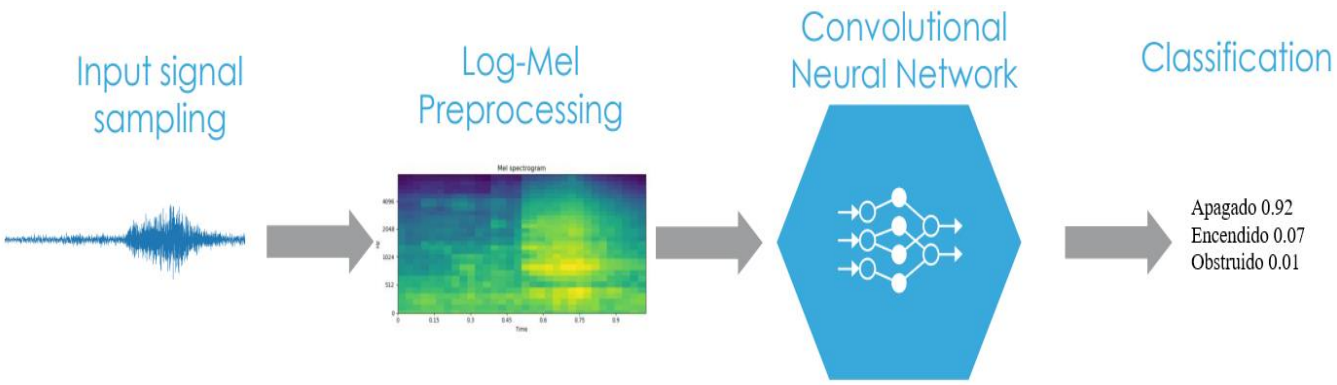


Figura 57 Proceso de funcionamiento de la red neuronal cargada en un microcontrolador STM32[17]

Se puede ver como el proceso es que, ante un audio recibido a través del micrófono, se convierte en un espectrograma Log-Mel y se introduce en la red neuronal convolucional, la cual, hará una clasificación probabilística para identificar con que estado (apagado, encendido u obstruido) se corresponde el audio introducido.

Puesto que el script utilizado para diseñar la red neuronal se basa en la utilización de Keras y Tensor Flow, se van a explicar más en profundidad estas dos herramientas típicas en el diseño de redes neuronales.

4.1. Keras

Keras es una biblioteca de código abierto (con licencia MIT) escrita en Python, que se basa principalmente en el trabajo de François Chollet, un desarrollador de Google, en el marco del proyecto ONEIROS (*Open-ended Neuro-Electronic Intelligent Robot Operating System)*. El objetivo de la biblioteca es acelerar la creación de redes

neuronales: para ello, Keras no funciona como un *framework* independiente, sino como una interfaz de uso intuitivo (API) que permite acceder a varios *frameworks* de aprendizaje automático y desarrollarlos. Entre los *frameworks* compatibles se encuentra TensorFlow.

Keras es una biblioteca que funciona a nivel de modelo: proporciona bloques modulares sobre los que se pueden desarrollar modelos complejos de aprendizaje profundo. A diferencia de los *frameworks*, este software de código abierto no se utiliza para operaciones sencillas de bajo nivel, sino que utiliza las bibliotecas de los *frameworks* de aprendizaje automático vinculadas, que en cierto modo actúan como un motor de *backend* para Keras. Las capas de la red neuronal que se quieren configurar se relacionan entre sí de acuerdo con el principio modular, sin que el usuario de Keras tenga que comprender o controlar directamente el propio *backend* del *framework* elegido.

Keras dispone de interfaces listos para usar que permiten un acceso rápido e intuitivo al *backend* de TensorFlow.

La biblioteca de código abierto Keras supone una excelente aportación a las herramientas que sirven para crear redes neuronales, ya que simplifica muchísimo este proceso. En este caso, la atención se centra sobre todo en mejorar la usabilidad: Keras funciona como una interfaz diseñada expresamente para personas y solo secundariamente para máquinas. Las acciones del usuario necesarias para los casos de uso más importantes se reducen al mínimo y, si aun así se producen errores, se proporciona un *feedback* que ayuda a resolverlos.

Keras también presenta las siguientes ventajas:

- Amplia compatibilidad entre plataformas para los modelos desarrollados: los modelos desarrollados con Keras son especialmente fáciles de utilizar en diferentes plataformas. Por ejemplo, son compatibles con iOS (Apple CoreML), Android (Keras TensorFlow Android Runtime), Google Cloud y Raspberry Pi por defecto.

- Compatibilidad con múltiples motores de *backend*: Keras no solo da mucha libertad a la hora de elegir el *backend*, sino que también permite combinar varios motores. Además, es posible transferir los modelos desarrollados a otro *backend* en cualquier momento.

- Excelente soporte para múltiples GPU: con Keras, los recursos necesarios para desarrollar los procesos de aprendizaje profundo se pueden distribuir fácilmente en varios chips o tarjetas gráficas.

- Desarrollo por parte de grandes empresas: el mantenimiento y el desarrollo de Keras cuentan con el apoyo de las empresas más importantes del sector. Google, Amazon AWS, Microsoft, Apple y Nvidia, entre otras, están implicadas en el proyecto.

4.2. TensorFlow

Creado por el equipo de Google Brain, TensorFlow es una biblioteca de código abierto para la computación numérica y *Machine Learning* a gran escala. TensorFlow reúne una serie de modelos y algoritmos de *Machine Learning* y *Deep Learning*.

Utiliza Python para proporcionar una práctica API para crear aplicaciones con el marco de trabajo, a la vez que ejecuta esas aplicaciones en C++.

TensorFlow puede entrenar y ejecutar redes neuronales profundas para la clasificación de dígitos escritos a mano, el reconocimiento de imágenes, la incrustación de palabras, las redes neuronales recurrentes, los modelos secuencia a secuencia para la traducción automática, el procesamiento del lenguaje natural y las simulaciones

basadas en ecuaciones diferenciales parciales. Lo mejor de todo es que TensorFlow admite predicción de producción a escala, con los mismos modelos utilizados para el entrenamiento.

TensorFlow permite a los desarrolladores crear gráficos de flujo de datos, estructuras que describen cómo los datos se mueven a través de un gráfico, o una serie de nodos de procesamiento. Cada nodo del gráfico representa una operación matemática, y cada conexión o arista entre nodos es una matriz de datos multidimensional, o tensor.

TensorFlow proporciona todo esto al programador a través del lenguaje Python. Los nodos y tensores en TensorFlow son objetos de Python, y las aplicaciones de TensorFlow son a su vez aplicaciones de Python.

Las operaciones matemáticas reales, sin embargo, no se realizan en Python. Las bibliotecas de transformaciones que están disponibles a través de TensorFlow están escritas como binarios C++ de alto rendimiento. Python solo dirige el tráfico entre las piezas, y proporciona abstracciones de programación de alto nivel para conectarlas entre sí.

Las aplicaciones de TensorFlow se pueden ejecutar en casi cualquier objetivo que sea conveniente: una máquina local, un clúster en la nube, dispositivos IOS y Android, CPUs o GPUs.

El mayor beneficio que ofrece TensorFlow para el desarrollo de *Deep Learning* es la abstracción. En lugar de ocuparse de los detalles de implementación de los algoritmos, o de averiguar la forma adecuada de conectar la salida de una función con la entrada de otra, el desarrollador puede centrarse en la lógica general de la aplicación.

4.3. Jupyter Notebook

El *script* donde se ha diseñado la red neuronal se ha hecho en Python y se ejecutará sobre Jupyter Notebook.

Jupyter Notebook es una herramienta de código abierto utilizada para desarrollar y compartir código en diferentes lenguajes de programación, incluyendo Python. Permite la integración de texto, código, gráficos y otros elementos multimedia en un documento interactivo, lo que lo convierte en una herramienta muy útil.

Jupyter Notebook, puede escribir y ejecutar código en celdas individuales, lo que permite explorar y analizar datos de manera interactiva. Además, puede agregar texto explicativo, ecuaciones matemáticas, imágenes y visualizaciones para crear un documento coherente y fácil de entender.

Los documentos, llamados notebooks, se organizan en celdas, que pueden ser de diferentes tipos: celdas de código, de texto, de *Markdown*, de ecuaciones, etc.

Cuando se ejecuta una celda de código, se muestra el resultado de la operación directamente debajo de la celda. Esto permite una interacción rápida y dinámica con el código y los datos,

Jupyter Notebook también cuenta con una amplia variedad de herramientas y funciones que facilitan el trabajo con datos y código. Por ejemplo, es posible importar bibliotecas de Python y otros lenguajes de programación para trabajar con datos, visualizar los datos en gráficos interactivos, exportar los notebooks a diferentes formatos, etc.

Figura 58 Árbol de archivos de Jupyter Notebook

El interfaz básico de ejecución de un notebook se puede ver bajo estas líneas:

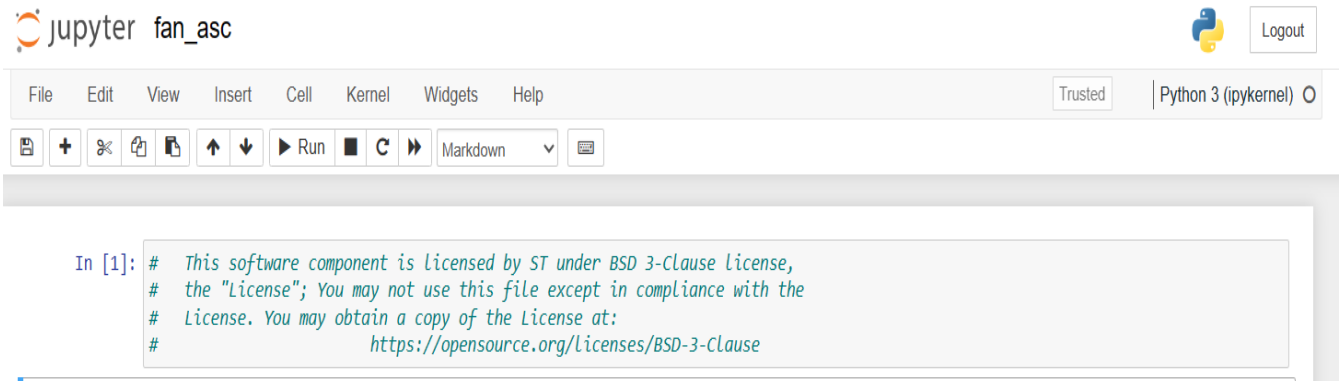


Figura 59 Interfaz básico de Jupyter Notebook

Para ir ejecutando el código hay que ir pulsando el botón Run.

4.4. Script en Python para diseñar la Red Neuronal

La inteligencia artificial se puede utilizar para clasificar el audio capturado por el micrófono en un dispositivo *IoT*. Mediante el siguiente código se crea una red convolucional con *Deep Learning* para clasificar según el audio capturado en tres categorías:

- Ventilador apagado.
- Ventilador encendido.
- Ventilador obstruido.

El código comienza de la siguiente manera:

1. Importar la biblioteca TensorFlow, que está explicada anteriormente.
2. Importar las librerías Numpy, que es una librería especializada en tratamiento y manipulación de datos con fórmulas matemáticas.
3. Importar la librería matplotlib para mostrar gráficos y visualizaciones de datos.
4. Importar el módulo os para tener acceso a archivos y sus rutas.
5. Importar la librería librosa, la cual permitirá el tratamiento de las señales de audio y también la visualización de espectrogramas y formas de onda.
6. Importar la librería sklearn para para hacer análisis predictivo, precisión de modelos de clasificación y también incluye clasificadores de datos (entrenamiento y prueba).

7. Importar Keras para definir capas de redes neuronales y modelos entre otras funciones.

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import os
from tqdm import tqdm
import librosa
import librosa.display
import librosa.util
from IPython.display import Audio
from sklearn import preprocessing
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
import keras.utils
from keras import layers
from keras import models
from keras import optimizers
```

Figura 60 Importación de librerías y módulos

También se corrobora las versiones instaladas de Keras y TensorFlow, así como de la librería librosa.

```
#Verificamos las versiones de keras, tensorflow y librosa
print("Keras:", keras.__version__)
print("TensorFlow:",tf.__version__)
print("librosa:", librosa.__version__)
```

```
Keras: 2.10.0
TensorFlow: 2.10.0
librosa: 0.9.2
```

Figura 61 Versiones utilizadas

El siguiente paso es importar el *dataset* de audios ubicado en la carpeta *Dataset*. El *dataset* debe contener un archivo .txt donde en cada línea tiene el nombre del arhivo.wav un espacio y el nombre de la etiqueta a usar en el set de entrenamiento. Es decir, de esta manera:

apagado. wav apagado

encendido.wav encendido

obstruido.wav obstruido

Se definen los tres estados que reconocerá la red neuronal que serán apagado (*fan_off*), encendido (*fan_on*) y obstruido (*fan_obstruct*) y se asigna una etiqueta con un número entero a cada clase.

El siguiente paso es hacer la carga del *dataset* y se hacen las siguientes operaciones de procesamiento:

- Se resamplean a 16kHz.
- Se convierten a monofónico.
- Se limita su duración a 30 segundos.
- Se convierten los valores a float32.

Por último, almacena las señales de audio y sus etiquetas en las listas “x” e “y”.

Hay que tener en cuenta que en las señales de audio, tanto el *sampling rate* como el número de canales y sus ajustes han de coincidir con la configuración del programa DataLog con que se hizo su grabación. Por ello el micrófono y el DFSDM se han de configurar de acorde a las características del *dataset*.

```
dataset_dir = './Dataset'
meta_path = path = os.path.join(dataset_dir, 'TrainSet.txt')
fileset = np.loadtxt(meta_path, dtype=str)

# 3 classes : 0 fan_off, 1 fan_on, 2 fan_obstruct
class_names = ['fan_off', 'fan_on', 'fan_obstruct']
labels = {
    'fan_obstruct' : 2,
    'fan_on' : 1,
    'fan_off' : 0,
}

x = []
y = []

for file in tqdm(fileset):
    file_path, file_label = file
    file_path = os.path.join(dataset_dir, file_path)
    ##note: resampling can take some time!
    signal, _ = librosa.load(file_path, sr=16000, mono=True, duration=30, dtype=np.float32)
    label = labels[file_label]
    x.append(signal)
    y.append(label)
```

```
100%|██████████| 3/3 [00:00<00:00, 104.05it/s]
```

Figura 62 Carga del dataset y operaciones sobre el mismo

Finalmente se muestran los espectros de las tres señales de audio cargadas utilizando la biblioteca Matplotlib a una frecuencia de muestreo de 16 Khz.

```
#plot first resampling audio input
plt.figure(figsize=(12, 2))
plt.title(class_names[y[2]])
librosa.display.waveshow(x[2], sr=16000)
Audio(x[2], rate=16000)

plt.figure(figsize=(12, 2))
plt.title(class_names[y[1]])
librosa.display.waveshow(x[1], sr=16000)
Audio(x[2], rate=16000)

plt.figure(figsize=(12, 2))
plt.title(class_names[y[0]])
librosa.display.waveshow(x[0], sr=16000)
Audio(x[2], rate=16000)
```

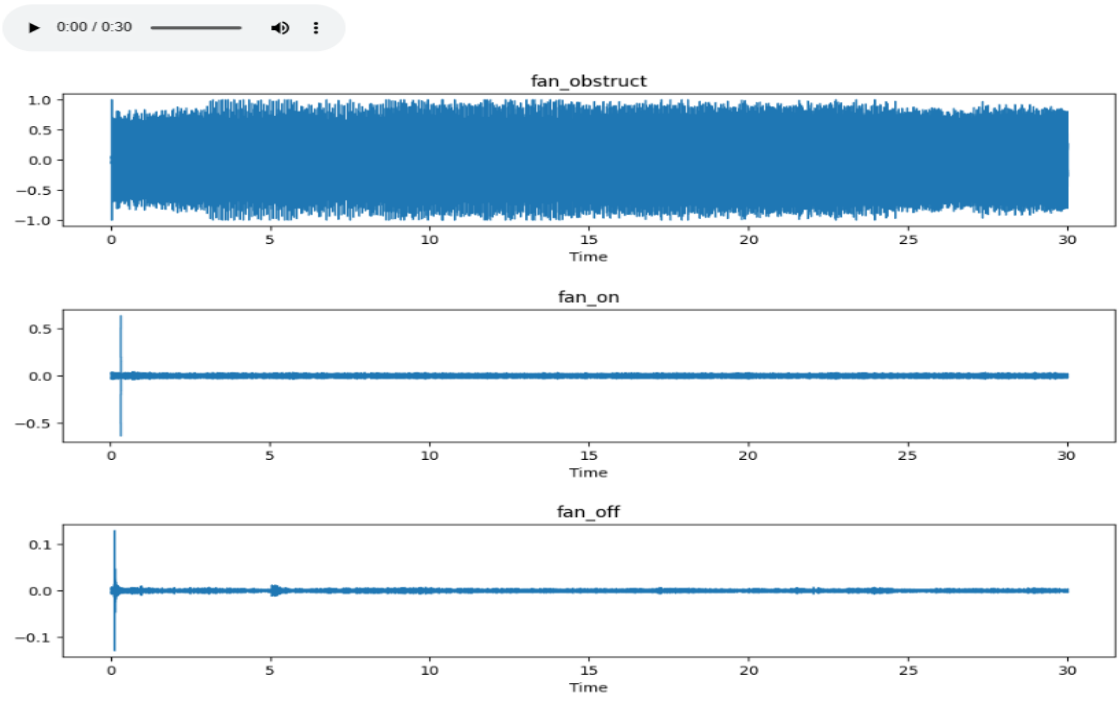


Figura 63 Espectros de señales de audio

Preparación de los datos

En lugar de alimentar directamente al modelo de red neuronal, los datos son previamente tratados y cortados en pequeños *subframes* para crear un conjunto de espectrogramas LogMel con unas características determinadas. Estos espectrogramas se utilizarán para entrenar el modelo, su validación y su testeo.

La duración de los audios se ha limitado a 30 segundos. Se decide tomar 64 milisegundos de duración por *frame* que compone la señal de audio (decisión tomada por los expertos en *Digital Signal Processing* de ST).

Al dividir 30 segundos entre 64 milisegundos se obtiene como resultado 468.75 *frames*, que serán redondeados a 468 *frames*.

Para evitar la discontinuidad de la señal y la pérdida de información cuando se realice la transformada rápida de Fourier para pasar del dominio del tiempo al dominio de la frecuencia, se solaparán los *frames* por tanto hay que multiplicar por dos esos 468 *frames*, obteniendo un total de 936 *frames*.

Cada *frame* es muestreado con 1024 muestras.

Por tanto la dinámica es tomar la mitad de un *frame* de 64 ms, procesarlo y pasarlo al dominio de la frecuencia e ir conformando columna a columna un espectrograma. De una manera más gráfica se puede ver en la siguiente figura.

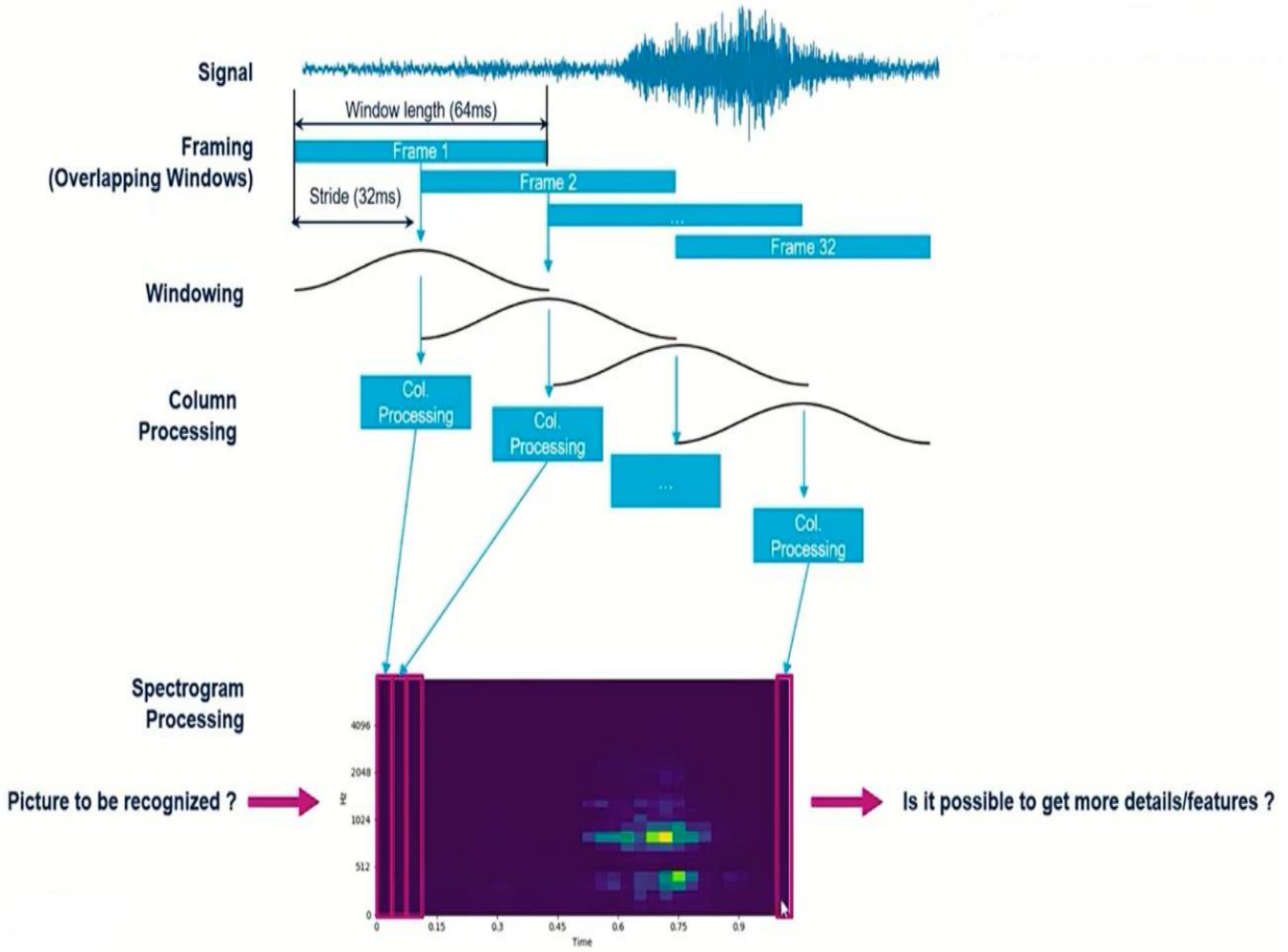


Figura 64 Preparación de los datos en espectrograma[17]

Cortar datos en *frames*

Cada *frame* contiene 16.896 muestras (32 * 512 + 512) para crear un espectrograma de 32 columnas con una longitud de ventana de 1024 (n_fft=1024) y una longitud de salto de 512 muestras (*hop_length*=512) para la transformada rápida de Fourier FFT.

La longitud de ventana n_fft es el número de muestras en cada ventana (columna) del espectrograma.

La longitud del salto se refiere al número de muestras entre *frames* sucesivos. Para el análisis de la señal, el tamaño del salto debe ser menor que el tamaño del *frame*, de modo que los *frames* se superpongan.

```
x_framed = []
y_framed = []

for i in range(len(x)):
    frames = librosa.util.frame(x[i], frame_length=16896, hop_length=512)
    x_framed.append(np.transpose(frames))
    y_framed.append(np.full(frames.shape[1], y[i]))

# merge sliced frames and label
x_framed = np.asarray(x_framed)
y_framed = np.asarray(y_framed)
x_framed = x_framed.reshape(x_framed.shape[0]*x_framed.shape[1], x_framed.shape[2])
y_framed = y_framed.reshape(y_framed.shape[0]*y_framed.shape[1], )

print("x_framed shape: ", x_framed.shape) # Each frame of 16,896 samples can be used to create spectrogram
print("y_framed shape: ", y_framed.shape) # Corresponding label for each frame
```

```
x_framed shape:  (2715, 16896)
y_framed shape:  (2715,)
```

Figura 65 Separación del audio en tramas

Este fragmento de código separa el audio en tramas más pequeñas para su análisis y extracción de características. A través de la librería librosa, separa el audio en tramas de longitud 16896 muestras y un desplazamiento entre tramas adyacentes de 512 muestras.

El código guarda las tramas en una lista y le asocia a una etiqueta, es decir, asocia cada trama a si el ventilador se encuentra apagado, encendido u obstruido.

Por último, las listas x_frame e y_framed se transforman en matrices NumPy para facilitar su operación haciéndolas de una dimensión. Con estos pasos se consigue tener un conjunto de tramas de datos listo para su entrenamiento.

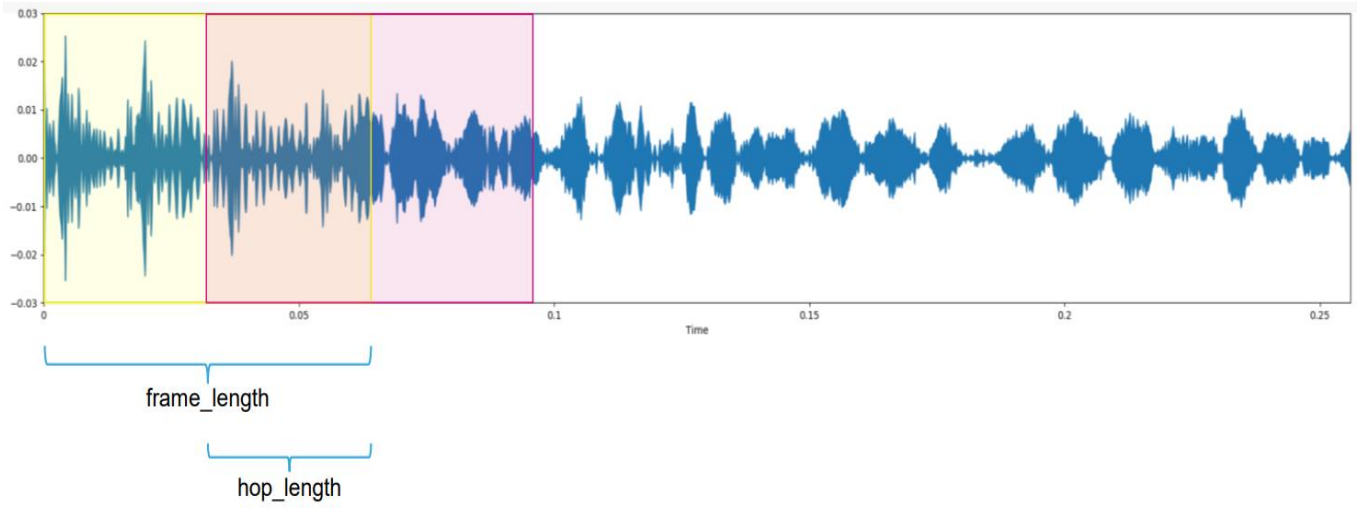


Figura 66 Longitud del frame y salto[17]

Preprocesamiento de datos para crear los espectrogramas LogMel

El siguiente paso es preprocesar las muestras de las tramas de audio y calcular el espectograma LogMel. Esto se hace porque el ser humano es mucho mejor discerniendo pequeños cambios en tonos percibidos en bajas frecuencias que a altas frecuencias, es decir, la 'distancia' percibida entre 300 Hz y el tono de 500Hz parece ser mas grande que entre 3200Hz y 3400 Hz. Se puede encontrar una relación entre percepción y frecuencia medida con la función logarítmica.

$$m = 2595\, log10\left(1 + \frac{f}{700}\right) \qquad (5)$$

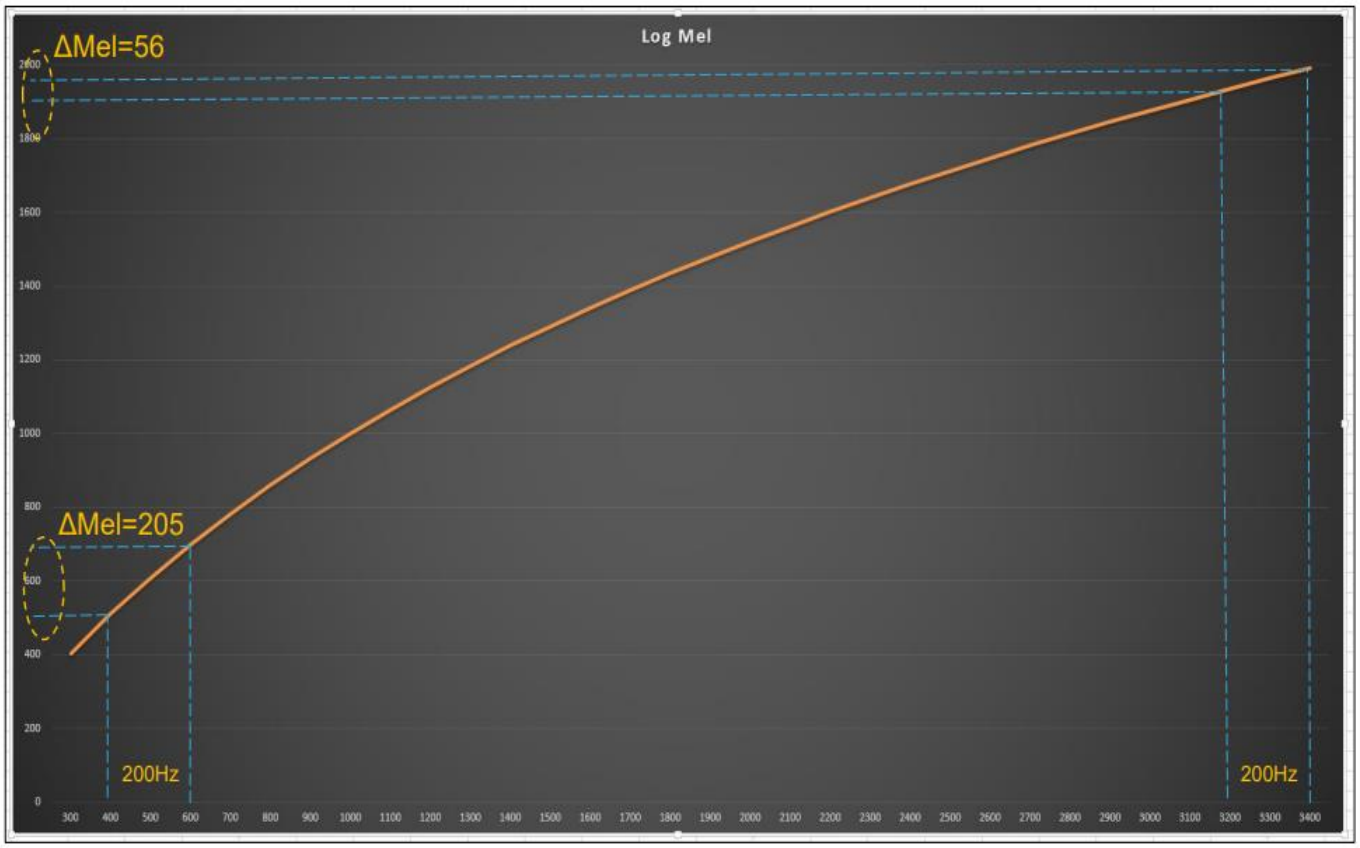


Figura 67 Escala Log Mel[17]

Para calcular el espectrograma LogMel hay que especificar la frecuencia de muestreo (16 KHz), el número de columnas del espectrograma de Mel (30 columnas), el número de muestras en la Transformada de Fourier (1024) y la distancia entre saltos que es de 512 muestras.

También se realiza una normalización del espectrograma al dividirlo por su valor máximo. Esto se hace para:

- Equilibrar las características extraídas del espectrograma y así evitar el dominio de unas características sobre otras.
- Limitar sus valores, los cuales estarán entre 0 y 1, y así evitar posibles desbordamientos en el proceso de cálculo.
- Obtener una generalización del modelo, lo cual ayuda a la red neuronal en su aprendizaje y puede mejorar su rendimiento.

La figura siguiente a modo de ejemplo muestra cómo se haría la convergencia con datos sin normalizar y con datos normalizados. Se ve claramente como el gradiente converge más rápido con los datos normalizados.

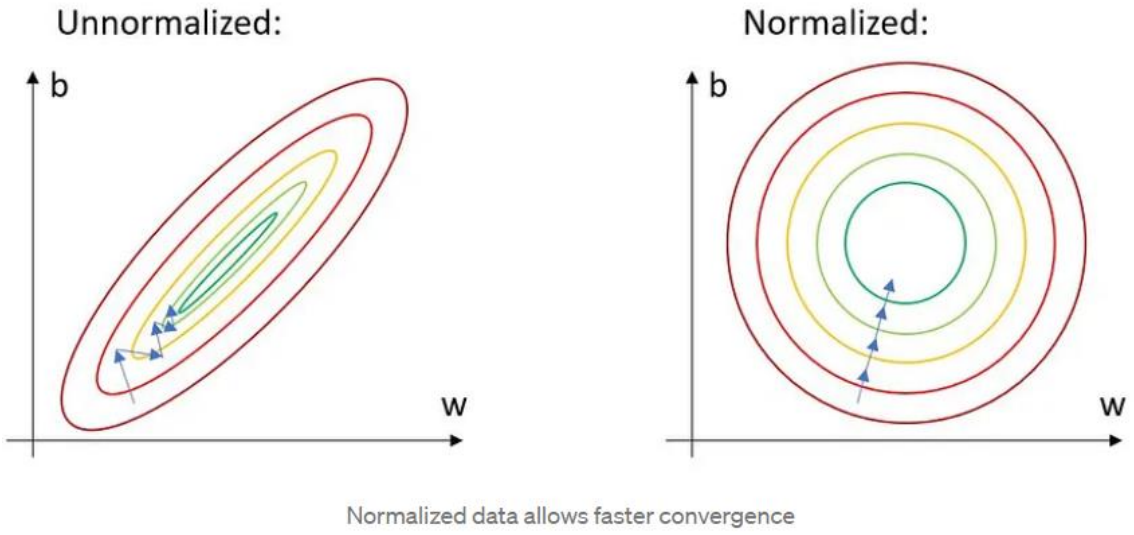


Figura 68 Diferencia en la convergencia entre datos normalizados y sin normalizar.[18]

El penúltimo paso es añadir el valor de intensidad de la señal en base logarítmica, siendo el tope -80 dB.

Finalmente se transforma a una matriz NumPy para facilitar el manejo de los datos de la matriz.

```
x_features = []
y_features = y_framed

for frame in tqdm(x_framed):
    # Create a mel-scaled spectrogram
    S_mel = librosa.feature.melspectrogram(y=frame, sr=16000, n_mels=30, n_fft=1024, hop_length=512, center=False)
    # Scale according to reference power
    S_mel = S_mel / S_mel.max()
    # Convert to dB
    S_log_mel = librosa.power_to_db(S_mel, top_db=80.0)
    x_features.append(S_log_mel)

# Convert into numpy array
x_features = np.asarray(x_features)
```

```
100%|██████████████████████████████████████████████████████████| 2715/2715 [00:03<00:00, 723.63it/s]
```

Obtenemos 2.715 características en el conjunto, con cada característica representada con un espectrograma de 30x32.

Figura 69 Cálculos para obtener espectrograma LogMel

Por tanto, se obtiene un conjunto con 2715 características representadas en espectrogramas de 30 filas x 32 columnas.

```
In [21]: print(x_features.shape)

         (2715, 30, 32)
```

Figura 70 Conjunto de características

Lo siguiente es imprimir los tres espectrogramas para cada tipo de estado que va a reconocer la red neuronal, es decir, entre apagado, encendido y obstruido.

```
# Plot the first spectrogram generated for each feature class

plt.figure(figsize=(10, 8))
plt.subplot(311)
indoor_index = np.argmax(y_framed == 0)
librosa.display.specshow(x_features[indoor_index], sr=16000, y_axis='mel', fmax=8000,
                         x_axis='time', cmap='viridis', vmin=-80.0)
plt.colorbar(format='%+2.0f dB')
plt.title('LogMel spectrogram for ' + class_names[y_features[indoor_index]])

plt.subplot(312)
outdoor_index = np.argmax(y_framed == 1)
librosa.display.specshow(x_features[outdoor_index], sr=16000, y_axis='mel', fmax=8000,
                         x_axis='time', cmap='viridis', vmin=-80.0)
plt.colorbar(format='%+2.0f dB')
plt.title('LogMel spectrogram for ' + class_names[y_features[outdoor_index]])

plt.subplot(313)
vehicle_index = np.argmax(y_framed == 2)
librosa.display.specshow(x_features[vehicle_index], sr=16000, y_axis='mel', fmax=8000,
                         x_axis='time', cmap='viridis', vmin=-80.0)
plt.colorbar(format='%+2.0f dB')
plt.title('LogMel spectrogram for ' + class_names[y_features[vehicle_index]])

plt.tight_layout()
plt.show()
```

Figura 71 Impresión de los 3 espectrogramas

El eje Y del espectrograma representa las frecuencias, el eje X el tiempo y los colores representan el nivel de intensidad, siendo el rango de 0 a -80 dB.

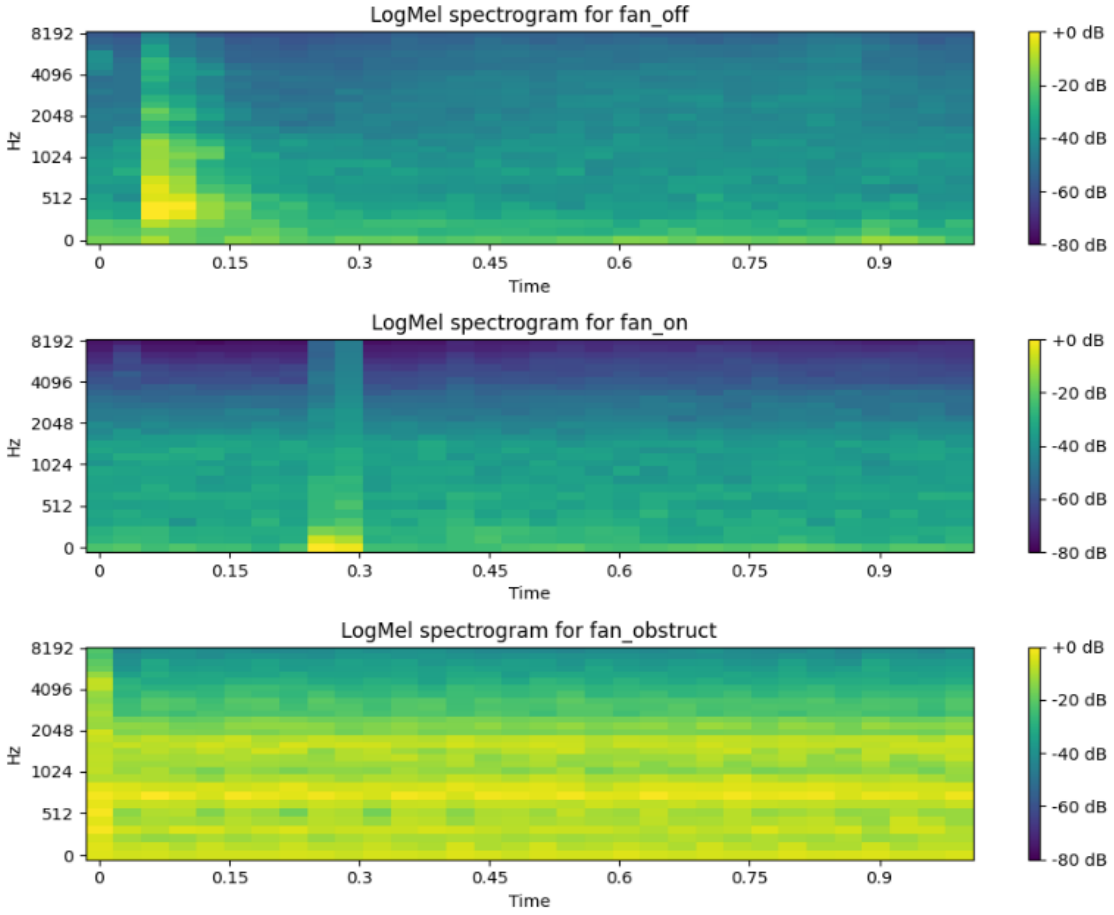


Figura 72 Espectrogramas de los 3 estados del ventilador

Como se puede apreciar en los tres espectrogramas, la información está dispersa y difusa. Lo que se requiere para procesar y mejorar la precisión en la red neuronal es concentrarla, con ello la red neuronal analizará mejor la información. Para ello se realizará una estandarización de los espectrogramas.

Estandarización de características

Con una estandarización de los datos, el gradiente converge más rápido hacía el mínimo global o local de la función de pérdida, se evitan fluctuaciones y obteniendo antes el resultado. Además, se evita la saturación de los datos durante el procesamiento de los mismos por parte del microcontrolador.

La estandarización de las características se calcula de la siguiente manera:

$$z = \frac{(x-u)}{s} \qquad (6)$$

Donde "x" es el valor de la muestra a calcular, "u" es la media de las muestras de entrenamiento y "s" es la desviación estándar de las muestras de entrenamiento.

La media (promedio) de un conjunto de datos se encuentra al **sumar todos los números en el conjunto de datos y luego al dividir entre el número de valores en el conjunto**.

La desviación estándar es una **medida de la dispersión de los datos**, cuanto mayor sea la dispersión mayor es la desviación estándar, si no hubiera ninguna variación en los datos, es decir, si fueran todos iguales, la desviación estándar sería cero.

```
# Flatten features for scaling
x_features_r = np.reshape(x_features, (len(x_features), 30 * 32))

# Create a feature scaler
scaler = preprocessing.StandardScaler().fit(x_features_r)

# Apply the feature scaler
x_features_s = scaler.transform(x_features_r)

#calculamos la media y la varianza
featureScalerMean = np.mean(x_features_r, axis=0)
featureScalerStd = np.std(x_features_r, axis=0)

#exportamos los datos para ser usados en el codigo STM32
feature_dir = './Dataset/FeatureSet/'
np.savetxt(feature_dir + 'featureScalerMean.csv', featureScalerMean, delimiter=",")
np.savetxt(feature_dir + 'featureScalerStd.csv', featureScalerStd, delimiter=",")
```

Figura 73 Estandarización de características

Por último, se exportan los cálculos obtenidos de media y desviación típica en unos archivos .csv para utilizarlos con el código que se cargará en el microcontrolador.

Preparación de los datos de salida

Cada característica tiene una etiqueta que lo identifica. Keras requiere una etiqueta tipo *one-hot*, que es una etiqueta del tipo binaria. Se realiza una conversión de vectores a matrices binarias. En la matriz cada fila es una muestra y los tipos de estados en los que se encuentra el ventilador son las columnas. Se distingue entre estados colocando un uno en la columna correspondiente y ceros en el resto de columnas.

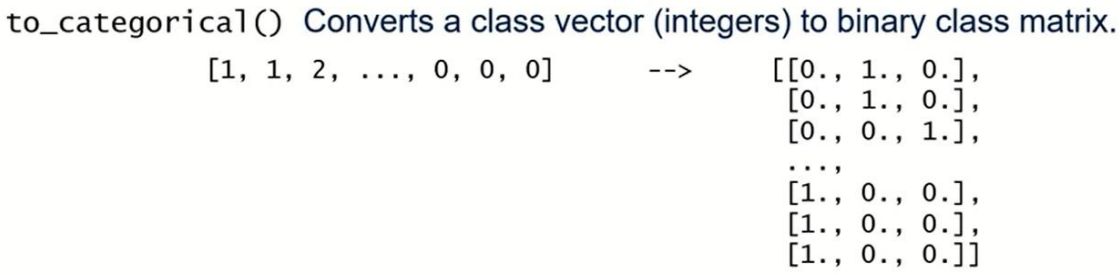


Figura 74 Conversión de vectores a matrices.[17]

```
# Convert labels to categorical one-hot encoding
y_features_cat = keras.utils.to_categorical(y_features, num_classes=len(class_names))

print("y_features_cat shape: " ,y_features_cat.shape)
```

```
y_features_cat shape:  (2715, 3)
```

Figura 75 Conversión a matrices binarias

Separación del *dataset* en un conjunto de datos de entrenamiento, validación y test

En todo proceso de aprendizaje automático, los datos se deben dividir en datos de entrenamiento, datos de validación y datos test para evaluar la pérdida y otras métricas del modelo al final de cada época de entrenamiento. El objetivo es desarrollar y ajustar el modelo utilizando solo los datos de entrenamiento y validación. El conjunto de datos de pruebas solo se utilizará para la evaluación final como si fuera desconocido, es decir, como si fueran nuevos datos. Por tanto, la función de cada conjunto de datos es la siguiente:

1. Entrenamiento: se utilizan para actualizar los parámetros de cada neurona dentro de la red como el peso, el *bias* y sus conexiones, esto mediante la función de pérdida por regresión lineal.

2. Validación: comprueba la capacidad de generalización de la red en cada época. Prueba con muestras que no ha visto durante el entrenamiento y sirve para monitorizar el entrenamiento de la red a título informativo, pero no interviene en ningún cálculo. Suele emplearse cuando se quieren ajustar los parámetros, siendo este conjunto el que indica qué parámetros es mejor utilizar. A más precisión en validación, mejor set de parámetros se tiene. Por este motivo, no se puede confiar en este resultado para dar una idea de la capacidad de generalización de la red, porque se ha elegido la configuración de la red para que dé una precisión más alta. Por tanto, se debe tener un conjunto extra que permita, ahora sí, decir si la red es buena con muestras que no haya visto nunca o no: el de *test*.

3. *Test*: da una intuición de lo buena que es nuestra red al generalizar con un conjunto (más grande que el de validación) nunca visto.

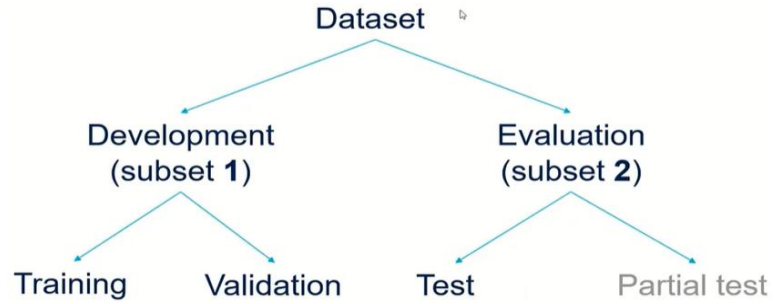


Figura 76 Distribución de un conjunto de datos en validación, entrenamiento y test. [17]

```
x_train, x_test, y_train, y_test = train_test_split(x_features_s,
                                                    y_features_cat,
                                                    test_size=0.25,
                                                    random_state=1)
x_train, x_val, y_train, y_val = train_test_split(x_train,
                                                  y_train,
                                                  test_size=0.25,
                                                  random_state=1)

print('Training samples:', x_train.shape)
print('Validation samples:', x_val.shape)
print('Test samples:', x_test.shape)
```

```
Training samples: (1527, 960)
Validation samples: (509, 960)
Test samples: (679, 960)
```

Figura 77 División entre datos de validación, entrenamiento y test

Se puede comprobar que la suma de las muestras de entrenamiento (1527), más las de validación (509) y las de *test* (679) dan como resultado las 2715 muestras totales que se obtuvieron en el paso anterior.

Salvar las características

Finalmente se guardan los valores de cada tensor en un archivo .CSV para utilizarlos como datos de entrada de prueba y validar la correcta conversión del modelo en Python a C para poder ser usado en el microcontrolador STM32 mediante la librería X-CUBE AI.

```
out_dir = './Output/'
np.savetxt(out_dir + 'x_train.csv', x_train.reshape(len(x_train), 30 * 32), delimiter=",")
np.savetxt(out_dir + 'y_train.csv', y_train, delimiter=",")
np.savetxt(out_dir + 'x_val.csv', x_val.reshape(len(x_val), 30 * 32), delimiter=",")
np.savetxt(out_dir + 'y_val.csv', y_val, delimiter=",")
np.savetxt(out_dir + 'x_test.csv', x_test.reshape(len(x_test), 30 * 32), delimiter=",")
np.savetxt(out_dir + 'y_test.csv', y_test, delimiter=",")
```

Figura 78 Guardado de los tensores

Construcción del modelo

El siguiente paso es construir una red convolucional de tipo secuencial de clasificación, la cual estará formada por las siguientes capas:

- Convolucional.
- MaxPooling.
- Convolucional.
- MaxPooling.
- Flatten.
- Densa con función de activación Relu.
- Densa con función de activación softmax.

Capa convolucional: esta capa actúa como un filtro que faciliten encontrar patrones que permitirán la clasificación de la imagen posteriormente. Este tipo de capas se definen por:

1. El número de filtros/*kernels* para aplicar a la imagen, es decir, el número de matrices por las que se va a convolucionar las imágenes de entrada.
2. El tamaño de esos filtros o *kernels*, que puede ser de 2x2, 3x3, etc.

Al comienzo de la convolución, el filtro se coloca en la parte superior izquierda de la imagen y luego se desplazará una cierta cantidad de cuadros hacia la derecha y cuando llegue al final de la imagen, se desplazará un paso hacia abajo y así sucesivamente hasta que el filtro haya cubierto toda la imagen:

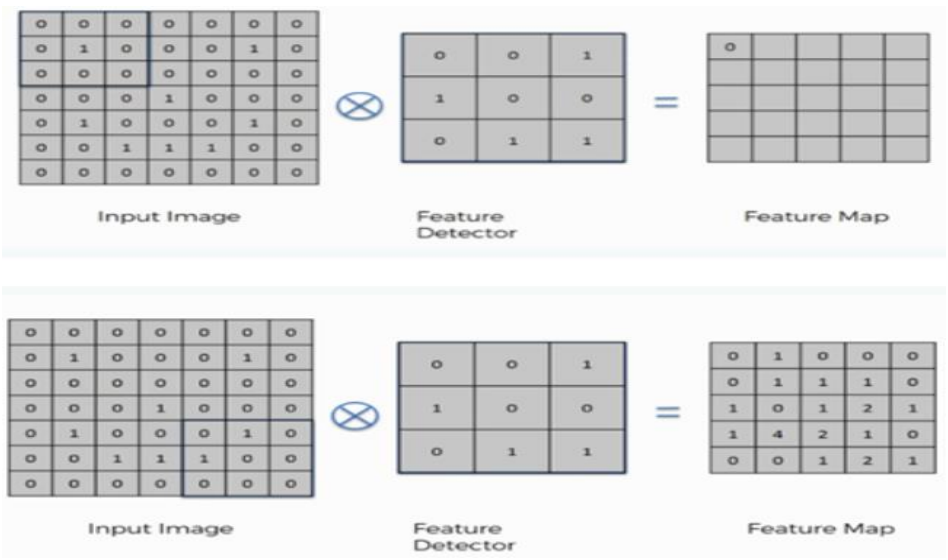


Figura 79 Funcionamiento de la convolución.[19]

La convolución tendrá el efecto de reducir la dimensión del "mapa de características" y facilitará el procesamiento de las imágenes, en este caso los espectrogramas.

Capa Max-Pooling: este tipo de capa se utiliza para ir reduciendo el tamaño de una imagen o matriz. Además, su interés es que reduce el coste de cálculo reduciendo el número de parámetros que tiene que aprender y proporciona una invariancia por pequeñas translaciones (si una pequeña translación no modifica el máximo de la región barrida, el máximo de cada región seguirá siendo el mismo y por tanto la nueva matriz creada será idéntica).

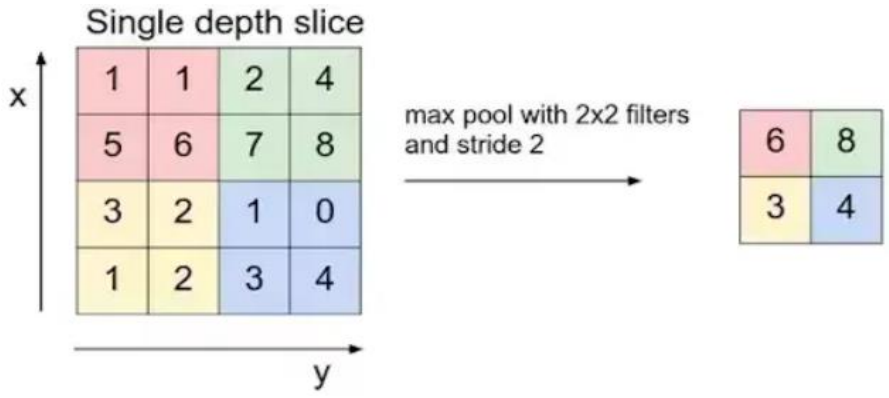


Figura 80 Funcionamiento de la capa max-pooling.[20]

Capa Flatten: transforma los datos de entrada en vector unidimensional sin perder la información extraída en las capas anteriores. Una vez hecho este aplanamiento se puede conectar al resto de capas densas para hacer la predicción final.

Capa Densa con función de activación Relu: Esa capa sustituye todos los valores negativos recibidos en la entrada por ceros. El interés de esas capas de activación es hacer que el modelo sea no lineal y por tanto, más complejo.

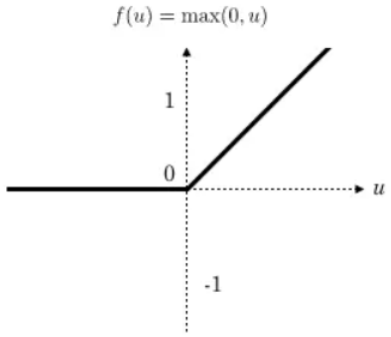


Figura 81 Activación relu.[21]

Capa Densa con función de activación Softmax: este tipo de capa se utiliza para asignar una probabilidad a cada clase. Esto significa que la función SoftMax es útil cuando se necesita saber la probabilidad de que una instancia pertenezca a una clase determinada. Este tipo de capa tiene varias ventajas en la clasificación de imágenes. En primer lugar, la función SoftMax convierte las salidas de la capa anterior en probabilidades que suman uno, lo que facilita la interpretación de los resultados y en segundo lugar, la función SoftMax se puede utilizar para clasificación multiclase.

$$f(z)j = \frac{e^{Z_j}}{\sum_{K=1}^{K} e^{zk}} \qquad (7)$$

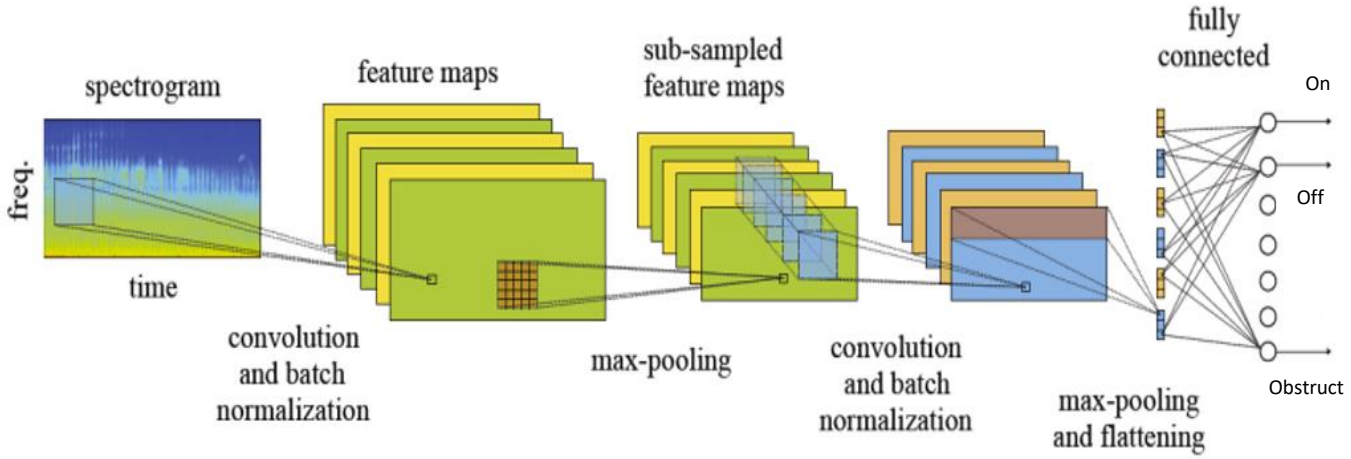


Figura 82 Esquema de funcionamiento de la red neuronal.[17]

```
model = models.Sequential()
model.add(layers.Conv2D(16, (3, 3), activation='relu', input_shape=(30, 32, 1), data_format='channels_last'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(16, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(9, activation='relu'))
model.add(layers.Dense(3, activation='softmax'))

# print model summary
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 30, 16)        160

 max_pooling2d (MaxPooling2D  (None, 14, 15, 16)       0
 )

 conv2d_1 (Conv2D)           (None, 12, 13, 16)        2320

 max_pooling2d_1 (MaxPooling  (None, 6, 6, 16)         0
 2D)

 flatten (Flatten)           (None, 576)               0

 dense (Dense)               (None, 9)                 5193

 dense_1 (Dense)             (None, 3)                 30

=================================================================
Total params: 7,703
Trainable params: 7,703
Non-trainable params: 0
_________________________________________________________________
```

Figura 83 Modelo de capas de la Red Neuronal.

Compilación del modelo

El siguiente paso es compilar el modelo diseñado.

```
sgd = optimizers.SGD(learning_rate=0.01, momentum=0.9, nesterov=True)
model.compile(optimizer=sgd, loss='categorical_crossentropy',  metrics=['acc'])
```

Figura 84 Compilación del modelo.

Para ello se emplea el optimizador “SGD” (*Stochastic Gradient Descent*) con los siguientes parámetros:

- *Learning rate* = 0.01. Es la tasa de aprendizaje que controla el tamaño de los pasos que el optimizador toma al actualizar los parámetros del modelo en cada iteración del entrenamiento. Este parámetro mide la diferencia entre los valores reales y las predicciones del modelo. Una tasa de aprendizaje adecuada es crucial para lograr una convergencia rápida y obtener un modelo óptimo.
- *Momentum*=0.9. El momento ayuda a acelerar el entrenamiento y evita que el optimizador se quede estancado en mínimos locales consiguiendo mejorar la convergencia.
- Nesterov=true. La ventaja del gradiente de Nesterov radica en que puede converger más rápido que el momento estándar y a menudo se considera una mejora sobre la versión clásica del descenso del gradiente. Al anticipar la posición futura de los parámetros, Nesterov tiene la capacidad de ajustar el momento de manera más precisa y lograr una convergencia más rápida.
- La función de pérdida (*loss=categorical entrophy*) realiza una comparación entre las etiquetas reales y la predicción que realiza el modelo para ver su diferencia. Esta función de pérdida provoca que si el modelo

hace mal una predicción lo penaliza de tal forma que hace que ajuste sus parámetros para mejorar dichas predicciones.

Metrica "acc": esta métrica viene de "*accuracy*" (precisión), y mide como su propio nombre indica la precisión con que el modelo clasifica las muestras. Se calcula de la siguiente manera:

$$precisión = \frac{(número\ de\ predicciones\ correctas)}{(número\ total\ de\ muestras)} \quad (8)$$

Es un porcentaje que indica el número de muestras que se han sido clasificadas correctamente. Lo que se pretende durante el entrenamiento es maximizar esta métrica para obtener la mayor precisión posible en las predicciones de las muestras.

Entrenamiento del Modelo

Como paso previo al entrenamiento del modelo, se añade un canal adicional al conjunto de datos para que coincida con la entrada de datos esperada por el modelo, que se definió como (30,32,1) en la capa de convolución inicial.

```
# Reshape features to include channel
x_train_r = x_train.reshape(x_train.shape[0], 30, 32, 1)
x_val_r = x_val.reshape(x_val.shape[0], 30, 32, 1)
x_test_r = x_test.reshape(x_test.shape[0], 30, 32, 1)
```

```
# Train the model for 15 epochs with 550 batch size
history = model.fit(x_train_r, y_train, validation_data=(x_val_r, y_val),
                    batch_size=550, epochs=20, verbose=2)
```

Figura 85 Entrenamiento del modelo.

Durante el entrenamiento, se mostrará información sobre la pérdida y la precisión en cada época tanto para los datos de entrenamiento como para los datos de validación. El resultado del entrenamiento es el siguiente:

Epoch 1/20

3/3 - 1s - loss: 0.8164 - acc: 0.7813 - val_loss: 0.6268 - val_acc: 0.9470 - 829ms/epoch - 276ms/step

Epoch 2/20

3/3 - 0s - loss: 0.5562 - acc: 0.9548 - val_loss: 0.3495 - val_acc: 0.9823 - 325ms/epoch - 108ms/step

Epoch 3/20

3/3 - 0s - loss: 0.2841 - acc: 0.9856 - val_loss: 0.1319 - val_acc: 1.0000 - 320ms/epoch - 107ms/step

Epoch 4/20

3/3 - 0s - loss: 0.1036 - acc: 0.9961 - val_loss: 0.0358 - val_acc: 1.0000 - 315ms/epoch - 105ms/step

Epoch 5/20

3/3 - 0s - loss: 0.0284 - acc: 0.9993 - val_loss: 0.0087 - val_acc: 1.0000 - 328ms/epoch - 109ms/step

Epoch 6/20

3/3 - 0s - loss: 0.0080 - acc: 1.0000 - val_loss: 0.0025 - val_acc: 1.0000 - 321ms/epoch - 107ms/step

Epoch 7/20

3/3 - 0s - loss: 0.0027 - acc: 1.0000 - val_loss: 9.7239e-04 - val_acc: 1.0000 - 310ms/epoch - 103ms/step

```
Epoch 8/20
3/3 - 0s - loss: 0.0012 - acc: 1.0000 - val_loss: 4.8389e-04 - val_acc: 1.0000 - 362ms/epoch - 121ms/step
Epoch 9/20
3/3 - 0s - loss: 6.5339e-04 - acc: 1.0000 - val_loss: 2.9303e-04 - val_acc: 1.0000 - 351ms/epoch - 117ms/step
Epoch 10/20
3/3 - 0s - loss: 3.9707e-04 - acc: 1.0000 - val_loss: 2.0337e-04 - val_acc: 1.0000 - 347ms/epoch - 116ms/step
Epoch 11/20
3/3 - 0s - loss: 2.7896e-04 - acc: 1.0000 - val_loss: 1.5563e-04 - val_acc: 1.0000 - 368ms/epoch - 123ms/step
Epoch 12/20
3/3 - 0s - loss: 2.2056e-04 - acc: 1.0000 - val_loss: 1.2762e-04 - val_acc: 1.0000 - 338ms/epoch - 113ms/step
Epoch 13/20
3/3 - 0s - loss: 1.8106e-04 - acc: 1.0000 - val_loss: 1.0924e-04 - val_acc: 1.0000 - 331ms/epoch - 110ms/step
Epoch 14/20
3/3 - 0s - loss: 1.5556e-04 - acc: 1.0000 - val_loss: 9.6660e-05 - val_acc: 1.0000 - 333ms/epoch - 111ms/step
Epoch 15/20
3/3 - 0s - loss: 1.3787e-04 - acc: 1.0000 - val_loss: 8.7314e-05 - val_acc: 1.0000 - 329ms/epoch - 110ms/step
Epoch 16/20
3/3 - 0s - loss: 1.2639e-04 - acc: 1.0000 - val_loss: 8.0429e-05 - val_acc: 1.0000 - 339ms/epoch - 113ms/step
Epoch 17/20
3/3 - 0s - loss: 1.1791e-04 - acc: 1.0000 - val_loss: 7.5004e-05 - val_acc: 1.0000 - 316ms/epoch - 105ms/step
Epoch 18/20
3/3 - 0s - loss: 1.1181e-04 - acc: 1.0000 - val_loss: 7.0597e-05 - val_acc: 1.0000 - 312ms/epoch - 104ms/step
Epoch 19/20
3/3 - 0s - loss: 1.0679e-04 - acc: 1.0000 - val_loss: 6.7010e-05 - val_acc: 1.0000 - 309ms/epoch - 103ms/step
Epoch 20/20
3/3 - 0s - loss: 1.0313e-04 - acc: 1.0000 - val_loss: 6.4125e-05 - val_acc: 1.0000 - 319ms/epoch - 106ms/step
```

Se puede apreciar como a partir de la época 6 el modelo ya tiene 100% de precisión en los datos de entrenamiento y los de validación. Además, en cada paso la función de pérdida va disminuyendo.

A continuación, se muestra una gráfica con las pérdidas de entrenamiento y validación.

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

plt.figure()
plt.clf()
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.plot(train_loss, color='r', label='training loss')
plt.plot(val_loss, color='g', label='validation loss')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1ca50d2c610>
```

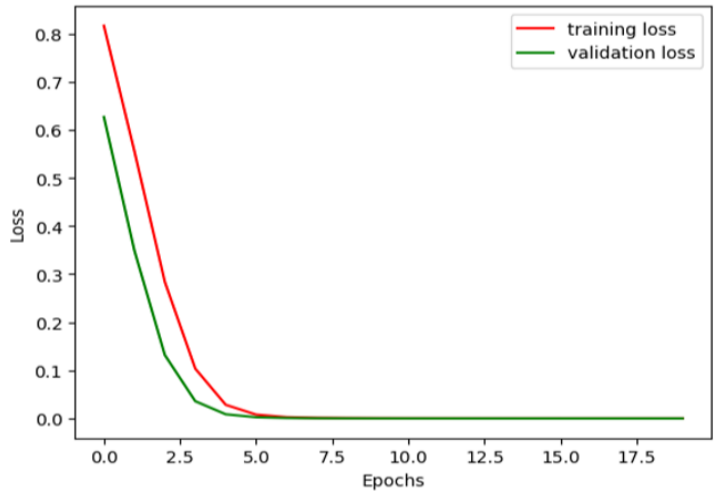


Figura 86 Pérdidas de entrenamiento y validación.

Viendo el resultado que arroja la gráfica mostrada se pude apreciar como los datos de entrenamiento y de validación tienden a converger lo que implica que el modelo de red neuronal está funcionando correctamente y no existe el ni el *overfitting* ni el *underfitting* en la red.

Evaluar la precisión

A continuación, se evalúan los datos de prueba y se obtiene como resultado los valores de pérdida y precisión del modelo.

```
print('Evaluate model:')
results = model.evaluate(x_test_r, y_test)
print(results)
print('Test loss: {:f}'.format(results[0]))
print('Test accuracy: {:.2f}%'.format(results[1] * 100))
```

```
Evaluate model:
22/22 [==============================] - 0s 3ms/step - loss: 8.4883e-05 - acc: 1.0000
[8.488346065860242e-05, 1.0]
Test loss: 0.000085
Test accuracy: 100.00%
```

Figura 87 Evaluación del modelo.

A tenor de los resultados obtenidos se puede comprobar que la precisión es del 100% y la pérdida es prácticamente cero.

Matriz de confusión

La matriz de confusión es una herramienta que permite la visualización del desempeño de un algoritmo que se emplea en aprendizaje supervisado. Cada columna de la matriz representa el número de predicciones de cada clase, mientras que cada fila representa a las instancias en la clase real. Uno de los beneficios de las matrices de confusión es que facilitan ver si el sistema está confundiendo dos clases.

```
y_pred = model.predict(x_test_r)

y_pred_class_nb = np.argmax(y_pred, axis=1)
y_true_class_nb = np.argmax(y_test, axis=1)

accuracy = accuracy_score(y_true_class_nb, y_pred_class_nb)
np.set_printoptions(precision=2)
print("Accuracy = {:.2f}%".format(accuracy * 100))

cm = confusion_matrix(y_true_class_nb, y_pred_class_nb, labels=[0,1,2])

# (optional) normalize to get values in %
# cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

# Loop over data dimensions and create text annotations.
thresh = cm.max() / 2.
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        plt.text(j, i, format(cm[i, j], 'd'),
                 ha="center", va="center",
                 color="white" if cm[i, j] > thresh else "black")

plt.imshow(cm, cmap=plt.cm.Blues)
```

```
22/22 [==============================] - 0s 3ms/step
Accuracy = 100.00%

<matplotlib.image.AxesImage at 0x1ca5281e500>
```

Figura 88 Calculo de la matriz de confusión .

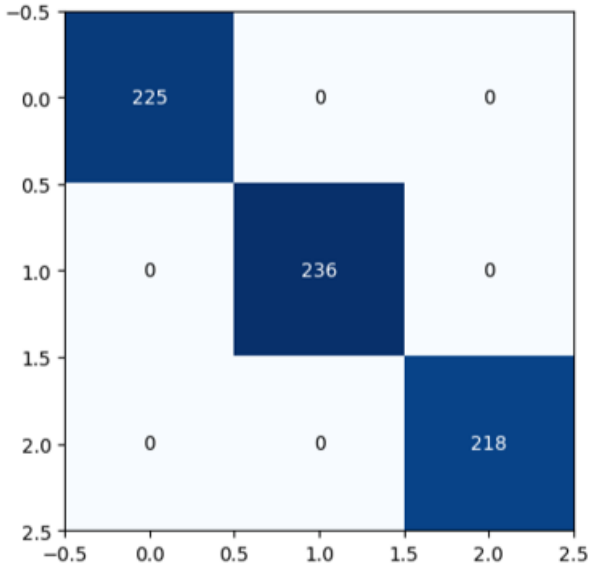


Figura 89 Matriz de confusión.

La primera columna es apagado, la segunda es encendido y la tercera es obstruido. Como se puede apreciar el modelo clasifica muy bien, ya que no hay predicciones en el resto de columnas cuando se está identificando a su clase correspondiente.

Salvar el modelo

Por último, se salva el modelo creado en un archivo .h5 para que la librería X-CUBE-AI proporcionada por ST pueda importar este modelo y generar un modelo en C optimizado equivalente para que lo interprete el microcontrolador del Sensor Tile.

```
# Save the model into an HDF5 file 'model.h5'
model.save(out_dir + 'fan_asc.h5')
```

Figura 90 Guardar el modelo en .h5.

5. VALIDACIÓN Y CONFIGURACIÓN

En este capítulo del Trabajo Fin de Grado se va a explicar cómo se hace la configuración en el programa STM32Cube para poder llevar a cabo la validación del código programado en Python a lenguaje C, proceso que hay que realizar para corroborar que es posible su conversión y su implementación en el microcontrolador seleccionado.

5.1. STM32CUBE

Esta parte del Trabajo Fin de Grado se va a desarrollar prácticamente en su totalidad en el entorno de trabajo proporcionado por ST llamado STM32Cube, este software cubre toda la cartera de aplicaciones de STM32, es por ello que se detalla a continuación sus principales características que tienen relación con el trabajo fin de grado llevado a cabo. El STM32Cube incluye:

- Un conjunto de herramientas de desarrollo de software fáciles de usar para cubrir el desarrollo de proyectos desde la concepción hasta realización, entre las que se encuentran:

– STM32CubeMX, una herramienta de configuración de software gráfico que permite la generación automática de código C de inicialización mediante asistentes gráficos.

– STM32CubeIDE, una herramienta de desarrollo todo en uno con configuración de periféricos, generación de código, código características de compilación y depuración.

– STM32CubeProgrammer (STM32CubeProg), una herramienta de programación disponible en versiones gráficas y de línea de comandos.

– STM32CubeMonitor (STM32CubeMonitor, STM32CubeMonPwr, STM32CubeMonRF, STM32CubeMonUCPD) son poderosas herramientas de monitoreo para ajustar el comportamiento y el rendimiento de aplicaciones STM32 en tiempo real.

- Paquetes de MCU y MPU STM32Cube, plataformas integrales de software específicas para cada serie de microcontroladores y microprocesadores, que incluyen:

– Capa de abstracción de hardware (HAL) de STM32Cube, lo que garantiza una portabilidad del portfolio de STM32.

– API de capa baja STM32Cube, lo que garantiza el mejor rendimiento y con un alto grado de control por parte del usuario sobre el hardware.

– Un conjunto consistente de componentes de middleware como RTOS, USB, TCP/IP, gráficos y sistema de archivos FAT.

– Todas las utilidades de software integradas con conjuntos completos de ejemplos de aplicaciones y periféricos.

- Paquetes de expansión STM32Cube, que contienen componentes de software integrados que complementan las funcionalidades de los paquetes STM32Cube MCU y MPU con:

– Extensiones de middleware y capas de aplicación.

–Ejemplos que se ejecutan en algunas placas de desarrollo específicas de STMicroelectronics.

5.2. LIBRERÍA X-CUBE-AI

La librería X-CUBE-AI amplía STM32CubeMX al proporcionar un generador automático de bibliotecas de redes neuronales, el cual está optimizado en computación y memoria (RAM y memoria Flash), y que convierte redes neuronales preentrenadas de las aplicaciones de *Deep Learning* más utilizadas (como Keras, TensorFlow™ Lite y ONNX) en una biblioteca que se integra automáticamente en el proyecto del usuario final. El proyecto se configura automáticamente, listo para su compilación y ejecución en el microcontrolador STM32.

X-CUBE-AI también amplía STM32CubeMX al agregar, para la creación de proyectos, filtrado de MCU para que se ajusten a los requisitos (como RAM o tamaño de memoria Flash) para la red neuronal seleccionada por el usuario.

La herramienta X-CUBE-AI puede generar tres tipos de proyectos:

- Proyecto de rendimiento del sistema que se ejecuta en la MCU STM32 que permite la medición precisa de la inferencia de la red neuronal en la carga de CPU y uso de memoria.

- Proyecto de validación que valida incrementalmente los resultados devueltos por la red neuronal, estimulada por métodos aleatorios o datos de prueba del usuario, tanto en el PC de escritorio como en el entorno integrado de MCU basado en STM32 Arm® Cortex®-M.

- Proyecto de plantilla de aplicación que permite la creación de una aplicación basada en IA.

En la siguiente figura se puede ver de manera gráfica todo el proceso de trabajo que realiza el CubeMx con la librería X-Cube-AI desde que se le entrega un modelo de red neuronal hasta que se convierte en código C entendible por el compilador GCC.

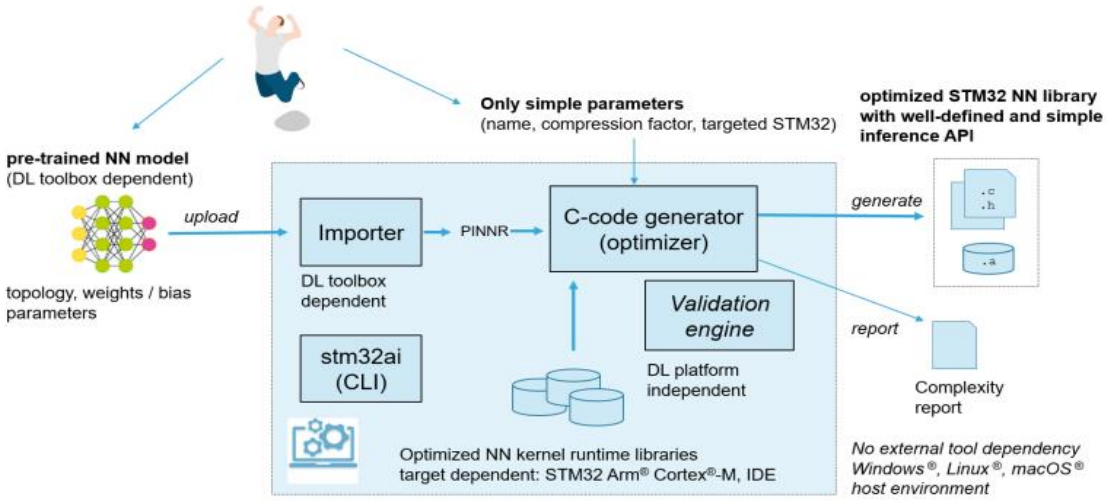


Figura 91 Proceso de trabajo en CubeMx [22]

Una vez explicadas las principales características del software CubeMx y la librería X-Cube-AI, se procede a continuación a detallar el proceso de validación del modelo construido en Python para su conversión en lenguaje C que pueda ser entendido por el compilador GCC del STM32CubeIde.

5.3. PROCESO DE VALIDACIÓN DEL CÓDIGO

Para validar el código hecho en Python hay que seguir estos pasos.

Dentro del STM32CubeIde se va a la pestaña de File→ New Project y se le da un nombre al proyecto. Después se abrirá el menú de board selector. Se selecciona el microcontrolador del Sensor Tile y se pulsa Next.

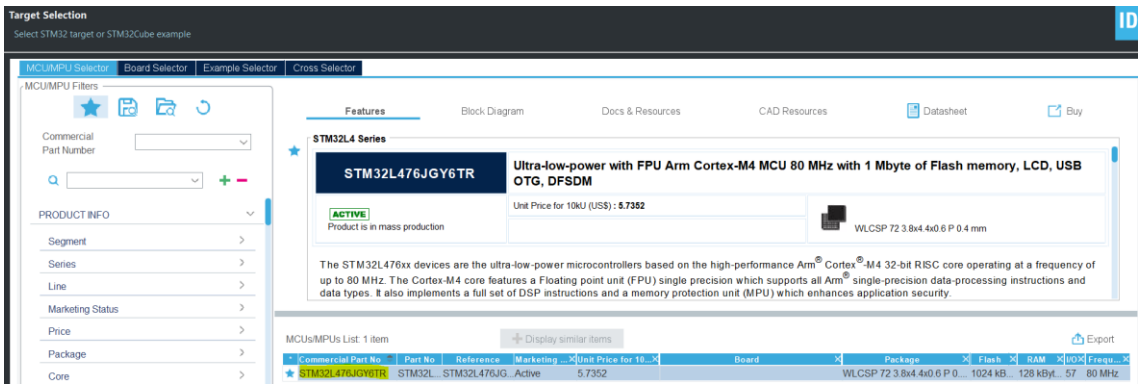


Figura 92 Selección del microcontrolador

El siguiente paso es seleccionar las componentes a utilizar del CubeMX por tanto hay que ir a Software Packs, selección de componentes.

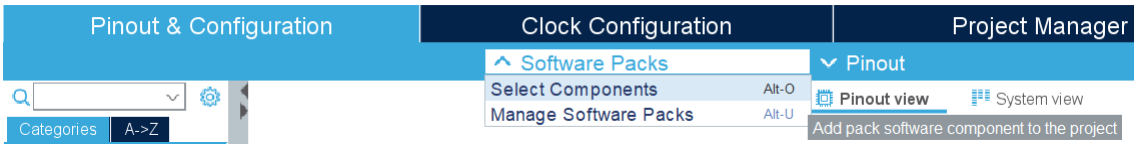


Figura 93 Selección de componentes

Dentro de selección de componentes hay que ir al apartado de STMicroelectronics-X-CUBE-AI, dentro del desplegable Artificial Intelligence X-CUBE-AI, marcar la opción de Core. También hay que seleccionar dentro de Device Application la opción de Validation, ya que lo que se pretende hacer con todos los pasos que se detallan a continuación es la validación del modelo de Keras hecho en Python y su conversión a lenguaje C entendible por el compilador GCC para luego ser cargado en el microcontrolador.

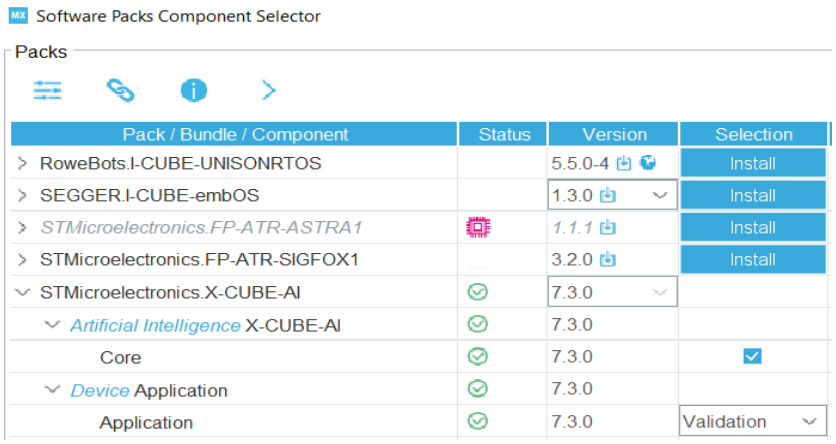


Figura 94 Selección para validación

Se vuelve a la pestaña Pinout & Configuration y aparecerá la imagen con los pines del microcontrolador sombreados, hay que definir la función de éstos.

Figura 95 Pantalla para configuración de pines.

La primera función a habilitar son los pines del depurador, que además servirá para cargar el código en el microcontrolador. Para ello se seleccionan las opciones resaltadas en amarillo, es decir, dentro de System Core se selecciona SYS y luego se escoge Debug como Serial Line. Como resultado se habilitan las señales SWCLK (pin PA14) y SWDIO (pin PA13) en verde en los pines del microcontrolador.

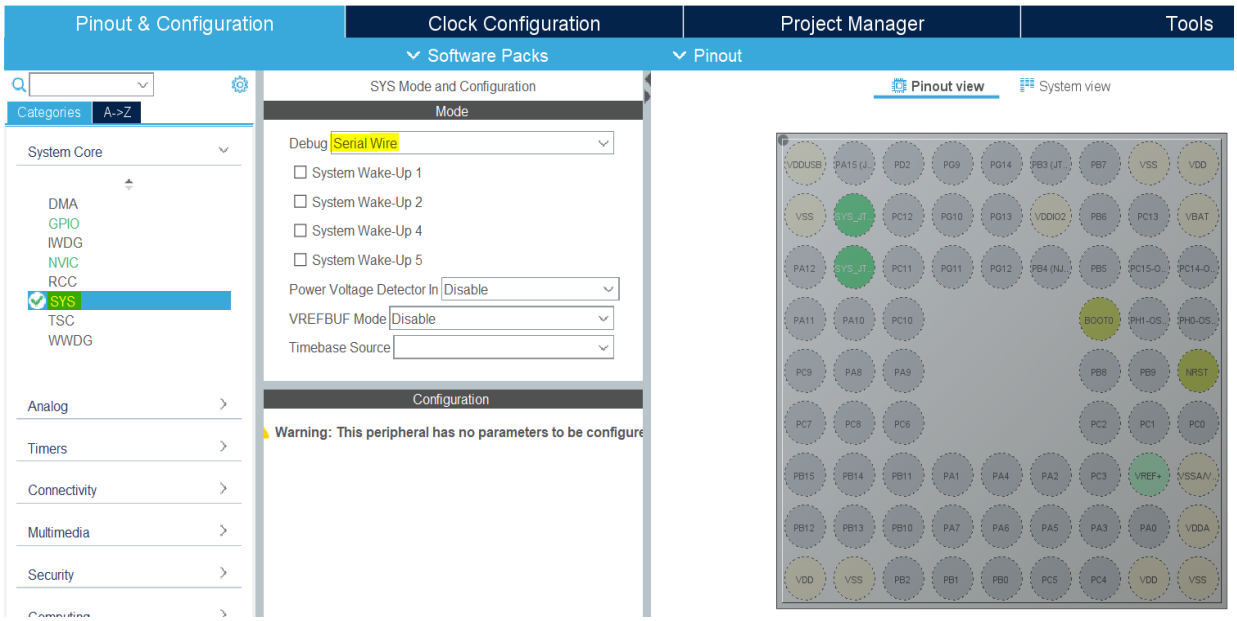


Figura 96 Configuración del depurador.

El Sensor Tile por cómo está rutada la PCB sólo permite la comunicación por USB. Por ello si se quiere utilizar la línea serie se ha de hacer utilizando el puerto USB como virtual COM y ahí enviar la información que se enviaría por línea serie. Por tanto, hay que configurar el USB para que exista comunicación entre el PC y la tarjeta del Sensor Tile.

Se selecciona la pestaña Connectivity y se pincha sobre USB_OTG_FS y se escoge el modo Device Only para que envíe los datos al PC.

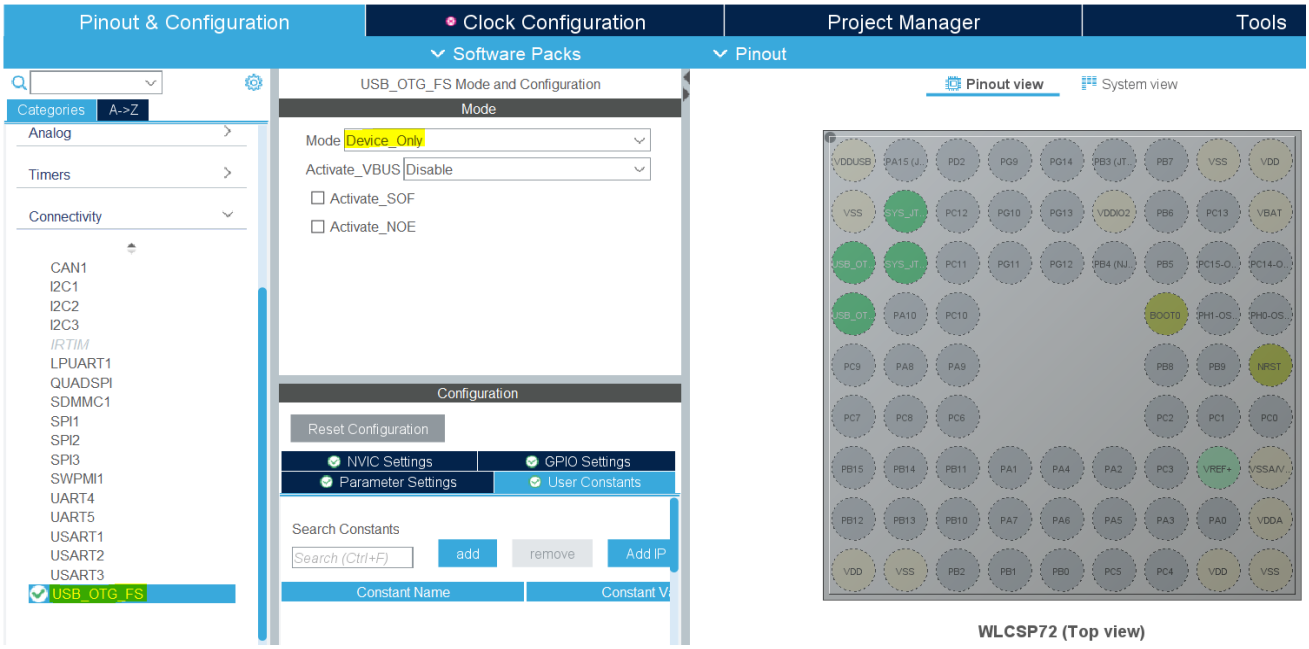


Figura 97 Configuración del USB.

Ahora hay que configurar la parte software del periférico. Se va a Middleware, se selecciona USB_DEVICE y el modo de configuración se elige Communication Device Class (Virtual Port Com). El resto de parámetros no hace falta cambiarlos.

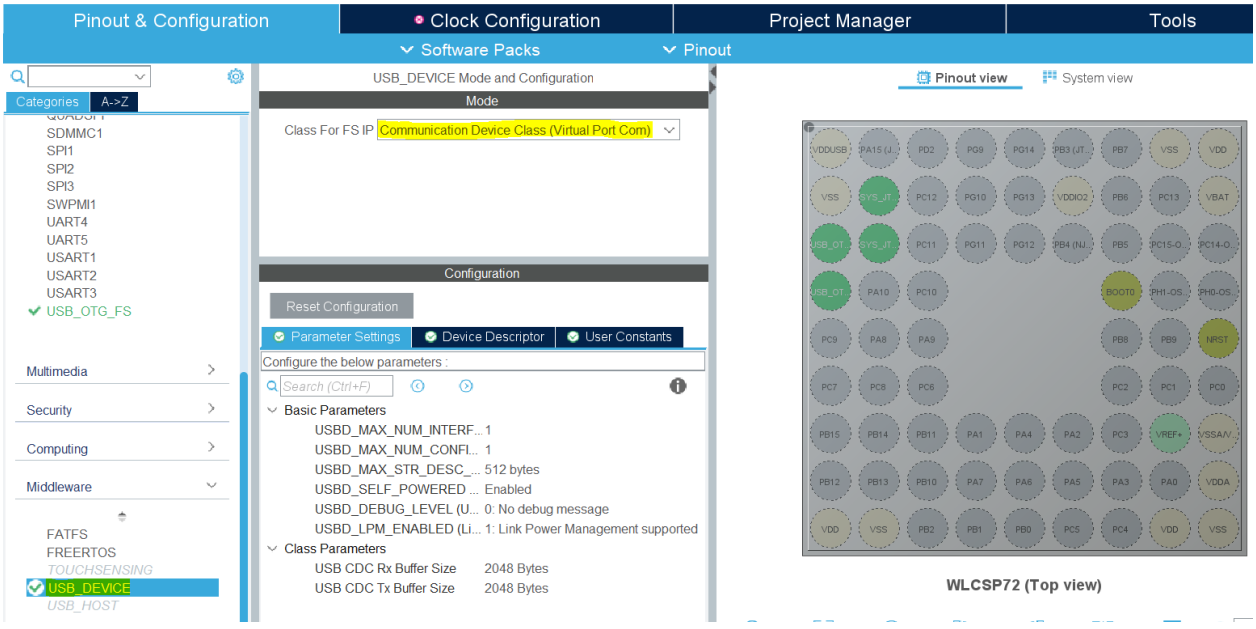


Figura 98 Configuración del USB como puerto COM virtual.

Una vez configurados los pines del microcontrolador hay que configurar el reloj del sistema. Para resolver los problemas que surgen al configurar el USB, se le da al botón de “Resolve clock issues” y automáticamente resuelve los problemas de configuración. Dejando la frecuencia de reloj del sistema en su máxima configuración 80 MHz y el USB a 48 MHz (que es su frecuencia de funcionamiento).

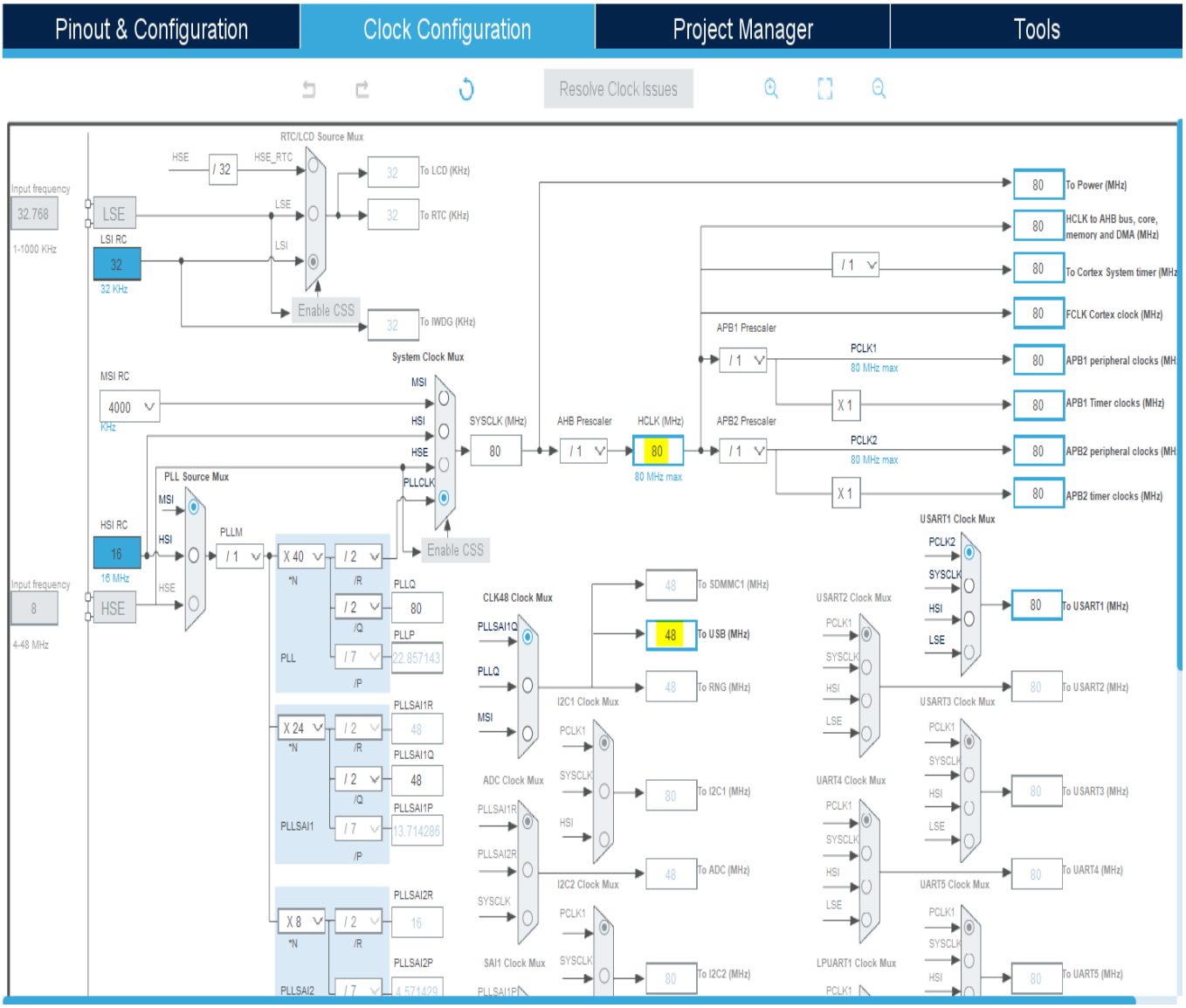


Figura 99 Configuración del reloj del sistema

Ahora lo que se va a hacer es habilitar una UART para realizar la comunicación con el Sensor Tile. Y así poder hacer el "Validate on Target" que es el objetivo de todo este proceso. Para ello, dentro de Connectivity, se selecciona la USART1, se selecciona el modo Asíncrono.

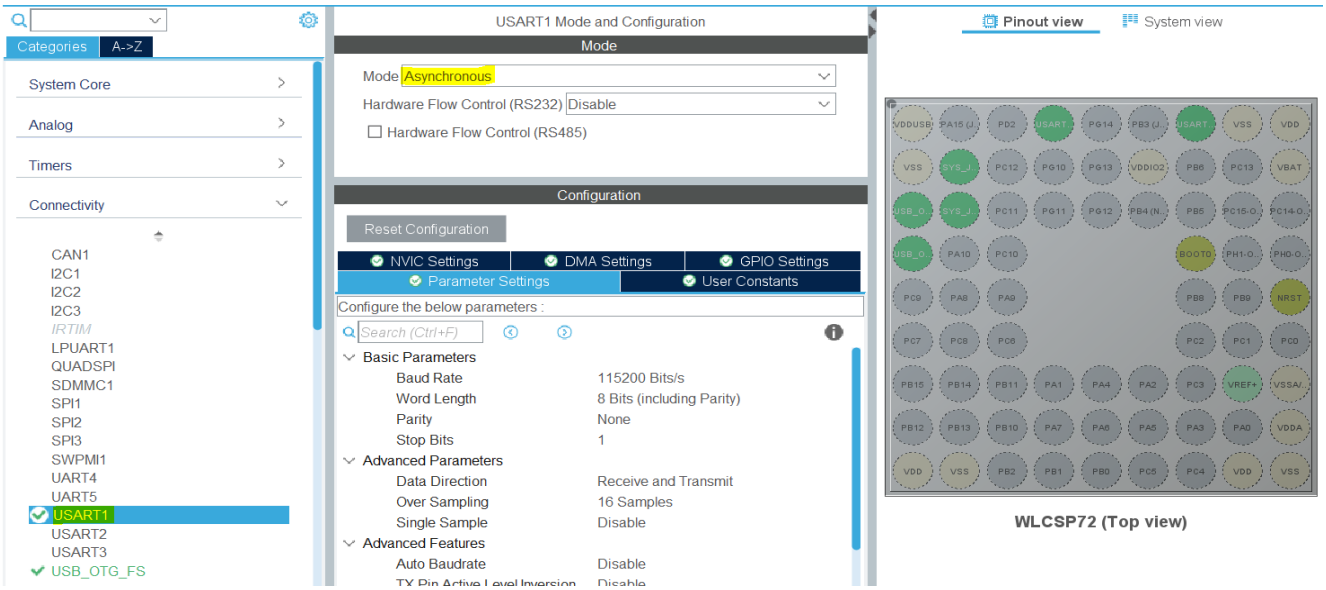


Figura 100 Configuración de la línea serie

Dentro de la pestaña Pinout & Configuration, en el apartado de Software Packs, pulsado el símbolo "+" hay que cargar el archivo .h5 que devuelve el Jupyter Notebook al ejecutar el script hecho en Python donde está diseñada la red neuronal.

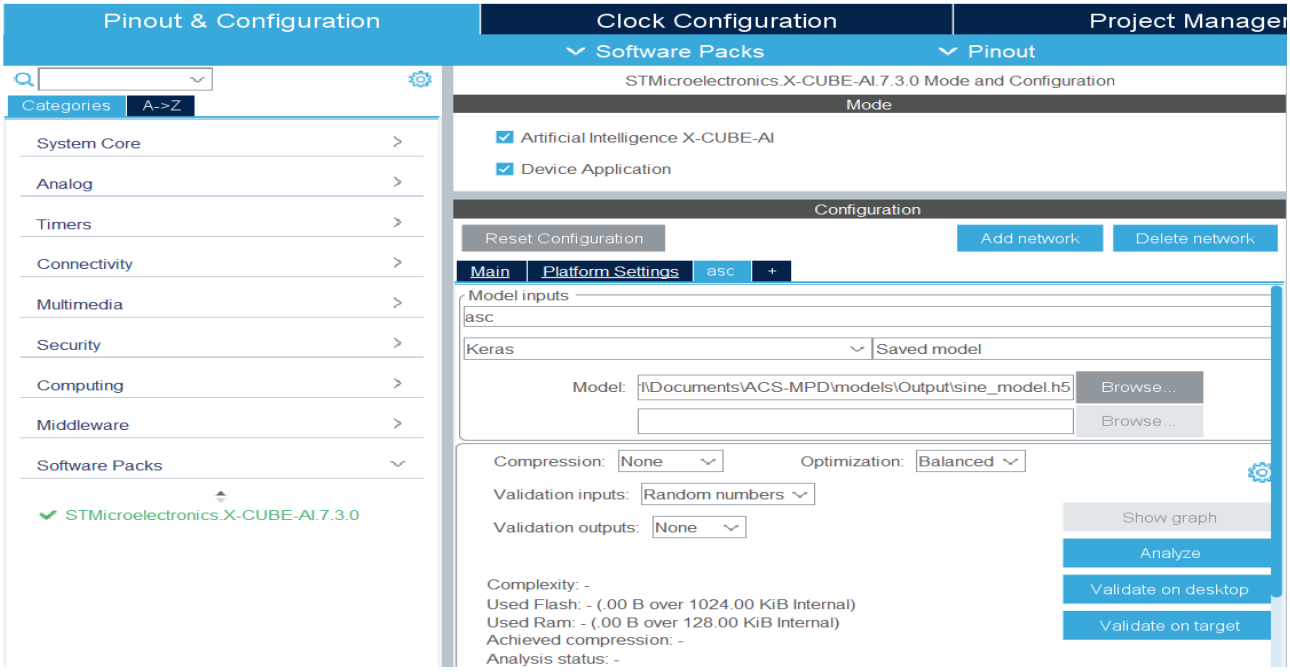


Figura 101 Carga del archivo .h5 devuelto por Jupyter Notebook

Para habilitar la comunicación entre el SensorTile y el PC dentro de Platform Settings hay que seleccionar la USART1 y de manera asíncrona.

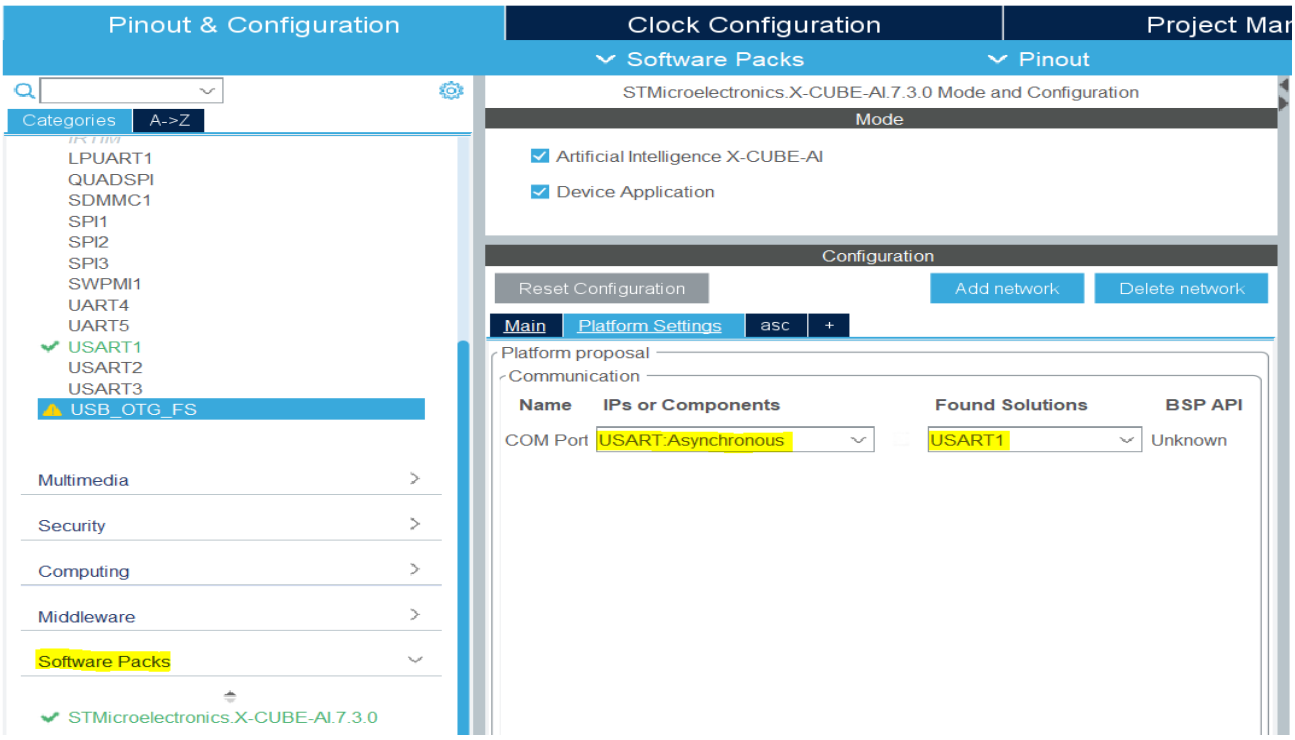


Figura 102 Comunicación por USART del Sensor Tile.

Una vez hecha toda la configuración, se pulsa el botón de generar código (recuadrado en rojo).

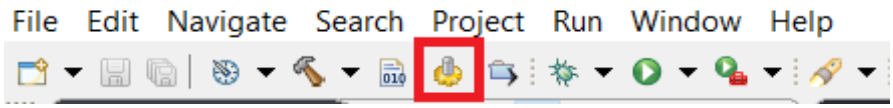


Figura 103 Generación de código.

Sobre el código generado hay que hacer unos cambios para que la comunicación por USB funcione. Dentro de la carpeta X-CUBE-AI → App → En el archivo aiTestutility.c en la parte de arriba dice lo que hay que hacer con el USB:

Description:

*

* History:

* - v1.0 - initial version (from initial aiSystemPerformance file - v5.1)

* - v1.1 - add Arm Compiler 6 support (MDK)

* - v1.2 - add io low level code to manage a COM through the STM32 USB CDC profile

* enabled with the USE_USB_CDC_CLASS = 1 define.

* - v1.3 - Fix compilation issue for H7 dual core

* - v2.0 - add Cortex-m0/m0plus support

* - v2.1 - add U575ZI id

* - v2.2 - clean the way to build/report the STM32 configuration

*

```
*/
NOTE about the USE_USB_CDC_CLASS support
*
*   When USE_USB_CDC_CLASS is set to 1 the code to manage the
*   STM32 USB_DEVICE and the CDC class should added in the project.
*   CDC_Receive_FS()/CDC_Control_FS() functions (file: "usb_cdc_if.c")
*   should be adapted/patched
*
*   o CDC_Receive_FS() to handle the received data
*    ...
*    static int8_t CDC_Receive_FS(uint8_t* Buf, uint32_t *Len) {
*      extern void ioPushInUserUsb(uint8_t *pw, uint32_t *len);
*      ioPushInUserUsb(Buf, Len);
*      USBD_CDC_SetRxBuffer(&hUsbDeviceFS, &Buf[0]);
*      USBD_CDC_ReceivePacket(&hUsbDeviceFS);
*      return (USBD_OK);
*      }
*    ...
*
*   o CDC_Control_FS() to return the valid data for CDC_GET_LINE_CODING
*     request.This is requested by the Python module in charge of the
*     Serial COM with the validation process.
*
*      ...
*      case CDC_GET_LINE_CODING:
*
*        if (length == 7) {
*        *(uint32_t *)pbuf = 115200;
*        pbuf[4] = 0;
*        pbuf[5] = 0;
*        pbuf[6] = 8;
*        }
*
```

```
*     break;
*     …
*
*/
```

Por tanto, lo primero es poner la variable USE_USB_CDC_CLASS a 1 porque por defecto se encuentra a cero dentro del archivo aiTestutiliy.c, en la línea 80 del código.

Lo siguiente en el archivo usb_cdc_if.c se copia lo que se especifica en el archivo aiTestutility.c para poder recibir los paquetes.

```
static int8_t CDC_Receive_FS(uint8_t* Buf, uint32_t *Len)
{
  /* USER CODE BEGIN 6 */
    extern void ioPushInUserUsb(uint8_t *pw, uint32_t *len);
    ioPushInUserUsb(Buf, Len);
    USBD_CDC_SetRxBuffer(&hUsbDeviceFS, &Buf[0]);
    USBD_CDC_ReceivePacket(&hUsbDeviceFS);
    return (USBD_OK);

  /* USER CODE END 6 */
}
```

Y para poder enviarlos hay que añadir esta parte:

```
case CDC_GET_LINE_CODING:
if (length == 7) {
*(uint32_t*) pbuf = 115200;
pbuf[4] = 0;
pbuf[5] = 0;
pbuf[6] = 8;
}
break;
```

Una vez hechos estos cambios, se compila el código y se carga en el SensorTile. Lo siguiente será analizar la viabilidad de la conversión del Script de Python a lenguaje C. Para ello, se vuelve al archivo de configuración .ioc, y dentro de Software Pack, en la pestaña ASC se pulsa el botón Analyze.

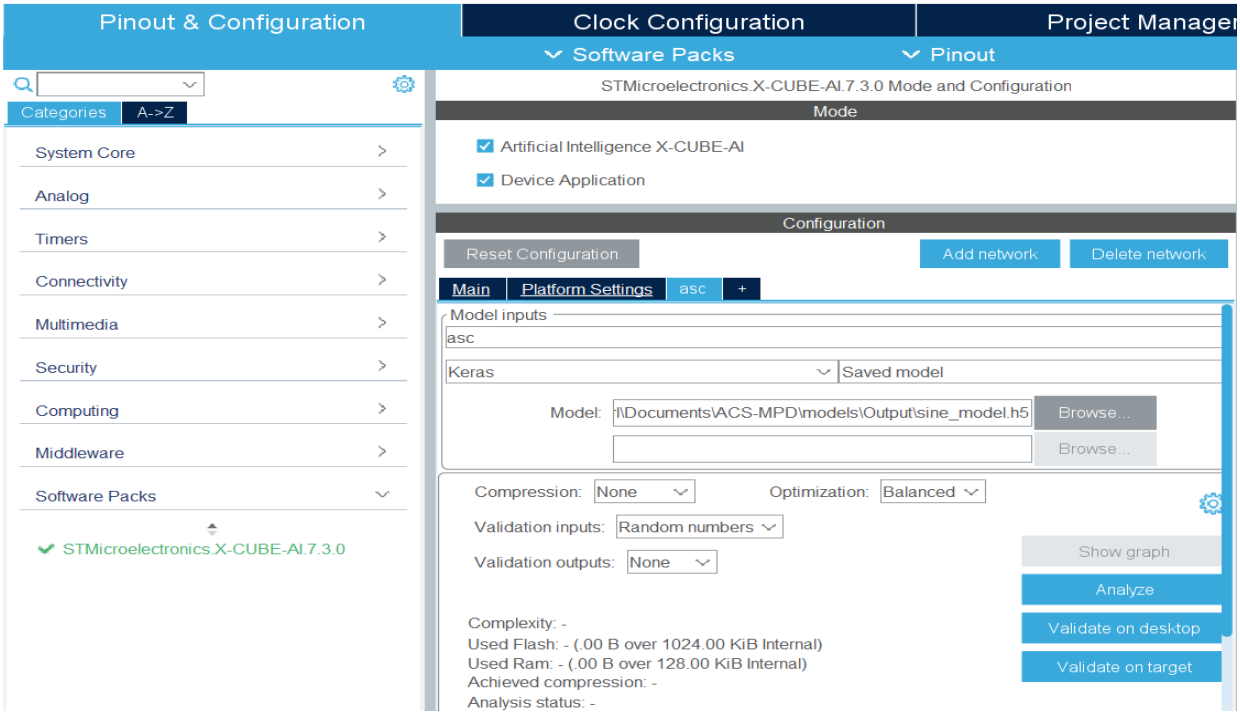


Figura 104 Analyze.

El programa devuelve la siguiente información:

```
Analyzing model
C:/Users/charl/STM32Cube/Repository/Packs/STMicroelectronics/X-CUBE-
AI/7.3.0/Utilities/windows/stm32ai analyze --name asc -m C:/Users/charl/STM32CubeIDE/z/pfinal 4/ACS-
MPD/models/fan_asc/Output/fan_asc.h5 --type keras --compression none --verbosity 1 --
workspace C:\Users\charl\AppData\Local\Temp\mxAI_workspace565288260643300859261127837365201 2 --
output C:\Users\charl\.stm32cubemx\network_output --allocate-inputs --allocate-outputs
Neural Network Tools for STM32AI v1.6.0 (STM.ai v7.3.0-RC5)

 Exec/report summary (analyze)
 ------------------------------------------------------------------------------------------------------------
-
 model file         :   C:\Users\charl\STM32CubeIDE\z\pfinal 4\ACS-MPD\models\fan_asc\Output\fan_asc.h5
 type               :   keras
 c_name             :   asc
 compression        :   none
 options            :   allocate-inputs, allocate-outputs
 optimization       :   balanced
 target/series      :   generic
 workspace dir      :   C:\Users\charl\AppData\Local\Temp\mxAI_workspace565288260643300859261127837365201
2
 output dir         :   C:\Users\charl\.stm32cubemx\network_output
 model_fmt          :   float
 model_name         :   fan_asc
 model_hash         :   f1a7a0dbd44d1d84a34b3604a65c466b
 params #           :   7,703 items (30.09 KiB)
 ------------------------------------------------------------------------------------------------------------
-
 input 1/1          :   'input_0' (domain:activations/**default**)
```

```
                  :  960 items, 3.75 KiB, ai_float, float, (1,30,32,1)
output 1/1        :  'dense_1' (domain:activations/**default**)
                  :  3 items, 12 B, ai_float, float, (1,1,1,3)
macc              :  517,373
weights (ro)      :  30,812 B (30.09 KiB) (1 segment)
activations (rw)  :  19,328 B (18.88 KiB) (1 segment) *
ram (total)       :  19,328 B (18.88 KiB) = 19,328 + 0 + 0
------------------------------------------------------------------------------------------------------------
-
(*) 'input'/'output' buffers can be used from the activations buffer

Model name - fan_asc ['input_0'] ['dense_1']
------------------------------------------------------------------------------------------------------------
id   layer (original)              oshape                param/size     macc      connected to
------------------------------------------------------------------------------------------------------------
0    input_0 (None)                [b:None,h:30,w:32,c:1]
     conv2d_conv2d (Conv2D)        [b:None,h:28,w:30,c:16]   160/640       120,976   input_0
     conv2d (Conv2D)               [b:None,h:28,w:30,c:16]                 13,440    conv2d_conv2d
------------------------------------------------------------------------------------------------------------
1    max_pooling2d (MaxPooling2D)  [b:None,h:14,w:15,c:16]                 13,440    conv2d
------------------------------------------------------------------------------------------------------------
2    conv2d_1_conv2d (Conv2D)      [b:None,h:12,w:13,c:16]   2,320/9,280   359,440   max_pooling2d
     conv2d_1 (Conv2D)             [b:None,h:12,w:13,c:16]                 2,496     conv2d_1_conv2d
------------------------------------------------------------------------------------------------------------
3    max_pooling2d_1 (MaxPooling2D) [b:None,h:6,w:6,c:16]                  2,304     conv2d_1
------------------------------------------------------------------------------------------------------------
4    flatten (Flatten)             [b:None,c:576]                                    max_pooling2d_1
------------------------------------------------------------------------------------------------------------
5    dense_dense (Dense)           [b:None,c:9]              5,193/20,772  5,193     flatten
     dense (Dense)                 [b:None,c:9]                            9         dense_dense
------------------------------------------------------------------------------------------------------------
6    dense_1_dense (Dense)         [b:None,c:3]              30/120        30        dense
     dense_1 (Dense)               [b:None,c:3]                            45        dense_1_dense
------------------------------------------------------------------------------------------------------------
model/c-model: macc=517,373/517,373  weights=30,812/30,812  activations=--/19,328 io=--/0

Number of operations per c-layer
-------------------------------------------------------------------------------
c_id    m_id   name (type)                    #op (type)
-------------------------------------------------------------------------------
0       1      conv2d_conv2d (optimized_conv2d)          147,856 (smul_f32_f32)
1       3      conv2d_1_conv2d (optimized_conv2d)        364,240 (smul_f32_f32)
2       5      dense_dense (dense)                         5,193 (smul_f32_f32)
3       5      dense (nl)                                  9 (op_f32_f32)
4       6      dense_1_dense (dense)                       30 (smul_f32_f32)
5       6      dense_1 (nl)                                45 (op_f32_f32)
-------------------------------------------------------------------------------
total                                          517,373

 Number of operation types
 ---------------------------------------------
 smul_f32_f32          517,319     100.0%
 op_f32_f32                 54       0.0%
```

```
Complexity report (model)
------------------------------------------------------------------------------------
m_id   name            c_macc                    c_rom                     c_id
------------------------------------------------------------------------------------
1      max_pooling2d    |||||||          28.6%   |                 2.1%    [0]
3      max_pooling2d_1  ||||||||||||||||  70.4%   |||||||           30.1%   [1]
5      dense_dense      |                 1.0%    ||||||||||||||||  67.4%   [2, 3]
6      dense_1_dense    |                 0.0%    |                 0.4%    [4, 5]
------------------------------------------------------------------------------------
macc=517,373 weights=30,812 act=19,328 ram_io=0
Creating txt report file C:\Users\charl\.stm32cubemx\network_output\asc_analyze_report.txt
elapsed time (analyze): 0.535s
Getting Flash and Ram size used by the library
Model file:      fan_asc.h5
Total Flash:     49956 B (48.79 KiB)
   Weights:      30812 B (30.09 KiB)
   Library:      19144 B (18.70 KiB)
Total Ram:       22000 B (21.48 KiB)
   Activations: 19328 B (18.88 KiB)
   Library:      2672 B (2.61 KiB)
   Input:        3840 B (3.75 KiB included in Activations)
   Output:       12 B (included in Activations)
Done
Analyze complete on AI model
```

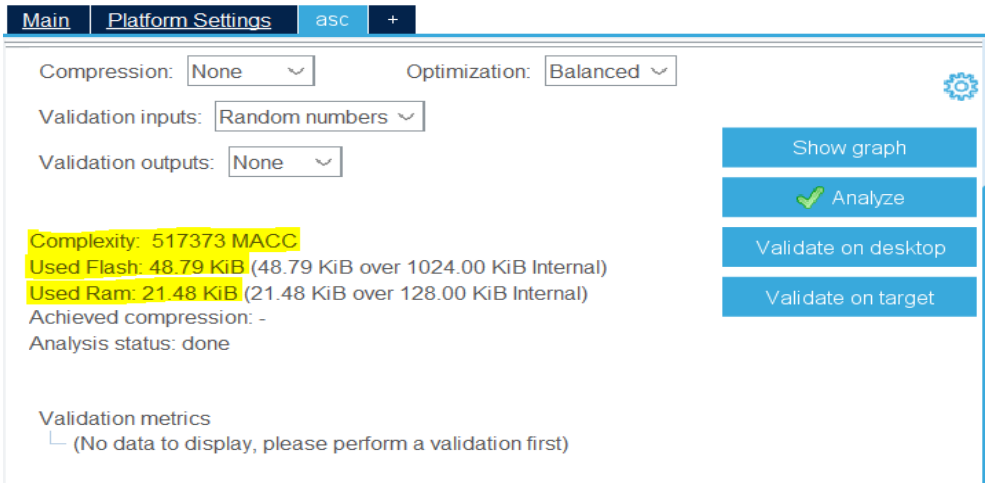


Figura 105 Resultados tras el análisis.

Se puede comprobar a tenor del resultado obtenido, el microcontrolador cubre los requisitos necesarios para ejecutar este script ya que ocupa menos memoria flash y RAM que la disponible. También indica el MACC (*Multiply-and-accumulate complexity*) que es una unidad que indica la complejidad de un modelo *Deep Learning* desde el punto de vista del procesamiento.

El siguiente paso es pulsar el botón de "Validate on desktop" para que el programa devuelva la siguiente información:

```
Starting AI validation on desktop with random data...

C:/Users/charl/STM32Cube/Repository/Packs/STMicroelectronics/X-CUBE-
AI/7.3.0/Utilities/windows/stm32ai validate --name asc -m C:/Users/charl/STM32CubeIDE/z/pfinal 4/ACS-
MPD/models/fan_asc/Output/fan_asc.h5 --type keras --compression none --verbosity 1 --
workspace C:\Users\charl\AppData\Local\Temp\mxAI_workspace565923274569600222132502292242891 7 --
output C:\Users\charl\.stm32cubemx\network_output --allocate-inputs --allocate-outputs
Neural Network Tools for STM32AI v1.6.0 (STM.ai v7.3.0-RC5)
Copying the AI runtime files to the user workspace: C:\Users\charl\AppData\Local\Temp\mxAI_workspace565923
27456960022213250229224289 17\inspector_asc\workspace

 Exec/report summary (validate)
 ------------------------------------------------------------------------------------------------------------
-
 model file         :   C:\Users\charl\STM32CubeIDE\z\pfinal 4\ACS-MPD\models\fan_asc\Output\fan_asc.h5
 type               :   keras
 c_name             :   asc
 compression        :   none
 options            :   allocate-inputs, allocate-outputs
 optimization       :   balanced
 target/series      :   generic
 workspace dir      :   C:\Users\charl\AppData\Local\Temp\mxAI_workspace565923274569600222132502292242891
7
 output dir         :   C:\Users\charl\.stm32cubemx\network_output
 model_fmt          :   float
 model_name         :   fan_asc
 model_hash         :   f1a7a0dbd44d1d84a34b3604a65c466b
 params #           :   7,703 items (30.09 KiB)
 ------------------------------------------------------------------------------------------------------------
-
 input 1/1          :   'input_0' (domain:activations/**default**)
                    :   960 items, 3.75 KiB, ai_float, float, (1,30,32,1)
 output 1/1         :   'dense_1' (domain:activations/**default**)
                    :   3 items, 12 B, ai_float, float, (1,1,1,3)
 macc               :   517,373
 weights (ro)       :   30,812 B (30.09 KiB) (1 segment)
 activations (rw)   :   19,328 B (18.88 KiB) (1 segment) *
 ram (total)        :   19,328 B (18.88 KiB) = 19,328 + 0 + 0
 ------------------------------------------------------------------------------------------------------------
-
 (*) 'input'/'output' buffers can be used from the activations buffer
Setting validation data...
 generating random data, size=10, seed=42, range=(0, 1)
 I[1]: (10, 30, 32, 1)/float32, min/max=[0.000, 1.000], mean/std=[0.494, 0.288], input_0
 No output/reference samples are provided
Running the STM AI c-model (AI RUNNER)...(name=asc, mode=x86)
 X86 shared lib (C:\Users\charl\AppData\Local\Temp\mxAI_workspace565923274569600222132502292242891 7\ins
pector_asc\workspace\lib\libai_asc.dll) ['asc']
 Summary "asc" - ['asc']
 ------------------------------------------------------------------------------
 inputs/outputs      : 1/1
 input_1             : (1,30,32,1), float32, 3840 bytes, in activations buffer
 output_1            : (1,1,1,3), float32, 12 bytes, in activations buffer
```

```
 n_nodes            : 6
 compile_datetime   : Jun 21 2023 21:48:47 (Wed Jun 21 21:48:46 2023)
 activations        : 19328
 weights            : 30812
 macc               : 517373
 ------------------------------------------------------------------------------
 runtime            : STM.AI 7.3.0 (Tools 7.3.0)
 capabilities       : ['IO_ONLY', 'PER_LAYER', 'PER_LAYER_WITH_DATA']
 device             : AMD64 Intel64 Family 6 Model 158 Stepping 10, GenuineIntel (Windows)
 ------------------------------------------------------------------------------
 Results for 10 inference(s) - average per inference
  device            : AMD64 Intel64 Family 6 Model 158 Stepping 10, GenuineIntel (Windows)
  duration          : 0.545ms
  c_nodes           : 6
 c_id  m_id  desc             output                  ms        %
 ------------------------------------------------------------------------------
 0     1     Conv2dPool (0x109)  (1,14,15,16)/float32/13440B      0.210   38.5%
 1     3     Conv2dPool (0x109)  (1,6,6,16)/float32/2304B        0.327   60.0%
 2     5     Dense (0x104)       (1,1,1,9)/float32/36B           0.006    1.1%
 3     5     NL (0x107)          (1,1,1,9)/float32/36B           0.001    0.1%
 4     6     Dense (0x104)       (1,1,1,3)/float32/12B           0.001    0.1%
 5     6     NL (0x107)          (1,1,1,3)/float32/12B           0.001    0.2%
 ------------------------------------------------------------------------------
                                                  0.545 ms
 NOTE: duration and exec time per layer is just an indication. They are dependent of the HOST-machine work-
load.
Running the Keras model...
Saving validation data...
 output directory: C:\Users\charl\.stm32cubemx\network_output
 creating C:\Users\charl\.stm32cubemx\network_output\asc_val_io.npz
 m_outputs_1: (10, 1, 1, 3)/float32, min/max=[0.000, 0.993], mean/std=[0.333, 0.466], dense_1
 c_outputs_1: (10, 1, 1, 3)/float32, min/max=[0.000, 0.993], mean/std=[0.333, 0.466], dense_1
Computing the metrics...
 Cross accuracy report #1 (reference vs C-model)
 ----------------------------------------------------------------------------------------------------
 notes: - the output of the reference model is used as ground truth/reference value
        - 10 samples (3 items per sample)
  acc=100.00%, rmse=0.000000023, mae=0.000000008, l2r=0.000000040, nse=100.00%
  3 classes (10 samples)
  ----------------------
  C0       0    .    .
  C1       .    0    .
  C2       .    .    10

 Evaluation report (summary)
 ----------------------------------------------------------------------------------------------------
----------------------------------------
 Output      acc      rmse       mae        l2r        mean       std        nse        tensor
 ----------------------------------------------------------------------------------------------------
----------------------------------------
 X-cross #1   100.00%   0.000000023   0.000000008   0.000000040   -
0.000000003   0.000000023   1.000000000   dense_1, ai_float, (1,1,1,3), m_id=[6]
 ----------------------------------------------------------------------------------------------------
```

```
-------------------------------------------

  rmse : Root Mean Squared Error
  mae  : Mean Absolute Error
  l2r  : L2 relative error
  nse  : Nash-Sutcliffe efficiency criteria
Creating txt report file C:\Users\charl\.stm32cubemx\network_output\asc_validate_report.txt
elapsed time (validate): 2.333s
Validation ended
```

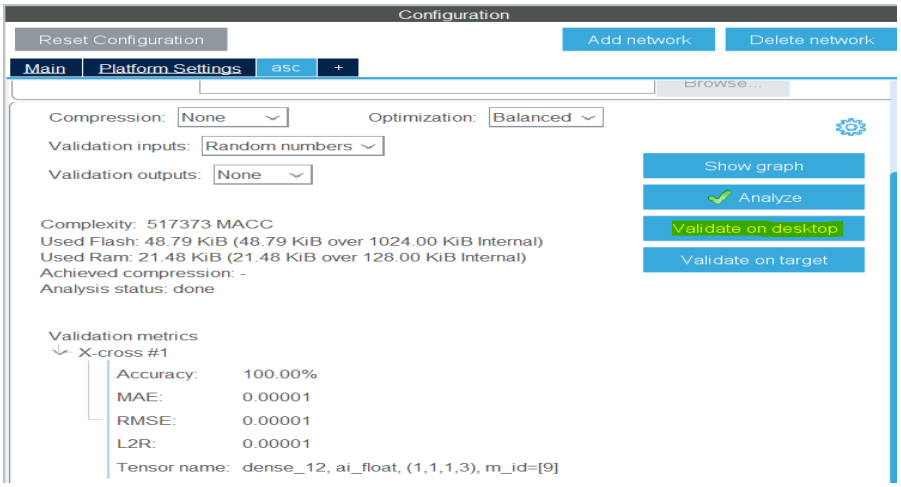


Figura 106 Validate on desktop.

Del informe devuelto es importante fijarse en el error "l2r", este parámetro indica el error entre el modelo original y el migrado a código C entendible por el microcontrolador (error relativo). Si este parámetro es menor a 0.01 se considera que la migración es correcta. En este caso su valor es de **0.000000040** , por tanto la migración es correcta. Para este paso hay que tener en cuenta que se puede hacer con un generador aleatorio o con un *dataset* predeterminado. En este Trabajo Fin de Grado se ha hecho con un generador aleatorio.

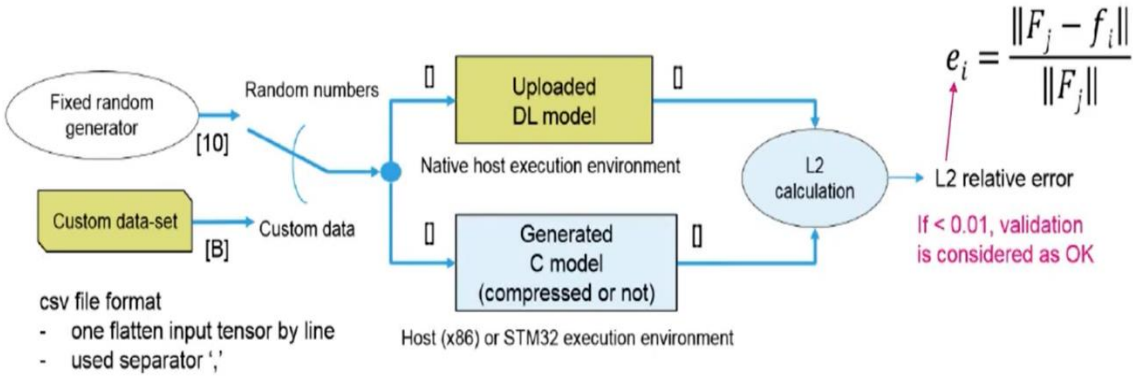


Figura 107 Diagrama de cálculo del error L2R.[23]

Por último se carga el código generado que se encuentra en el STM32CubeIde en el SensorTile, se vuelve al archivo .ioc y se pulsa sobre "Validate on Target". En la pantalla siguiente se selecciona el puerto de comunicación que se corresponde con el Sensor Tile, en este caso con el COM4 y se pulsa Ok.

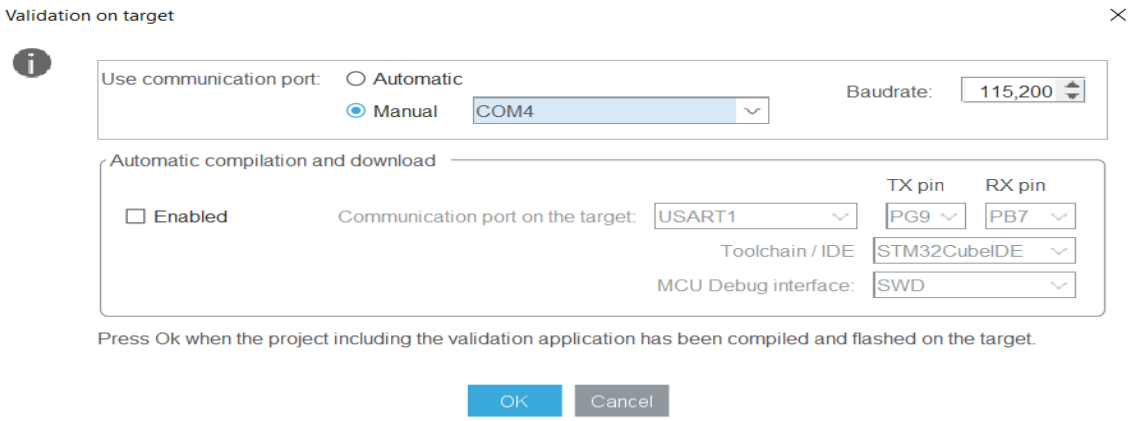


Figura 108 Puerto de comunicación del Sensor Tile.

El software devuelve la siguiente información:

```
Starting AI validation on target with random data...
C:/Users/charl/STM32Cube/Repository/Packs/STMicroelectronics/X-CUBE-
AI/7.3.0/Utilities/windows/stm32ai validate --name asc -m C:/Users/charl/STM32CubeIDE/z/pfinal 4/ACS-
MPD/models/fan_asc/Output/fan_asc.h5 --type keras --compression none --verbosity 1 --
workspace C:\Users\charl\AppData\Local\Temp\mxAI_workspace566836295452400106848091056326052 91 --
output C:\Users\charl\.stm32cubemx\network_output --allocate-inputs --allocate-outputs --mode stm32 --
desc COM4:115200
Neural Network Tools for STM32AI v1.6.0 (STM.ai v7.3.0-RC5)

 Exec/report summary (validate)
 ------------------------------------------------------------------------------------------------------------
--
 model file         :   C:\Users\charl\STM32CubeIDE\z\pfinal 4\ACS-MPD\models\fan_asc\Output\fan_asc.h5
 type               :   keras
 c_name             :   asc
 compression        :   none
 options            :   allocate-inputs, allocate-outputs
 optimization       :   balanced
 target/series      :   generic
 workspace dir      :   C:\Users\charl\AppData\Local\Temp\mxAI_workspace566836295452400106848091056326052
91
 output dir         :   C:\Users\charl\.stm32cubemx\network_output
 model_fmt          :   float
 model_name         :   fan_asc
 model_hash         :   f1a7a0dbd44d1d84a34b3604a65c466b
 params #           :   7,703 items (30.09 KiB)
 ------------------------------------------------------------------------------------------------------------
--
 input 1/1          :   'input_0' (domain:activations/**default**)
                    :   960 items, 3.75 KiB, ai_float, float, (1,30,32,1)
 output 1/1         :   'dense_1' (domain:activations/**default**)
                    :   3 items, 12 B, ai_float, float, (1,1,1,3)
 macc               :   517,373
 weights (ro)       :   30,812 B (30.09 KiB) (1 segment)
 activations (rw)   :   19,328 B (18.88 KiB) (1 segment) *
 ram (total)        :   19,328 B (18.88 KiB) = 19,328 + 0 + 0
```

```
 ------------------------------------------------------------------------------------------------------
--
 (*) 'input'/'output' buffers can be used from the activations buffer
Setting validation data...
 generating random data, size=10, seed=42, range=(0, 1)
 I[1]: (10, 30, 32, 1)/float32, min/max=[0.000, 1.000], mean/std=[0.494, 0.288], input_0
 No output/reference samples are provided
Running the STM AI c-model (AI RUNNER)...(name=asc, mode=stm32)
 STM Proto-buffer protocol 2.3 (SERIAL:COM4:115200:connected) ['asc']
 Summary "asc" - ['asc']
 ------------------------------------------------------------------------------
 inputs/outputs      : 1/1
 input_1             : (1,30,32,1), float32, 3840 bytes, in activations buffer
 ouputs_1            : (1,1,1,3), float32, 12 bytes, in activations buffer
 n_nodes             : 6
 compile_datetime    : Jun 21 2023 21:24:49 (Wed Jun 21 21:09:13 2023)
 activations         : 19328
 weights             : 30812
 macc                : 517373
 ------------------------------------------------------------------------------
 runtime             : Protocol 2.3 - STM.AI (/gcc) 7.3.0 (Tools 7.3.0)
 capabilities        : ['IO_ONLY', 'PER_LAYER', 'PER_LAYER_WITH_DATA']
 device              : 0x415 - STM32L4x6xx @80/80MHz fpu,art_lat=4,art_icache,art_dcache
 ------------------------------------------------------------------------------
STM.IO:   0%|          | 0/10 [00:00<?, ?it/s]
STM.IO:  90%|######### | 9/10 [00:00<00:00, 67.42it/s]
 Results for 10 inference(s) - average per inference
  device            : 0x415 - STM32L4x6xx @80/80MHz fpu,art_lat=4,art_icache,art_dcache
  duration          : 90.704ms
  CPU cycles        : 7256323
  cycles/MACC       : 14.03
  c_nodes           : 6
 c_id  m_id  desc             output                  ms        %
 ------------------------------------------------------------------------------
 0     1     Conv2dPool (0x109)  (1,14,15,16)/float32/13440B     43.843   48.3%
 1     3     Conv2dPool (0x109)  (1,6,6,16)/float32/2304B        46.241   51.0%
 2     5     Dense (0x104)       (1,1,1,9)/float32/36B           0.585    0.6%
 3     5     NL (0x107)          (1,1,1,9)/float32/36B           0.006    0.0%
 4     6     Dense (0x104)       (1,1,1,3)/float32/12B           0.014    0.0%
 5     6     NL (0x107)          (1,1,1,3)/float32/12B           0.016    0.0%
 ------------------------------------------------------------------------------
                                           90.704 ms
Running the Keras model...
Saving validation data...
 output directory: C:\Users\charl\.stm32cubemx\network_output
 creating C:\Users\charl\.stm32cubemx\network_output\asc_val_io.npz
 m_outputs_1: (10, 1, 1, 3)/float32, min/max=[0.000, 0.993], mean/std=[0.333, 0.466], dense_1
 c_outputs_1: (10, 1, 1, 3)/float32, min/max=[0.000, 0.993], mean/std=[0.333, 0.466], dense_1
Computing the metrics...
 Cross accuracy report #1 (reference vs C-model)
 ------------------------------------------------------------------------------------------------------
 notes: - the output of the reference model is used as ground truth/reference value
        - 10 samples (3 items per sample)
```

```
 acc=100.00%, rmse=0.000000005, mae=0.000000002, l2r=0.000000009, nse=100.00%
 3 classes (10 samples)
 ----------------------
 C0       0    .    .
 C1       .    0    .
 C2       .    .    10

 Evaluation report (summary)
 ------------------------------------------------------------------------------------------------------------------
----------------------------------------
 Output      acc      rmse       mae        l2r        mean       std        nse        tensor
 ------------------------------------------------------------------------------------------------------------------
----------------------------------------
 X-cross #1  100.00%  0.000000005  0.000000002  0.000000009  -
0.000000001  0.000000005  1.000000000  dense_1, ai_float, (1,1,1,3), m_id=[6]
 ------------------------------------------------------------------------------------------------------------------
----------------------------------------

  rmse : Root Mean Squared Error
  mae  : Mean Absolute Error
  l2r  : L2 relative error
  nse  : Nash-Sutcliffe efficiency criteria
Creating txt report file C:\Users\charl\.stm32cubemx\network_output\asc_validate_report.txt
elapsed time (validate): 2.583s
Validation ended
```

Del informe devuelto por el programa se ve claramente que la conversión del modelo hecho en Python es correcta ya que el error "l2r" es de **0.000000009** , valor inferior a 0.01. También hay que destacar que hay una precisión del 100%.

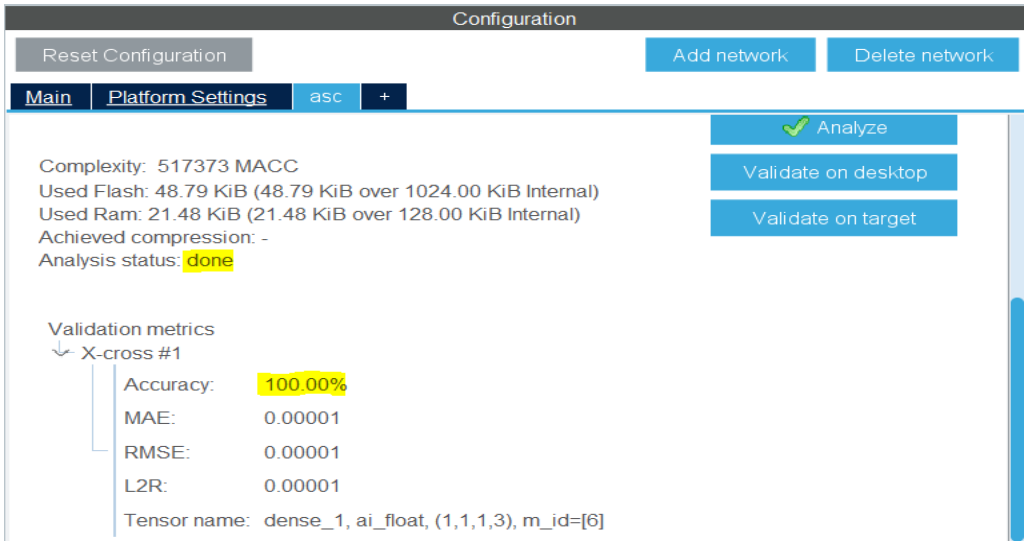


Figura 109 Resultado de validate on target.

5.4.CONFIGURACIÓN DEL MICROCONTROLADOR Y PERIFÉRICOS

Antes de hacer el código final hay que configurar el microcontrolador y los periféricos que se van a utilizar. Lo primero a realizar es crear un nuevo proyecto y en el CubeMX cambiar la configuración ya que no se va a validar el código del script de Python porque ya se ha validado. Por tanto, dentro de la pestaña Software Pack se selecciona la opción “Not Selected”.

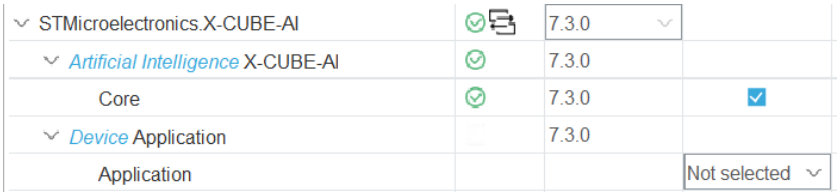


Figura 110 Configuración para elaborar el código final.

Se configuran los pines del DFSDM, para ello primero se marca dentro de Computing el DFSDM1 y luego se selecciona el Chanel 5 porque en el diseño esquemático el micrófono está cableado a esos pines. Se marca también el CKOUT para que genere el reloj para el micrófono.

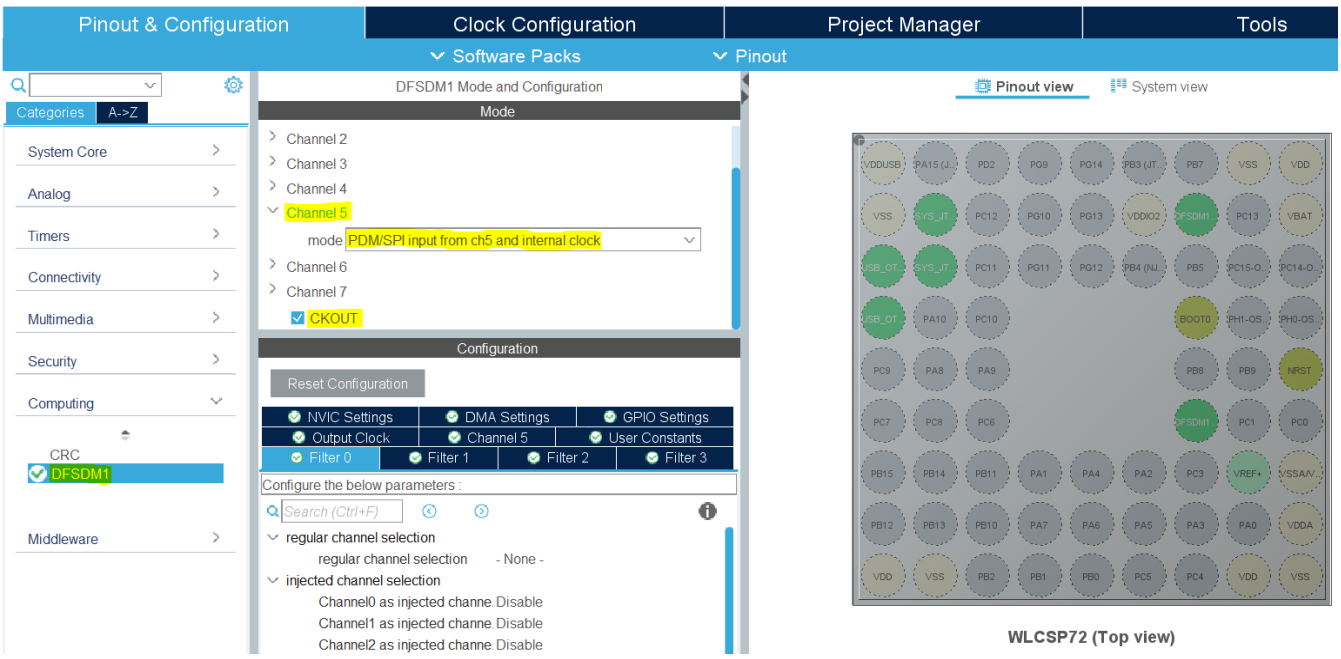


Figura 111 Configuración del canal 5 del DFSDM.

El DFSDM dispone de 4 filtros, se va a emplear el filtro 0 y se configura de la siguiente manera. Se utiliza una conversión regular, denominada también FAST, por tanto, se habilita ésta. Después se va a la pestaña DMA Settings y se habilita la transferencia desde el periférico hacia la memoria y se estable el *buffer* circular.

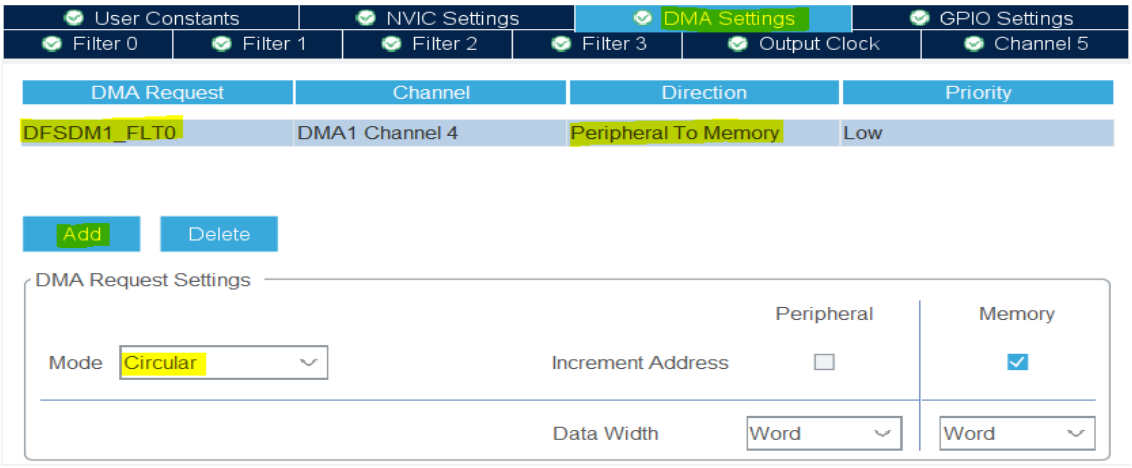


Figura 112 Configuración del DMA y buffer circular.

Se vuelve a la pestaña Filter 0, se habilita el modo DMA.

El DFSDM ha de estar configurado igual que en el *firmware* de grabación DataLog proporcionado por ST, esto es así porque los audios con los que se ha entrenado la red neuronal en el *script* de Python han sido grabados a través de este *firmware*. Por tanto, la red neuronal cuando esté cargada en el microcontrolador para que funcione de la mejor manera posible ha de estar configurado igual que lo estaba cuando fueron grabados los audios de entrenamiento.

La frecuencia de muestreo deseada es de 16KHz, que es la frecuencia de trabajo del script de Python. Para obtener una frecuencia de muestreo de 16 KHz, hay que utilizar esta fórmula:

regular conversion with FAST = 1 (except first conversion):

for Sincx and FastSinc filters:

$$t = \frac{CNVCNT}{f_{DFSDMCLK}} = \frac{F_{OSR}\ I_{OSR}}{f_{CKIN}} \qquad (9)$$

Donde:

- f_{CKIN} es la frecuencia de reloj del canal de entrada.
- F_{OSR} es la relación de sobre muestreo del filtro.
- I_{OSR} es la relación de sobre muestreo del integrador.

Tomando los datos de configuración del *firmware* DataLog el valor de la relación de sobre muestreo del integrador I_{OSR} es de 1, la relación de sobre muestreo del filtro F_{OSR} es 128 y el preescalador del reloj DIV es 24 y se utiliza el reloj de audio del DFSDM, que como su propio nombre indica es especial para aplicaciones de audio.

La frecuencia más cercana a la calculada que se puede conseguir con los preescaladores en la configuración del reloj es de 49.142857 MHz.

Se calcula la frecuencia de trabajo del periférico DFSDM mediante la fórmula anterior:

$$\text{Frecuencia de muestreo} = \frac{F_{DFSDMCLK/DIV}}{F_{OSR}\ I_{OSR}};\ 16\ \text{KHz} = \frac{F_{DFSDMCLK/24}}{128*1} = 49.152\ \text{MHz}$$

Por último, se selecciona también el filtro que nos dará una resolución de 16 bit al igual que estaba configurado el firmware DataLog que es el Sinc4.

Se muestran a continuación las configuraciones en el CubeMX.

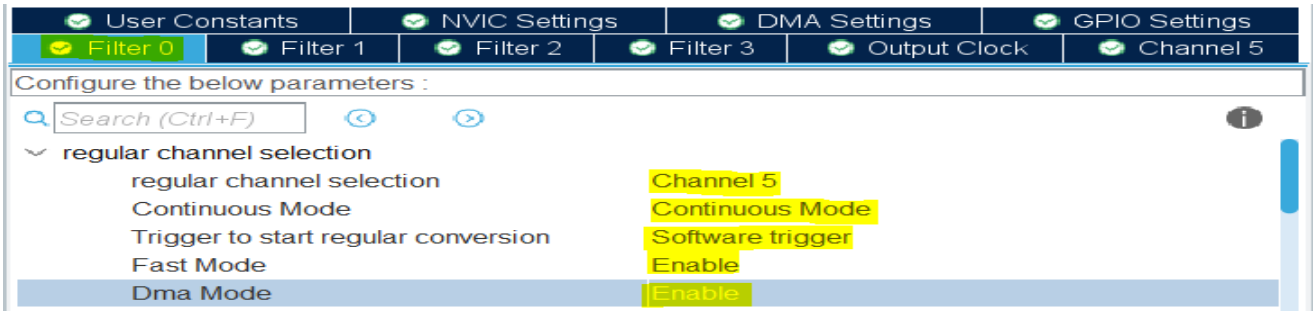


Figura 113 Configuración del Filter 0.

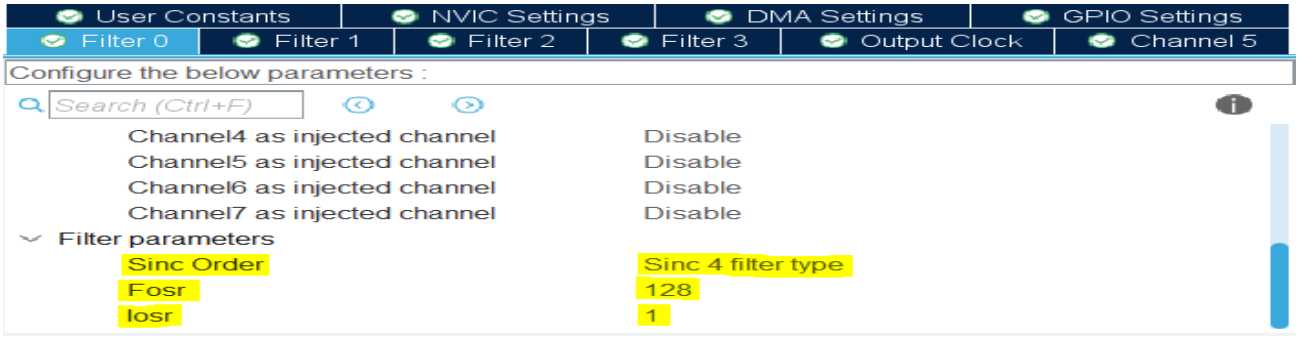


Figura 114 Configuración del Filter 0 bis.

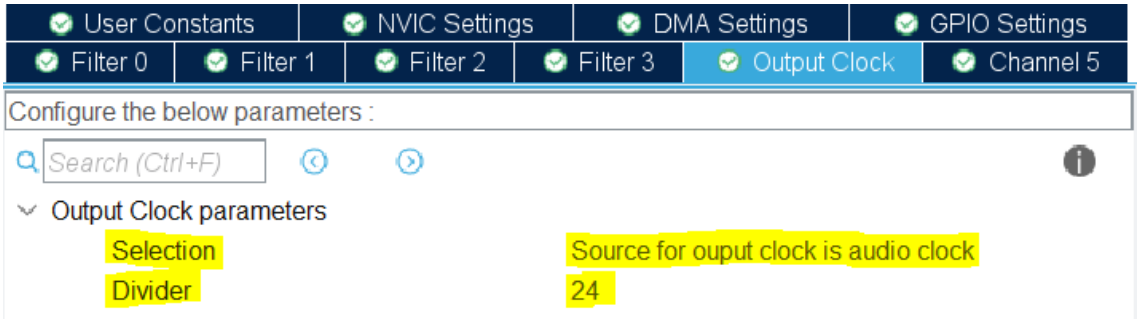


Figura 115 Configuración de la salida de reloj.

El último paso es configura en el canal utilizado, en este caso el canal 5, un desplazamiento de 8 bits porque la máxima resolución que se obtiene es de 24 bit y se desea obtener una resolución de 16 bits.

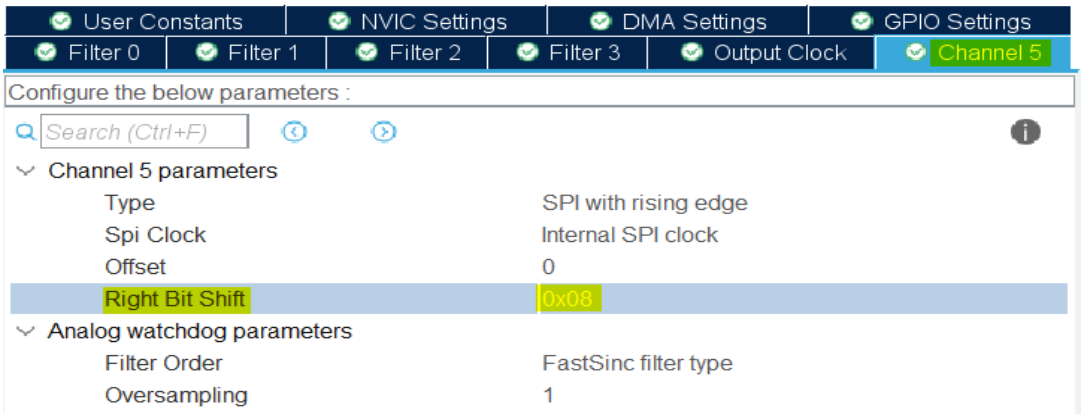


Figura 116 Configuración del desplazamiento del canal 5.

Completando el resto de configuración del microcontrolador, también se habilita un GPIO para que el led naranja que está rutado en ese punto proporcione un *feedback* del funcionamiento de la red.

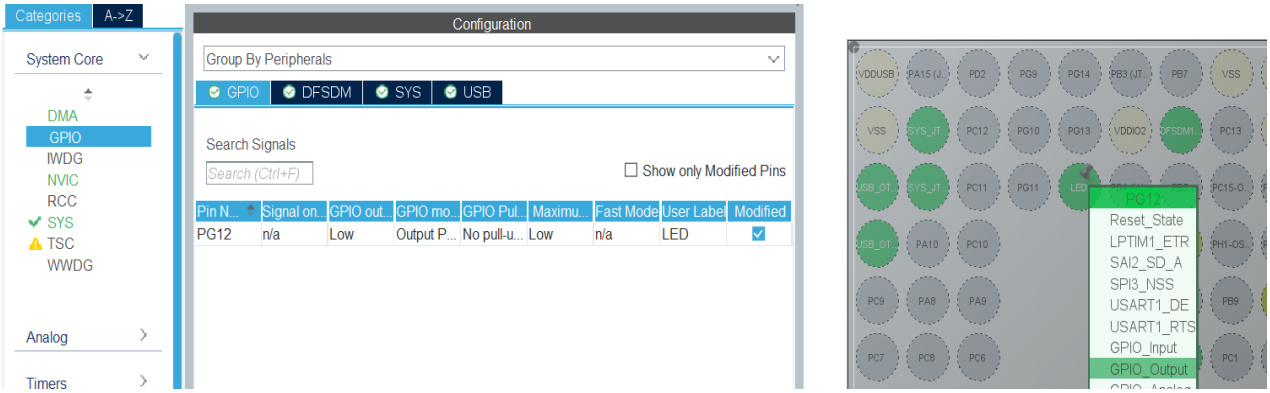


Figura 117 Habilitación del led.

El reloj de todo el sistema queda configurado de la siguiente manera:

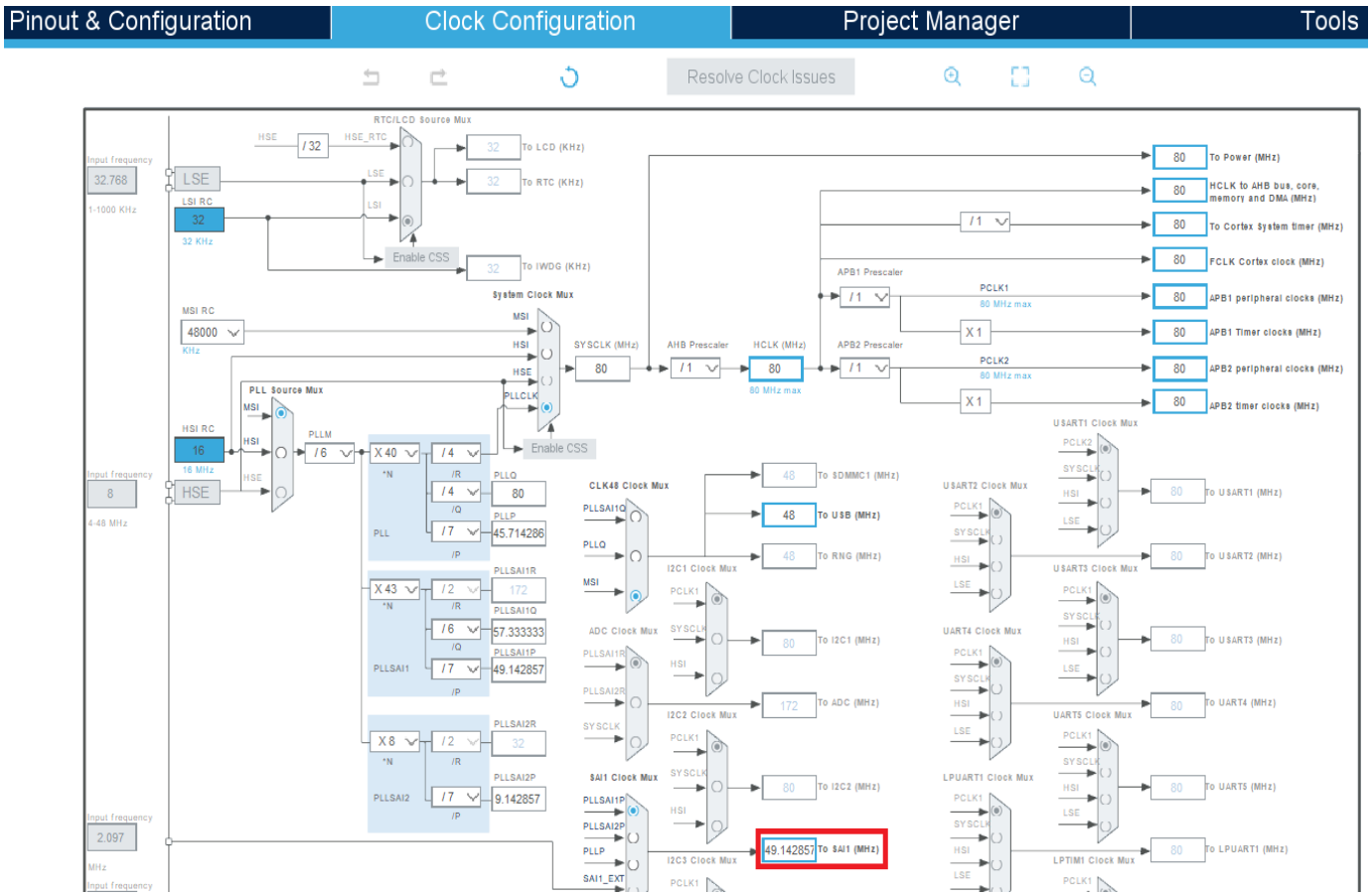


Figura 118 Configuración del reloj del sistema.

El último paso es configurar las interrupciones del sistema, que se encuentra dentro de System Core en NVIC. Cuanto más bajo el valor que se ponga más prioridad tiene la interrupción. En este caso, se ha dotado de más prioridad a la interrupción del DFSDM y posteriormente a la del USB.

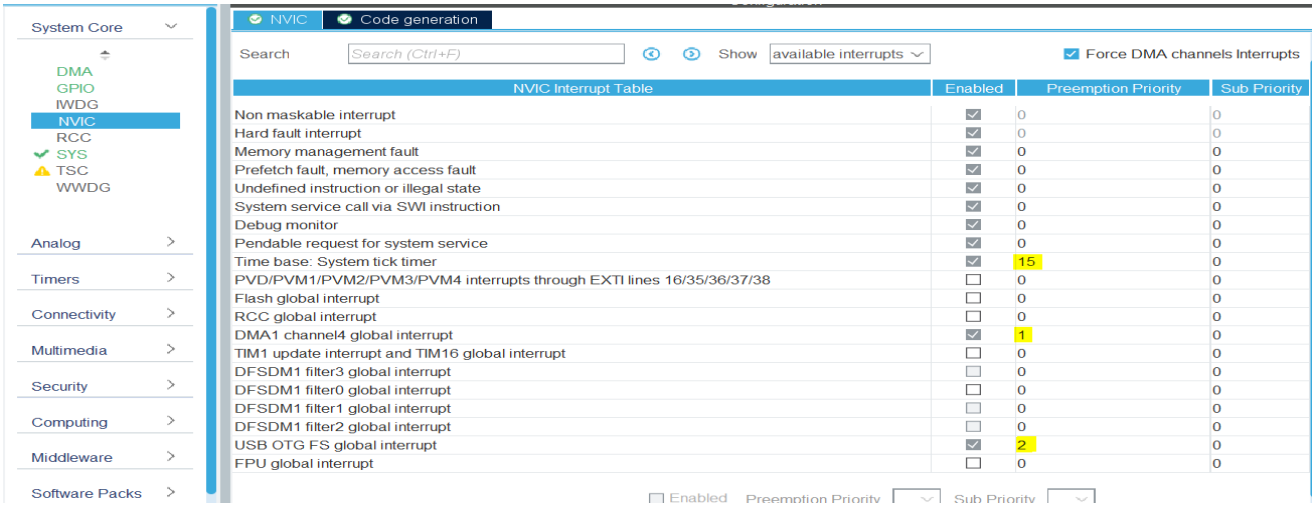


Figura 119 Configuración de las interrupciones.

Una vez que se ha terminado de configurar todos los periféricos y sus interrupciones se pulsa el botón de generar código para comenzar con la programación del sistema de detección de escenas de audio.

6. CÓDIGO FUENTE

Una vez generado el código se muestra a continuación el árbol de carpetas del proyecto, se describirán las carpetas más importantes del proyecto y se hará una explicación extensa en los dos archivos más importantes del proyecto, el main.c y el archivo asc_processing.c. A continuación, se describen las carpetas más importantes del proyecto.

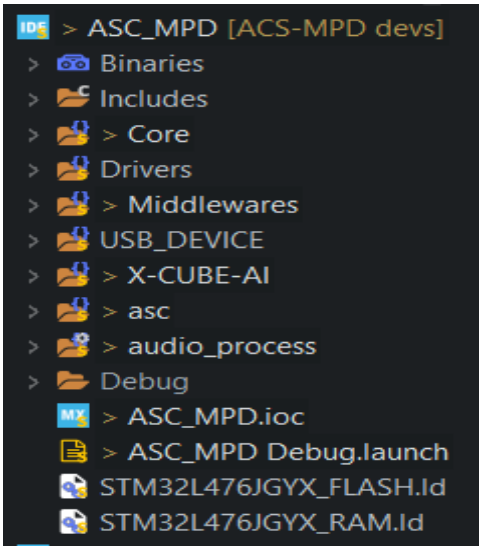


Figura 120 Árbol del proyecto.

Carpeta Core: contiene archivos de arranque y archivos del sistema necesarios para inicializar y configurar el microcontrolador, así como el entorno de ejecución básico. Dentro de la carpeta Core se encuentran dos subcarpetas que son la carpeta Inc y carpeta Src.

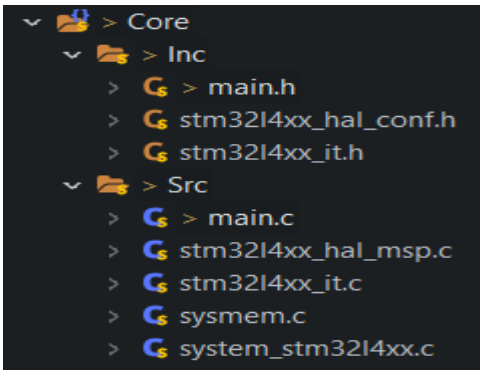


Figura 121 Carpeta Core

Sub carpeta Inc: contiene los archivos de cabecera (.h) que tienen las declaraciones de funciones, estructuras y constantes utilizadas en el proyecto.

Sub carpeta Src: contiene los archivos fuente con la implementación de las funciones y la lógica principal del proyecto. Aquí se encuentra los siguientes archivos:

- El archivo principal del proyecto, el main.c donde se desarrolla la declaración y ejecución de las funciones de la Red Neuronal.

- El archivo stm32l4xx_hal_msp, cuya función principal es implementar las funciones de inicialización y configuración de los periféricos utilizados como pines GPIO, temporizadores, UART, etc.

- El archivo stm32l4xx_it.c, el cual, se emplea para el manejo de las interrupciones.

- El archivo sysmem que es utilizado para la gestión, asignación y manejo de la memoria del microcontrolador.

- El archivo system_stm32l4xx.c, el cual permite la inicialización del sistema y la actualización del reloj principal durante la ejecución del programa. Este archivo se encarga de configurar el sistema antes de iniciar el programa principal.

Carpeta Drivers: contiene las biblioteca y controladores proporcionados por ST para poder utilizar los periféricos del microcontrolador.

Carpeta Middleware: contiene archivos adicionales que pueden ser utilizados en el proyecto, proporcionados por ST o terceros y pueden incluir protocolos de comunicación, sistemas de archivos, protocolos de red, etc.

Carpeta USB_DEVICE: contiene todos los archivos relacionados con el manejo y control del USB.

Carpeta X-CUBE-AI: esta carpeta la genera la librería X-CUBE-AI y ahí se encuentran todos los archivos de la red neuronal generados por esta librería, es decir, los parámetros de la red, los pesos, las activaciones, la definición de capas de la red, las dimensiones de los datos, etc. En definitiva, contiene estructuras de datos y constantes que definen los parámetros necesarios para la inferencia de la red neuronal en el microcontrolador.

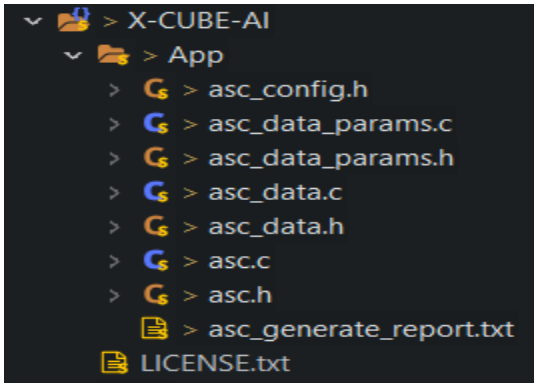


Figura 122 Carpeta X-CUBE-AI

En la **carpeta asc** se encuentran dos archivos:

- Archivo asc_featurescaler.c, es el archivo donde se guardan los valores de media y desviación típica devueltos por el script de Python durante su ejecución en Jupyter notebook, estos valores serán utilizados en la ejecución del código.

- Archivo asc_processing.c que es donde se ejecuta todo el código de la Red Neuronal y que se verá más en detalle más adelante.

- El archivo de test fue utilizado para hacer pruebas.

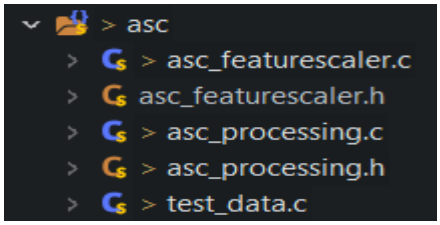


Figura 123 Carpeta asc.

Por último, hay que tener en cuenta también la carpeta audio_process, la cual está basada en el código proporcionado por ST en el *function pack* STM32CubeFunctionPack SENSING1 V4.0.3.

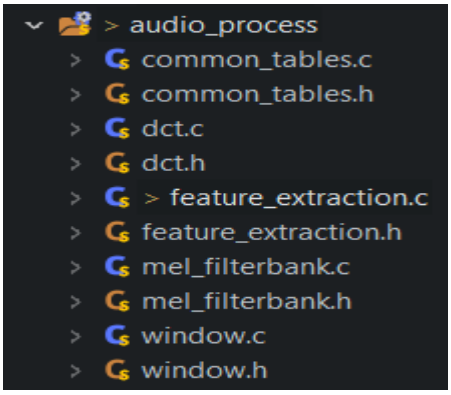


Figura 124 Carpeta audio_process.

En esta carpeta se encuentran una serie de archivos que son los siguientes:

- El archivo common_tables.c que se utilizan para el procesamiento digital de la señal proporcionando los valores de la ventana de Hanning. La ventana de Hanning es una función matemática que se aplica a la señal de audio. Su propósito principal es suavizar los bordes de cada ventana, lo que ayuda a reducir los efectos indeseables de las discontinuidades que pueden ocurrir al analizar señales de audio.

- El archivo dct.c el cual realiza la transformada coseno discreta.

- El archivo feature_extraction.c en el que se emplean las funciones para crear columnas de espectrograma, espectrograma Mel y espectrograma logarítmico Mel.

- El archivo mel_filterbank.c el cual genera un banco de filtros Mel para aplicarlo a un espectrograma para obtener las energías Mel.

- El archivo window.c que proporciona funciones para generar funciones de ventana, como la ventana Hanning.

Antes de comenzar con la explicación del archivo main.c hay que recalcar para la correcta ejecución del código la instalación en el entorno de desarrollo STM32CubeIde de la librería arm_math.h, esta librería incluye funciones para poder hacer la Tranformada Rápida de Fourier, operaciones de convolución, funciones exponenciales, trigonométricas, logarítmicas, de redondeo, etc.

El archivo main.c comienza definiendo las librerías que necesita y las variables.

```
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "usb_device.h" //libreria para el usb
#include "asc_processing.h" //libreria propia creada para el procesamiento del audio
#include <stdbool.h>  //libreria para trabajar con variables booleanas
/* USER CODE BEGIN PD */
#define AUDIO_CHANNELS        1 //canales del microfono
#define AUDIO_SAMPLING_FREQ   16000 //frecuencia de muestreo
#define AUDIO_IN_BUF_SIZE     (AUDIO_SAMPLING_FREQ/1000)*2 // tamaño del buffer de 32
#define AUDIO_VOLUME_INPUT    64U //volumen de entrada del microfono
//#define AUDIO_IN_BUF_SIZE     1024
#define SaturaLH(N, L, H) (((N)<(L))?(L):(((N)>(H))?(H):(N))) //macro empleada por ST
/* USER CODE END PD */
```

Figura 125 Definición de librerías y variables.

Continúa declarando los manejadores de los periféricos que se van a utilizar. El CRC, el filtro 0 y el canal 5 del DFSDM y el *timer*16.

```
CRC_HandleTypeDef hcrc; //manejador del hcrc
DFSDM_Filter_HandleTypeDef hdfsdm1_filter0; //manejador del filtro0 del DFSDM
DFSDM_Channel_HandleTypeDef hdfsdm1_channel5; //manejador del canal 5
DMA_HandleTypeDef hdma_dfsdm1_flt0; //manejador del filtro cero del DMA
TIM_HandleTypeDef htim16; //timer del timer 16
```

Figura 126 Declaración de manejadores de los periféricos.

Lo siguiente es la declaración de variables que van a intervenir en la captura del audio del micrófono.

```
//variables audio microfono
int32_t micRecBuff[AUDIO_IN_BUF_SIZE]; //buffer de grabacion del microfono de tamaño 32
uint16_t PCM_Buffer[AUDIO_CHANNELS * AUDIO_SAMPLING_FREQ / 1000]; //buffer para conversión
//PCM de tamaño 16 porque maneja la mitad de las muestras del micrófono
AUDIO_IN_t audioIn; //estructura para configurar el audio de entrada a la NN

bool dmaHalfBuff = false; //flag para indicar la interrupcion de mitad de buffer
bool dmaFullBuff = false;   //flag para indicar la interrupcion full de buffer
```

Figura 127 Variables para la captura del audio.

Para clarificar se definen las funciones de cada *buffer* que se utilizan:

- micRecBuff[32]: este *buffer* guarda las muestras que toma el micrófono de 32 en 32 y es utilizado en la interrupción del DMA del DFSDM para copiar al puntero pBuff.

-PCM_Buffer[16]: este *buffer* guarda la mitad de las muestras que toma el micrófono en micRecBuff y las mapea a 16 bits de profundidad.

- Proc_Buffer_f: *buffer* empleado para el procesamiento de la Transformada Rápida de Fourier, será usado cuando esté lleno Fill_Buffer.

- Fill_Buffer: *buffer* que se emplea para recibir el audio convertido a PCM por el DFSDM.

- Audio_in_t es una estructura donde se definen los parámetros de entrada del audio definida en la librería main.h. En el puntero *pBuff será asignada la dirección de memoria del PCM_buffer (encargado de almacenar el audio PCM ya mapeado a 16 bits).

```
typedef struct {
  uint32_t SampleRate; /* Audio IN Sample rate           */
  uint32_t BitsPerSample; /* Audio IN Sample resolution     */
  uint32_t ChannelsNbr; /* Audio IN number of channel     */
  uint16_t *pBuff; /* Audio IN record buffer          */
  uint32_t Size; /* Audio IN record buffer size    */
  uint32_t Volume; /* Audio IN volume                */
  uint32_t State; /* Audio IN State                 */
  HP_FilterState_t HP_Filters; /*!< HP filter state for each channel*/
  uint32_t DecimationFactor;
} AUDIO_IN_t;
```

Figura 128 Estructura AUDIO_IN_t.

El main.c continua con la declaración de las funciones que hace el CubeMx sobre los periféricos que se utilizan.

```
void SystemClock_Config(void); //función para configurar el reloj
static void MX_GPIO_Init(void); //función para inicializar los GPIO
static void MX_DMA_Init(void); //función para inicializar el DMA
static void MX_CRC_Init(void); //función para inicializar el CRC
static void MX_DFSDM1_Init(void); //función para inicializar el DFSDM
static void MX_TIM16_Init(void); //función para inicializar el Timer16
/* USER CODE BEGIN PFP */
extern uint8_t CDC_Transmit_FS(uint8_t *Buf, uint16_t Len); //función para tx por USB
//VirtualComPort, se le pasa como parámetro el buffer a tx y su longitud
```

Figura 129 Declaración de periféricos del CUBE MX

El programa se mete ahora dentro del bucle principal que es el main y lo primero que hace es definir los parámetros del audio de entrada al DFSDM a través de la estructura audioIn.

```
int main(void)
{
  /* USER CODE BEGIN 1 */
  //se configuran los parámetros del audio de entrada del periferico - DFSDM
  audioIn.BitsPerSample = 16; //16 bits por muestra porque el micro maneja un buffer de 32
  audioIn.ChannelsNbr = AUDIO_CHANNELS; // 1 canal de entrada
  audioIn.SampleRate = AUDIO_SAMPLING_FREQ; //16KHz de frec de muestreo
  audioIn.Volume = AUDIO_VOLUME_INPUT; // volumen 64
  audioIn.State = 0; // estado inicial
  audioIn.pBuff = PCM_Buffer;//se pasa el buffer con el audio ya convertido a PCM por
  //el DFSDM
```

Figura 130 Definición de parámetros del DFSDM.

El main continua con la llamada de las funciones que inicializan la librería HAL, la configuración del reloj del sistema y las funciones que cargan los valores de inicialización de los periféricos.

```
/* MCU Configuration--------------------------------------------------------*/

/* Reset of all peripherals, Initializes the Flash interface and the Systick. */
HAL_Init();
/* Configure the system clock */
SystemClock_Config();
/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_DMA_Init();
MX_CRC_Init();
MX_USB_DEVICE_Init();
MX_DFSDM1_Init();
MX_TIM16_Init();
/* USER CODE BEGIN 2 */
HAL_TIM_Base_Start(&htim16);  // Start timer/counter
//este contador se utiliza para medir el tiempo que tarda la red neuronal en
//hacer una inferencia
```

Figura 131 Funciones para librería HAL, reloj y periféricos.

Antes de comenzar el bucle infinito se introduce un retardo para que le de tiempo al PC a reconocer el puerto USB en modo virtual COM del SensorTile. A continuación, se llama a la función ASC_Init para configurar los parámetros de la red neuronal.

```
//se inicia la red neuronal
HAL_Delay(1500); //retardo para que el PC detecte el puerto USB

ASC_Init(); //se inicia la configuración de los parámetros de la red neuronal
//se inicia la captura de audio con el periferico DFSDM

//(filtro digital para moduladores sigma-delta)
HAL_DFSDM_FilterRegularStart_DMA(&hdfsdm1_filter0, micRecBuff, AUDIO_IN_BUF_SIZE);
//se le pasa como parámetro el filtro configurado en el cubeMX, el buffer
//donde el microfono almacena los datos y el tamaño de entrada del buffer
//de entrada que es 32
```

Figura 132 Retardo y llamada a ASC_Init.

Dentro de la función ASC_Init se muestra el mensaje de ASC_MPD inicializando, por favor espere y se llama a dos funciones de configuración, la función ai_boostrap y la función Preprocessing_Init, estas dos funciones se encuentran dentro del archivo asc_processing.c

```
//inicia la red neuronal convolucional ACS
void ASC_Init(void) {
  printf("ASC_MPD - initialization, please wait\r\n");

  //Se configura la NN
  ai_boostrap(data_activations0);

  //Se configura el preprocesamiento de audio, es decir la FFT,
  //el espectrograma, el filtro Mel y el espectrograma Mel
  Preprocessing_Init();
}
```

Figura 133 Función ASC_Init.

La función ai_bootstrap crea, inicializa y configura una instancia del modelo de red neuronal. Asigna los punteros de entrada y salida necesarios y maneja posibles errores durante el proceso.

```
//función para configuración de la NN
static int ai_boostrap (ai_handle *act_addr) {
  ai_error err; //variable para mostrar errores de la red

  /* Se crea y se inicializa una instancia del modelo */
  /*ai_asc_create_and_init toma como argumentos la dirección del
  puntero a la instancia asc y act_addr que contiene la dirección de memoria donde se
  almacenarán los datos del modelo. El tercer argumento es NULL y se utiliza para
  configuraciones adicionales que no se están utilizando en este caso.  */

  err = ai_asc_create_and_init(&asc, act_addr, NULL);
  if (err.type != AI_ERROR_NONE) {
    ai_log_err(err, "ai_asc_create_and_init");
    return -1; //si hay un error en la creacion e inicialización devuelve -1
  }

  /*Se obtiene un puntero al array de entradas de la red neuronal. ai_input se utilizará
   posteriormente para proporcionar los datos de entrada al modelo durante la inferencia*/
  ai_input = ai_asc_inputs_get(asc, NULL); //ai_input es el array de entrada a la Red

  /*Se obtiene un puntero al array de salidas de la red neuronal. ai_output se utilizará
   posteriormente para acceder a las predicciones realizadas por la NN*/
  ai_output = ai_asc_outputs_get(asc, NULL); //ai output es el array de salida de la Red
```

Figura 134 Función ai_boostrap.

La función ai_bootstrap comienza declarando la variable err para mostrar los posibles errores de la red y se le asigna el valor devuelto por la función ai_asc_create_and_init. Esta función crea e inicializa una instancia del modelo de red neuronal utilizando la dirección del puntero a la instancia asc y la dirección de memoria act_addr para almacenar los datos del modelo. El tercer argumento NULL se utiliza para configuraciones adicionales que no se están utilizando en este caso.

Si hubiera un error en la creación e inicialización de la red se muestra a través de la función ai_log_err que mostrará el texto “ai_asc_create_and_init” y además retorna un valor -1 para mostrar por pantalla dicho error.

Se crea el puntero de entrada a la red neuronal ai_input a través de la función ai_asc_inputs_get. Ai_input se utiliza para proporcionar los datos de entrada al modelo durante la inferencia de la red neuronal.

De la misma forma, se obtiene un puntero al array de salidas de la red neuronal ai_output a través de la función ai_asc_outputs_get. Ai_output proporciona los valores de salida de las predicciones de la Red Neuronal.

La función continúa a través de un bucle for asignando los punteros de entrada data_ins los valores almacenados en el puntero ai_input, es decir, los parámetros de la red, los pesos, los *bias*, las funciones de activación…etc.

```
#if defined(AI_ASC_INPUTS_IN_ACTIVATIONS)
  /*  In the case where "--allocate-inputs" option is used, memory buffer can be
   *  used from the activations buffer. This is not mandatory.
   */
  for (int idx = 0; idx < AI_ASC_IN_NUM; idx++) {
    data_ins[idx] = ai_input[idx].data;//comienza a llenar la red con los pesos,
    //los bias, las funciones de activacion...
  }
#else
  for (int idx=0; idx < AI_ASC_IN_NUM; idx++) {
    ai_input[idx].data = data_ins[idx];
  }
#endif

#if defined(AI_ASC_OUTPUTS_IN_ACTIVATIONS)
  /*  In the case where "--allocate-outputs" option is used, memory buffer can be
   *  used from the activations buffer. This is no mandatory.
   */
  /*se asignan los punteros a los datos de salida del modelo (data_outs[idx]) a los
   correspondientes punteros en ai_output. Esto permite que el resultado de la inferencia
   se almacene en los punteros de salida*/
  for (int idx = 0; idx < AI_ASC_OUT_NUM; idx++) {
    data_outs[idx] = ai_output[idx].data;
  }
#else
  for (int idx=0; idx < AI_ASC_OUT_NUM; idx++) {
  ai_output[idx].data = data_outs[idx];
  }
#endif

  return 0;
}
```

Figura 135 Contenido de la función ai_boostrap.

De la misma forma hace lo mismo pero con los punteros de salida. También a través de un bucle for asigna los punteros de salida data_outs los valores almacenados en el puntero ai_output. Esto permite que el resultado de las predicciones de la Red Neuronal se almacene en los punteros de salida.

Por último, la función devuelve un cero para que se continue con la ejecución de la función Preprocessing_Init. Esta función se encarga de configurar la Transformada rápida de Fourier para realizarla sobre 1024 muestras.

```
static void Preprocessing_Init(void) {

  /* Inicialización RFFT */
  arm_rfft_fast_init_f32(&S_Rfft, 1024); //se configura la FFT con un tamaño de 1024

  /* Init Spectrogram */
  S_Spectr.pRfft = &S_Rfft;
  S_Spectr.Type = SPECTRUM_TYPE_POWER;//espectograma en potencia
  S_Spectr.pWindow = (float32_t*) hannWin_1024; //ventana de Hanning
  S_Spectr.SampRate = 16000; //frecuencia de muestreo
  S_Spectr.FrameLen = 1024; //longitud de las muestras
  S_Spectr.FFTLen = 1024; //longitud de la FFT
  S_Spectr.pScratch = aWorkingBuffer1; //buffer de trabajo

  /* Inicialización del filtro Mel*/
  S_MelFilter.pStartIndices = (uint32_t*) melFiltersStartIndices_1024_30;
  S_MelFilter.pStopIndices = (uint32_t*) melFiltersStopIndices_1024_30;
  S_MelFilter.pCoefficients = (float32_t*) melFilterLut_1024_30;
  S_MelFilter.NumMels = 30;// 30 filas

  /* Inicializacion del MelSpectrogram */
  S_MelSpectr.SpectrogramConf = &S_Spectr;
  S_MelSpectr.MelFilter = &S_MelFilter;
}
```

Figura 136 Función Preprocessing_Init.

El siguiente paso es inicializar los parámetros para que se pueda calcular el espectrograma. Como el tipo de espectro en potencia, ventana de Hannwin, frecuencia de muestreo de 16 Khz, longitud de muestreo y de la Transformada Rápida de Fourier de 1024 muestras y el *buffer* para trabajar con estos valores que es aWorkingBuffer1.

Se inicializan también los parámetros necesarios para el filtro Mel, el índice de comienzo, el de final, los coeficientes y el número de filas.

Por último, se inicializa la configuración del Espectrograma Mel y del filtro Mel.

El programa continúa en el main.c y llama a la función HAL_DFSDM_FilterRegularStart_DMA para iniciar la captación de audio del DFSDM a través del micrófono, por tanto el DFSDM ya está convirtiendo el audio PDM a PCM. Esta conversión se hace a través de la llamada de interrupción del DMA. Hay que recordar que el DMA iba a ofrecer una interrupción cuando llenase el *buffer* a la mitad y cuando lo tuviera completamente lleno (trabaja con un bucle circular).

El *callback* para el llenado de la mitad del *buffer* es HAL_DFSDM_FilterRegConvHalfCpltCallback y para el *buffer* completo es HAL_DFSDM_FilterRegConvCpltCallback. Los dos *callback* hacen el mismo código.

Estos *callbacks* comprueban que la interrupción la haya provocado el filtro cero del DMA si es así, se aplica un filtro paso alto para eliminar el ruido provocado por la conversión de PDM a PCM. El último paso a través de la macro SaturaLH se limita el valor de salida entre -32760 y 32760 para que la profundidad del audio PCM sea de 16 bits. Después se convierte a valor entero sin signo de 16 bit y se almacenan las muestras de audio dentro de la estructura audioIn en Pbuff.

El último paso de esta parte del código es indicar que la mitad del *buffer* se ha llenado activando el *flag* de mitad de *buffer*.

```
//callback almacenamiento mitad de buffer del DMA cada 16 muestras.
//filtra las muestras y normaliza el rango de valor entre +/- 32760
void HAL_DFSDM_FilterRegConvHalfCpltCallback(DFSDM_Filter_HandleTypeDef *hdfsdm_filter)
{
  if (hdfsdm_filter == &hdfsdm1_filter0) {
    for (uint32_t i = 0; i < (audioIn.SampleRate / (uint32_t) 1000); i++) {
      audioIn.HP_Filters.Z = ((micRecBuff[i] / 256) * (int32_t) (audioIn.Volume)) / 128;
      audioIn.HP_Filters.oldOut = (0xFC
          * (audioIn.HP_Filters.oldOut + audioIn.HP_Filters.Z
              - audioIn.HP_Filters.oldIn)) / 256;
      audioIn.HP_Filters.oldIn = audioIn.HP_Filters.Z;
      audioIn.pBuff[(i * audioIn.ChannelsNbr)] = (uint16_t) (SaturaLH(
          audioIn.HP_Filters.oldOut, -32760, 32760));
    }
    dmaHalfBuff = true;
  }
}
```

Figura 137 Callback del DFSDM para mitad de buffer.

```
//callback almacenamiento completo de buffer del DMA
//filtra las muestras y normaliza el rango de valor entre +/- 32760
void HAL_DFSDM_FilterRegConvCpltCallback(
    DFSDM_Filter_HandleTypeDef *hdfsdm_filter) {
  if (hdfsdm_filter == &hdfsdm1_filter0) {
    for (uint32_t i = 0; i < (audioIn.SampleRate / (uint32_t) 1000); i++) {
      audioIn.HP_Filters.Z = ((micRecBuff[i
          + (audioIn.SampleRate / (uint32_t) 1000)] / 256)
          * (int32_t) (audioIn.Volume)) / 128;
      audioIn.HP_Filters.oldOut = (0xFC
          * (audioIn.HP_Filters.oldOut + audioIn.HP_Filters.Z
              - audioIn.HP_Filters.oldIn)) / 256;
      audioIn.HP_Filters.oldIn = audioIn.HP_Filters.Z;
      audioIn.pBuff[(i * audioIn.ChannelsNbr)] = (uint16_t) (SaturaLH(
          audioIn.HP_Filters.oldOut, -32760, 32760));
    }
    dmaFullBuff = true;
  }
}
```

Figura 138 Callback del DFSDM para buffer completo.

El siguiente paso dentro del main.c es entrar en el bucle infinito donde se ejecuta la función ASC_Process, la cual se encuentra desarrollada en el archivo asc_processing.c.

```
  while (1) {
    //se procesa el audio y pone a trabajar la NN
    ASC_Process();
      }
  /* USER CODE END 3 */
}
```

Figura 139 Bucle infinito.

Hay que tener en cuenta la función que va a mostrar los datos por pantalla, es la función int _write(int file, char *ptr, int len) . Esta función permite transferir la función printf estándar de lenguaje C para cadenas de texto hacia el puerto USB del microcontrolador, mediante la función CDC_Transmit_FS para transmitir los datos contenidos en el búfer ptr a través del sistema de comunicación serie. CDC_Transmit_FS es una función específica de la biblioteca del dispositivo de comunicación clase CDC (Communication Device Class) utilizada comúnmente para comunicaciones por USB en modo puerto COM virtual.

```
int _write(int file, char *ptr, int len) {
  CDC_Transmit_FS((uint8_t*) ptr, len);
  return len;
}
```

Figura 140 Función _write.

Antes de comenzar con la explicación de la función ASC_Process, se va a detallar el contenido del archivo asc_processing. El archivo comienza con la definición de librerías necesarias para la ejecución del código.

```
/* Includes ------------------------------------------------------------------*/
#include <stdlib.h>
#include <stdio.h>
#include <inttypes.h>
#include <string.h>

#include "main.h"
#include "ai_datatypes_defines.h"
#include "ai_platform.h"
#include "asc.h"
#include "asc_data.h"
#include "asc_processing.h"
#include "asc_featurescaler.h"
#include "feature_extraction.h"
```

Figura 141 Librerias del archivo ASC_Process.

Se continúa definiendo las constantes como el número de muestras para hacer la Transformada de Fourier y el número de filas y columnas del espectrograma.

```
#define NFFT             FILL_BUFFER_SIZE //nº de muestras 1024 para hacer la FFT
#define NMELS            30 //nº de filas del espectrograma Mel

#define SPECTROGRAM_ROWS NMELS //nº de filas 30 del espectrograma Mel
#define SPECTROGRAM_COLS 32 //nº de columnas 32 para el espectrograma Mel
```

Figura 142 Definición de constantes para FFT y Espectrograma.

Se definen las variables externas utilizadas en el main que son necesarias en el archivo asc_processing.c

```
//variables externas definidas en main.c
extern TIM_HandleTypeDef htim16; //timer para calcular las inferencias de la NN.
//extern float sine2000_1[];
extern int32_t micRecBuff[]; //buffer de grabacion del microfono de tamaño 32
extern uint16_t PCM_Buffer[]; //buffer para conversión del audio PCM tamaño 16. Recibe
//el audio filtrado por el filtro paso alto del DFSDM.
extern AUDIO_IN_t audioIn; //estructura para configurar el audio de entrada a la NN
extern bool dmaHalfBuff;  //flag para indicar la interrupcion de mitad de buffer
extern bool dmaFullBuff;  //flag para indicar la interrupcion full de buffer
volatile uint16_t contador = 0; //variable para no mostrar las primeras predicciones
bool predict = false; // variable para gestionar las predicciones de la NN

//variables de procesamiento de audio
float32_t Proc_Buffer_f[FILL_BUFFER_SIZE]; //Buffer de procesamiento de la FFT de 1024
//es de 1024 porque a la NN siempre le va a entrar un buffer de 1024.
int16_t Fill_Buffer[FILL_BUFFER_SIZE];//Buffer que se usa para recibir el audio convertido
//a PCM y que se le ha pasado el filtro paso alto del DFSDM
//Cuando esté lleno Fill_Buffer se utilizará Proc_Buffer_f para calcular la FFT
//esto se hace así para tener buffer separados y así evitar solapamientos en los datos
//de los buffer, los del DFSDM y los de la FFT que será lo que se entregue a la NN
static uint32_t index_buff_fill = 0;//indica cuando el buffer Fill_Buffer, el cual
//se llena de 16 en 16
```

Figura 143 Definición de variables necesarias en el asc_processing.

Se definen los *arrays* y estructuras que intervienen en el procesamiento y confección del espectrograma, espectrograma Mel y filtros.

```
static float32_t aSpectrogram[SPECTROGRAM_ROWS * SPECTROGRAM_COLS];
//espectrograma amplitud de 30x32 que se llena con aColBuffer.
static float32_t aColBuffer[SPECTROGRAM_ROWS]; //Buffer para almacenar las columnas del
//espectrograma, aSpectrogram se llena 32 veces con aColBuffer.
static uint32_t SpectrColIndex; //indice para las columnas del espectrograma
float32_t aWorkingBuffer1[NFFT]; //buffer de tamaño 1024

static arm_rfft_fast_instance_f32 S_Rfft; //variable para hacer la FFT
static MelFilterTypeDef S_MelFilter; //filtro Mel
static SpectrogramTypeDef S_Spectr; //espectrograma en voltaje
static MelSpectrogramTypeDef S_MelSpectr; //espectrograma Mel
```

Figura 144 Definición de variables necesarias en el asc_processing.

También se definen las variables para la impresión por pantalla de los estados del ventilador y las que intervienen para guardar los resultados de las predicciones hechas por la Red Neuronal.

```
char buf[100]; //buffer para la impresión por USB
int buf_len = 0; //variable que indica la cantidad de caracteres a imprimir por USB
uint32_t timestamp; //variable para ver lo que tarda en hacer una inferencia la red
char label[AI_ASC_OUT_1_SIZE + 1][10] = { "off", "on", "obstruct", "???" };
//definición de los estados posibles del ventilador
uint8_t label_index; // indice de las etiquetas, 0 es off, 1 es on, etc
float out_val[AI_ASC_OUT_1_SIZE]; // se almacena el valor de la predicción de la NN
float out_sum[AI_ASC_OUT_1_SIZE]; // se almacenan las predicciones de la NN
float out_prom[AI_ASC_OUT_1_SIZE]; //se almacena el promedio de todas las predicciones
bool run_asc = false; // variable que indica cuando puede funcionar la NN
```

Figura 145 Definición de variables para impresión por pantalla y procesar las predicciones.

Se continúa definiendo los punteros a los *buffers* de entrada y salida de la Red Neuronal.

```
//AI networks variables
/* IO buffers ----------------------------------------------------------------*/

#if !defined(AI_ASC_INPUTS_IN_ACTIVATIONS)
AI_ALIGNED(4) ai_i8 data_in_1[AI_ASC_IN_1_SIZE_BYTES];
ai_i8* data_ins[AI_ASC_IN_NUM] = {
data_in_1
};
#else
ai_i8 *data_ins[AI_ASC_IN_NUM] = { //espectrograma de entrada de la red neuronal
NULL };
#endif

#if !defined(AI_ASC_OUTPUTS_IN_ACTIVATIONS)
AI_ALIGNED(4) ai_i8 data_out_1[AI_ASC_OUT_1_SIZE_BYTES];
ai_i8* data_outs[AI_ASC_OUT_NUM] = {
data_out_1
};
#else
ai_i8 *data_outs[AI_ASC_OUT_NUM] = { //salida de la NN tamaño 3 posiciones
NULL };
#endif
```

Figura 146 Definición de buffers de entrada y salida de la Red Neuronal.

Se definen los *buffers* de activación de la Red Neuronal, el manejador de la Red Neuronal (asc), y los punteros a los *buffers* de entrada y salida de la Red Neuronal.

```
/* Activations buffers -------------------------------------------------------*/
AI_ALIGNED(32)
static uint8_t pool0[AI_ASC_DATA_ACTIVATION_1_SIZE]; //array con 19328 activaciones
ai_handle data_activations0[] = { pool0 }; // activaciones de la red

/* AI objects ----------------------------------------------------------------*/

static ai_handle asc = AI_HANDLE_NULL; //  este handle gestiona el estado de la NN
//si fue inicializada, si está haciendo una inferencia, etc
static ai_buffer *ai_input; //puntero al buffer de entrada de la red neuronal
static ai_buffer *ai_output; //puntero al buffer de salida de la red neuronal
```

Figura 147 Definición de buffers de activación, manejador y punteros a los buffers de entrada y salida.

Se definen los prototipos de todas las funciones que se utilizan en este archivo asc_processing.c.

```
/* Private function prototypes -----------------------------------------------*/
static void Preprocessing_Init(void); // función donde se hace la FFT y se calcula
//el espectrograma LogMel

static void PowerTodB(float32_t *pSpectrogram); //transformación del espectrograma a dB
static void ai_log_err(const ai_error err, const char *fct);//imprimir errores de la NN
static int ai_boostrap(ai_handle *act_addr); //configuración de la NN
int acquire_and_process_data(ai_i8 *data[]); // crea las columnas del espectrograma,
//lo convierte a dB, realiza la estandarización de características y llena el buffer de
//entrada que va a la red neuronal.

static int ai_run(void); //lanza el procesamiento de la NN
void AudioProcess_FromMics(void); //Procesamiento de audio de los microfónos
static void AudioProcess(void); //Procesamiento de audio para seguir adquiriendo
//muestras en Fill_Buffer y para transferir a ProcBuffer_f
int post_process(ai_i8 *data[]); //procesa la salida NN para enviar la info por USB
extern uint8_t CDC_Transmit_FS(uint8_t *Buf, uint16_t Len);//función para imprimir por USB
```

Figura 148 Funciones que se utilizan en asc_processing.c

Continuando con la ejecución del programa en el main.c, cuando se llega al bucle infinito while (1) se llama a la función Asc_Process.

Esta función se estructura de la siguiente manera:

1. Primero define la variable “res” que es la que permitirá ir comprobando los diferentes estados de la red neuronal y es la que se utilizará para ver si hay algún error en la red neuronal.
2. Se comprueba si el manejador de la red neuronal “asc” es diferente de nulo (NULL), es decir, que si contiene datos, y en ese caso se ejecuta un bucle do-while cuya condición de salida es que “res” sea distinta de cero, es decir, que se produzca un error en la red neuronal.
3. Una vez dentro del bucle do-while se comprueban tres condiciones:
    a. Si la mitad del *buffer* del DMA se ha llenado.
    b. Si la otra mitad del *buffer* del DMA se ha llenado (*buffer* completo)
    c. Si la red neuronal se ha activado porque ya estén listos los espectrogramas y tenga que procesarlos (run_asc=true).

```
//procesa las muestras de audio capturadas y realiza la prediccion
void ASC_Process(void) {
  int res = 0; //variable para conocer el estado en el que se encuentra la NN. Si hay
               //un error se muestra a través de ella.

  if (asc) { //se comprueba si el handle de la NN es correcto, si es así entra en el if
    do {
      //se procesa el audio PCM, en la primera mitad del buffer
      if (dmaHalfBuff == true) {
        AudioProcess_FromMics();
        dmaHalfBuff = false;
      }

      //se procesa el audio PCM, en la segunda mitad del buffer
      if (dmaFullBuff) {
        AudioProcess_FromMics();
        dmaFullBuff = false;
      }
      if (run_asc == true) { //cuando ya se han llenado los buffers se pone a true
        //y se ejecuta la red para realizar la prediccion
        HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin);
```

Figura 149 Función ASC_Process

El programa comenzará a ejecutarse e irá entrando en la condición a) y b) llenando su *buffer* circular y transfiriendo sus muestras de 16 en 16 al Fill_Buffer.

Si se cumple la condición a) o b), es decir, la mitad del *buffer* del DMA se ha llenado o el *buffer* completo del DMA, se llama a la función AudioProcess_FromMics, la cual, copia los datos de audio del PCM_Buffer al Fill_Buffer en cada llamada, avanzando la posición de inicio en el Fill_Buffer en 16 elementos cada vez.

```
void AudioProcess_FromMics(void) {
  // Copia PCM_Buffer del micrófono en Fill_Buffer
  memcpy(Fill_Buffer + index_buff_fill, PCM_Buffer, sizeof(int16_t) * 16);
  //se copia en memoria PCM_Buffer a Fill_buffer en cada llamada a la función
  index_buff_fill += 16; //vamos llenando el buffer de llenado de 16 en 16.

  AudioProcess(); //funcion para copiar Fill_Buffer en Proc_Buffer_f
}
```

Figura 150 Función AudioProcess_FromMics

Siempre que se entre a esta función se llama a la función AudioProcess, que procesará el audio si el índice de llenado de *buffer* "index_buff_fill" es 1024. En caso contrario esta función no hace nada.

```
static void AudioProcess(void) {
  float32_t sample; //variable para copiar el Fill_Buffer a Proc_Buffer_f
  //esto se hace así porque el Fill_Buffer es un buffer circular y así cuando
  //esté lleno se pasa al Proc_Buffer_f y así se evita que se solapen muestras

/*Se crea una ventana de 64 ms (1024 muestras) cada 32 ms (512 muestras)
La extracción de características del audio se ejecuta cada 32 ms en una ventana
de 64 ms (50 % de superposición) */

    //si el índice no es 1024 no se copia fill_buffer en proc_buffer
  if (index_buff_fill == FILL_BUFFER_SIZE) {
    // Se copia Fill Buffer en Proc Buffer_f(buffer que se le hace la FFT)
    for (uint32_t i = 0; i < FILL_BUFFER_SIZE; i++) {
      sample = ((float32_t) Fill_Buffer[i]);//se pasa de entero a float para trabajar
      //posteriorme la FFT
  /* realiza una conversión de escala de las muestras de audio de tipo int16_t a valores
     de punto flotante en el rango normalizado de -1 a 1. Se hace para asegurar
     que los datos estén en el rango antes de aplicar la FFT*/
      sample /= (float32_t) ((1 << (8 * sizeof(int16_t) - 1)));
      Proc_Buffer_f[i] = sample; //se copian las muestras a Proc_Buffer_f
    }

    // Se desplaza hacia la izquierda Fill Buffer 512 muestras
    memmove(Fill_Buffer, Fill_Buffer + (FILL_BUFFER_SIZE / 2),
        sizeof(int16_t) * (FILL_BUFFER_SIZE / 2));
    index_buff_fill = (FILL_BUFFER_SIZE / 2); //se fija en 512 el indice

//se activa la bandera para ejecutar la NN porque ya está el buffer completamente lleno
    run_asc = true;
  }
}
```

Figura 151 Función AudioProcess.

Dentro de esta función lo primero que se hace es declarar la variable sample de tipo float32 para almacenar una muestra de audio. Esta variable se utiliza para pasar datos del Fill_Buffer al Proc_Buffer_f para su posterior procesamiento.

Lo siguiente es comparar si index_buff_fill es 1024, si es así, el Fill_Buffer está lleno y listo para ser procesado. Se entra en un bucle for que recorre cada elemento del Fill_Buffer y se copia a la variable sample haciendo una conversión previa a float32. Después se hace una normalización de los valores de las muestras de audio en el rango [-1 1].

Por último, se asigna el valor de sample a cada posición del array Proc_Buffer_f, array que será utilizado para aplicar la transformada de Fourier y se sale del bucle for.

Fuera del bucle for anterior se realiza un desplazamiento de las muestras en el Fill_Buffer. Esto implica copiar la segunda mitad del Fill_Buffer hacia la primera mitad, descartando la primera mitad original. Esto se hace para mantener la naturaleza circular del Fill_Buffer, donde las muestras más antiguas se eliminan y se agregan nuevas muestras al final.

El penúltimo paso es actualizar el índice de index_buff_fill para indicar la nueva posición de inicio en el Fill_Buffer después del desplazamiento anterior.

El último paso es activar el flag que indica que se active la red neuronal “run_asc” y se vuelve a la función ASC_Process.

```
      if (dmaFullBuff) {
        AudioProcess_FromMics();
        dmaFullBuff = false;
      }
      if (run_asc == true) { //cuando ya se han llenado los buffers se pone a true
        //y se ejecuta la red para realizar la prediccion
        HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin);
        //se activa el led para indicar que la red está funcionando

        //estos pasos están copiados de la plantilla
        /* 1 - acquire and pre-process input data */
        res = acquire_and_process_data(data_ins); //crea las columnas del espectrograma,
        //lo convierte a dB, realiza la estandarización de características y llena el
        //buffer de entrada que va a la red neuronal.

        /* 2 - process the data - call inference engine */
        //timestamp = htim16.Instance->CNT; //capturo el tiempo de inicio para ver cuanto
        //tiempo le lleva a la NN el cálculo

        if (res == 0) res = ai_run();//se inicia el funcionamiento de la red

        /* 3- post-process the predictions */
        if (res == 0) res = post_process(data_outs); ////procesa la salida NN
        //para enviar la info por USB
```

Figura 152 Función ASC_Process.

Si run_asc es “true” se pone el led naranja del Sensor Tile a parpadear para indicar que la red neuronal está activa y se ejecuta la función “acquire_and_process_and_data”.

```
int acquire_and_process_data(ai_i8 *data[]) {
  // Crea una columna de espectrograma de escala de Mel
  //se le pasa el buffer de procesamiento y el buffer de columna
  MelSpectrogramColumn(&S_MelSpectr, Proc_Buffer_f, aColBuffer);

  // Se copia la columna correspondiente en el espectrograma de salida
  for (uint32_t i = 0; i < NMELS; i++) {
    aSpectrogram[i * SPECTROGRAM_COLS + SpectrColIndex] = aColBuffer[i];
  }
  SpectrColIndex++; //una vez copiados los valores se incrementa el índice

  if (SpectrColIndex == SPECTROGRAM_COLS) { //si indice de columna es 32
    SpectrColIndex = 0; //se inicializa el indice a cero

    // Se convierte el espectrograma a decibelios
    PowerTodB(aSpectrogram);

// estandarización de las características del espectrograma como hace el código de python
    for (uint32_t i = 0; i < SPECTROGRAM_ROWS * SPECTROGRAM_COLS; i++) {
      aSpectrogram[i] = (aSpectrogram[i] - featureScalerMean[i])
          / featureScalerStd[i];
    }
  }
  // fill the inputs of the c-model
  //Llenado del buffer para el procesamiento de la red neuronal
  for (int idx = 0; idx < AI_ASC_IN_1_SIZE; idx++) {
    //((ai_float*) *data)[idx] = (ai_float) sine2000_1[idx];
    ((ai_float*) *data)[idx] = (ai_float) aSpectrogram[idx];
  }
  return 0;
}
```

Figura 153 Función acquire_and_process_and_data.

A partir de aquí se empiezan a ejecutar ciertos pasos y funciones que se venían ejecutando en el script de Python. Se crea una columna del espectrograma Mel a través de la función MelSpectrogramColumn. Esta función aplica la Transformada de Fourier a Proc_Buffer_f y aplica los filtros Mel para ir generando las columnas del espectrograma Mel que se almacenan en aColBuffer.

Cada columna generada por MelSpectrogramColumn se copia a aSpectrogram mediante un bucle for. En aSpectrogram se va conformado el espectrograma que será pasado a la red neuronal.

Ahora se realiza la estandarización de las características que también se hizo en el script de Python, por ello se aprovechan los valores que devolvió el script y que se encuentran en el archivo asc_featurescaler.c dentro del árbol del proyecto.

El último paso que hace esta función es copiar todas las columnas de aSpectrogram en el puntero a puntero data como *buffer* de entrada a la red neuronal.

Como la función acquire_and_process_data devuelve un cero se pasa a la siguiente condición del if de ASC_Process que es entrar en la función ai_run.

```
static int ai_run(void) { //función para que la NN comience a operar
  ai_i32 batch;

  batch = ai_asc_run(asc, ai_input, ai_output);
  /*recibe el puntero a la intancia del modelo (asc), el puntero a los datos de entrada
  y el puntero a los datos de salida que proporciona la red neuronal. La función devuelve
   un valor que indica el número de muestras procesadas durante la inferencia.*/

  if (batch != 1) { //si bacht es distinto de 1 es que hay un error
    ai_log_err(ai_asc_get_error(asc), "ai_asc_run");
    return -1;
  }

  return 0;
}
```

Figura 154 Función ai_run.

Esta función define la variable batch del tipo ai_32 y en ella guarda lo que devuelve la función ai_asc_run que es el número de muestras procesadas por la red durante la inferencia. La función ai_asc_run recibe los punteros a las instancias del modelo (asc), el puntero a los datos de entrada y el puntero a los datos de salida. La función ai_asc_run, reenvía la llamada a ai_platform_network_process con los mismos argumentos de entrada y salida para procesar la red neuronal. La función ai_platform_network_process es proporcionada por la librería X-CUBE-AI y no se muestra su contenido en esta memoria.

Dentro de la función ai_run se comprueba el valor de batch, y si no es igual a 1 implica que ocurrió un error durante la inferencia. Si esto es así se llama a la función ai_log_err para mostrar por pantalla el mensaje “ai_asc_run” y la función retorna el valor -1 que hará que el programa salga del bucle do-while de la función ASC_process y muestre el error.

En caso de no haber errores retorna un cero y el programa vuelve a la función ASC_Process y ejecutaría la función post_process.

```
//funcion para procesar las predicciones y hacer la media de las predicciones
int post_process(ai_i8 *data[]) {
    contador++; //contador de predicciones realizadas por la NN
  for (int idx = 0; idx < AI_ASC_OUT_1_SIZE; idx++) {
    out_val[idx] = ((float*) *data)[idx] * 100;// por 100 para tenerlo en porcentaje
    out_sum[idx] += out_val[idx]; //se van sumando las predicciones de cada clase
    if (contador >= 20) { //si contador es 20, se calcula el promedio de las predicciones
      out_prom[idx] = out_sum[idx] / contador; //se guardan todos los promedios
      out_sum[idx] = 0; //se inicializa el array donde se guardan las predicciones.
    }
  }

  //se asigna ya la posición máxima y por eso se empieza evaluar desde 1 para ver si las
  //otras dos son mayores

  float max_out = out_prom[0];
  label_index = 0;
  //se compara la posicion 0 con las otras dos posiciones para ver cual es mayor
  for (uint32_t i = 1; i < AI_ASC_OUT_1_SIZE; i++) {
    if (out_prom[i] > max_out) {
      max_out = out_prom[i]; //se asigna la posición mayor como predicción válida
      label_index = i; //la posición se corresponde con el estado del ventilador
      //off, on u obstruido
    }
  }

  //si la probabilidad de max_out es inferior a 50.1 es un estado desconocido
  if (max_out < 50.1) label_index = 3;
  return 0; //se devuelve un cero para continuar ejecutando la NN.
}
```

Figura 155 Función post_process.

La función post_process cuenta con una variable contador encargada de contar el número de predicciones que hace la red. Se utiliza para saber cuándo se pueden mostrar por pantalla las predicciones, así como una vez llegado a 20 se calcula su media.

Una vez incrementado el valor de contador, se inicia un bucle for en el que se van recorriendo las tres posiciones que tiene el puntero al *buffer* de salida de la red donde se almacena el valor decimal de las tres posibles situaciones en las que se encuentra el ventilador (apagado, encendido u obstruido). Este valor se multiplica por 100 para tenerlo en porcentaje y se copian las tres predicciones al array out_val.

El paso siguiente es ir guardando en las tres posiciones de out_sum los valores de las predicciones que arroja la red en cada iteración, es decir:

Iteración 1: out sum[]=[60 20 20]

Iteración 2: si out_val []= [50 30 20] entonces out_sum []=[60+50 30+20 20+20] = [110 50 40].

Cuando se llegue a la inferencia número 20, se entra en el if y se calcula el valor promedio de los tres estados en los que se encuentra el ventilador al dividir los valores almacenados en las 3 posiciones de out_sum por 20. El resultado se guarda en el array out_prom.

A continuación, se fija la posición cero de out_prom como el valor más alto de las predicciones y en un bucle for que recorre las otras dos posiciones se compara el valor de la posición 1 y de la posición 2 con el valor de la posición cero de out_prom. El valor que resulte mayor se almacena en la variable max_out y se asigna a la variable label_index la posición donde está la predicción mayor. Esto es así porque el valor de label_index va ligado al estado del ventilador, si es cero está apagado, si es 1 está encendido y si es 2 está obstruido.

Por último, se comprueba si el valor máximo de la predicción es menor al 50.1%, si es así no es una predicción válida y se asigna el valor a label_index de 3, que se corresponde con un estado de indeterminado (???).

Se devuelve un return 0 para que continue la función ASC_Process.

El programa continúa y se prepara para mostrar los resultados de las predicciones por pantalla.

```
        // 4- Print output of neural network along with inference time (microseconds)

        if (contador >= 20) { //si ya se ha hecho la media de las 20 predicciones
          if (predict == true) { //Si ya puede mostrar predicciones predict es true
            buf_len = sprintf(buf, "Fan_status: %s\r\n", label[label_index]);
            /*buf_len = sprintf(buf, "%.2f%%, %.2f%%, %.2f%% | %s\r\n", out_prom[0],
             out_prom[1], out_prom[2], label[label_index]);*/
            CDC_Transmit_FS((uint8_t*) buf, buf_len); //imprimo por USB
            contador = 0; //se inicializa el contador de predicciones
          }
          if (contador > 70) { //las 70 primeras predicciones no se muestran
            printf("ASC run - main loop\r\n");
            predict = true; //avisamos que ya puede mostrar las predicciones en pantalla
            contador = 0; //se inicializa el contador de predicciones
          }
        }
        run_asc = false; //se pone a cero para que el Fill_Buffer se vuelva a llenar
      }
    } while (res == 0); //si no hay errores en la NN permanece en el bucle do while
  }
```

Figura 156 Función ASC_Process.

Si contador es 20, es que se han hecho las primeras 20 predicciones y si además el flag predict está en true, quiere decir que ya se pueden mostrar las predicciones y se envía por pantalla el resultado de la predicción.

El flag predict se utiliza para saber si ya se han realizado las primeras 70 predicciones de la red, las cuales no se muestran porque suelen ser erróneas. Esto es así porque la red tiene que estabilizar sus pesos, bias, y demás parámetros para poder realizar las predicciones de manera correcta y este proceso toma un tiempo determinado por la complejidad del modelo de la red.

La última parte de la función ASC_Process es ver si la variable res es distinta de 0, lo que implicará que hay algún error y se mostrará el texto “Process has FAILED”).

```
  if (res!=0) { //si se llenó mal el buffer o si ocurrió algún error se muestra este error
    ai_error err = { AI_ERROR_INVALID_STATE, AI_ERROR_CODE_NETWORK };
    ai_log_err(err, "Process has FAILED");
  }
}
```

Figura 157 Función ASC_Process.

7. PRESUPUESTO

En este capítulo se desglosa los costes del proyecto, se realiza un desglose de los cotes del proyecto tanto de hardware, como de software y como de mano de obra.

7.1. Coste de Software

El desarrollo de este Trabajo Fin de Grado se ha realizado con los siguientes programas:

- Python: licencia gratuita.
- Jupyter Notebook: licencia gratuita.
- STM32CubeIde: licencia gratuita.
- TeraTerm: licencia gratuita.
- Microsoft Office: licencia gratuita gracias a la Universidad de Alcalá de Henares.

Por tanto, al ser todo gratuito no hay ningún coste asociado.

7.2. Coste de Hardware

El material empleado para el desarrollo de este Trabajo Fin de Grado es el que se muestra en la Tabla 2. El cable micro USB se utiliza para conectar el Sensor Tile al ordenador y así poder transmitir las predicciones que hace. El kit de desarrollo Núcleo L476RG se utiliza para programar el Sensor Tile y el cable mini USB se utiliza para conectar este kit al PC. Por último, la tarjeta Micro SD se emplea para realizar la grabación de los audios con el Sensor Tile y que sean utilizadas en el script de Python.

A continuación, se muestra el desglose del coste de material:

Descripción	Precio	Cantidad	Coste total
Ordenador Portátil HP Pavilion Notebook 15-bc517ns	800 €	1	800 €
Kit de desarrollo STEVAL-STLKT01V1 (Sensor Tile)	94 €	1	94 €
Cable micro usb	2,50 €	1	2,50 €
Kit de desarrollo Nucleo L476RG	13,41 €	1	13,41 €
Cable mini USB para	3,25 €	1	3,25 €
Tarjeta Micro SD 32Gb	10,90 €	1	10,90 €
Coste total del material			**924,06 €**

Tabla 1. Coste del material

7.3. Coste de mano de obra

Concepto	Tiempo (horas)	Coste hora	Coste total
Planificación	8	50 €	400 €
Desarrollo	200	50 €	10.000 €
Pruebas	32	50 €	1.600 €
Coste total ejecución			**12.000 €**

Tabla 2. Coste de mano de obra

7.4. Coste de ejecución material

Concepto	Coste Total
Coste de material	924,06 €
Coste de mano de obra	12.000 €
Total	**12.924,06 €**

Tabla 3. Coste de ejecución material

7.5. Presupuesto de ejecución por contrata

Concepto	Coste Total
Ejecución material	12.924,06 €
Gastos generales (5%)	646,20 €
Beneficio industrial (50%)	6.462,03 €
Total	**20.032,29 €**

Tabla 4. Presupuesto de ejecución por contrata

7.6. Presupuesto Total

Concepto	Coste Total
Ejecución por contrata	20.032,29 €
IVA (21%)	4.206,78 €
Total	**24.239,07 €**

Tabla 5. Presupuesto Total

8. CONCLUSIONES Y LÍNEAS FUTURAS

A lo largo de este Trabajo Fin de Grado se ha conseguido demostrar la implementación de una red neuronal diseñada en Python en un microcontrolador STM32.

En todo el proceso es muy importante la correcta instalación de todos los programas, así como trabajar con las mismas versiones durante todo el diseño. Un cambio de versión en Python, Keras, o en la librería X-CUBE-AI puede provocar fallos en las compilaciones del código.

Durante el diseño y pruebas se comprobó que es muy importante el utilizar un *dataset* grabado con la misma configuración en los periféricos encargados de procesar el audio tanto en la configuración de grabación como en la de procesar el audio que se presenta a la red para realizar la clasificación. Esto es así, porque la red neuronal se entrena con unas características de audio específicas de frecuencia y resolución, así como de ruido de fondo. Si la configuración de los periféricos fuera distinta a la configuración realizada en la grabación del audio, y si las condiciones del entorno son diferentes a como se hizo la grabación, los resultados de predicción serían mucho peores.

Otro aspecto que hay que tener muy en cuenta es el tamaño de la red. Si se aumenta el número de capas de la red, se aumenta el espacio que ocupa la red en la memoria del microcontrolador. Por ello, a través de la comprobación de Validate on Target se puede ver si la red neuronal entra dentro del espacio de memoria del microcontrolador.

También hay que tener cuidado con el tiempo que tarda la red en realizar una inferencia, es decir, en procesar el audio. Puesto que la frecuencia de muestreo era de 16 KHz y el Fill_Buffer es de 1024 muestras, el tiempo máximo de procesamiento es de 64 ms. Con un *timer* se midió el tiempo desde que se lanza la red hasta que arroja un resultado y estaba por debajo de esos 64 ms. Si esto no es así, las predicciones serían imprecisas debido al solapamiento de datos de audio en el *buffer*.

Una vez visto y documentado todo el proceso de la integración de la red neuronal en el microcontrolador se facilita bastante el abordar un problema de este tipo ya que se han documentado todos los procesos de configuración y validación del código.

El hacer un diseño de esta manera es bastante ventajoso ya que Python es un lenguaje de programación muy extendido debido a su alto número de bibliotecas especializadas en *Deep Learning* como Keras. Además, Python se caracteriza por tener una gran comunidad de usuarios que lo respalda, tiene una sintaxis legible y entendible que proporciona reducir el tiempo de desarrollo de un diseño. A todo esto hay que añadir las ventajas que aporta el implementar una red neuronal en un microcontrolador como son:

- El procesamiento directo, lo que hace que se puedan clasificar los diferentes estados en tiempo real y si fuera necesario tomar las decisiones que se considerasen oportunas. Esto hace que se reduzca enormemente la latencia proporcionando una gran ventaja al sistema.
- El bajo consumo de energía, ya que los microcontroladores de STM32 están optimizados para consumir muy poco.
- El coste reducido del Sensor Tile ya que no es necesario invertir grandes cantidades de dinero en procesadores tipo FPGA que, aunque aumentan las capacidades, también aumentan el coste y el tiempo de desarrollo e implementación. Además, si la red neuronal no es excesivamente compleja con un microcontrolador es suficiente ya que entra dentro de la memoria del microcontrolador.
- Capacidad para trabajar en condiciones industriales, lo que hace que aporten un grado de fiabilidad al sistema importante.
- La facilidad para actualizar y modificar el *firmware* es otra de las ventajas al utilizar este tipo de microcontroladores ya que se pueden dotar de nuevas mejoras y características adicionales sin cambiar el hardware.

Como líneas futuras de mejora se podrían hacer los siguientes trabajos:

- Transmitir las predicciones al PC a través de bluetooth y mediante una aplicación se muestre el estado de las predicciones. En función del estado del ventilador se puede activar un sensor o un relé que pare el ventilador, aumente la velocidad o lo ponga en marcha.
- Hacer más robusta la red neuronal, para ello, se puede hacer un trabajo un tratamiento digital de la señal de audio con una o varias etapas de filtrado para eliminar el ruido de fondo.

Con esta base en proyectos de clasificación de señales de audio, puede ser utilizado para desarrollar proyectos del siguiente tipo:

- Mantenimiento de equipos para comprobar el funcionamiento correcto de los mismos dependiendo del sonido que emitan.
- Conocer el estado en el que se encuentra un bebé, durmiendo, despierto o llorando.
- Controlar un bluetooth de un vehículo en función de comandos por voz enviados.

Hay que recalcar que con el conjunto de sensores que incorpora el Sensor Tile se pueden hacer multitud de proyectos. El Sensor Tile incorporaba un micrófono digital, un sensor inercial con acelerómetro y giróscopo, un eCompass, un magnetómetro, un sensor de presión y la capacidad de comunicación por Bluetooth entre otras características.

Por ello se podrían hacer diseños para:

- Control de movimiento sobre personas que necesiten prótesis.
- Control de movimientos inusuales y activación de alarmas.
- Sistemas de navegación para determinar su posición, orientación y velocidad como puede ser un dron.
- Orientación y navegación para senderistas.
- Estabilización de cámaras y su empleo en drones.
- Monitoreo deportivo, contador de pasos, medir distancia recorrida, etc.

Cada vez el mundo está más sensorizado, los dispositivos IoT son cada vez más y todo tiende a estar conectado. Las redes neuronales están presentes en multitud de aplicaciones de uso diario, como los contadores de pasos, reconocimiento de objetos, equipos de mantenimiento, cadenas de montaje, en vehículos…

Por ello, hay que destacar la importancia de las redes neuronales y su amplio rango de utilización. El poder programar una red neuronal en un lenguaje comúnmente usado como Python y poder instanciarlo en un microcontrolador proporciona grandes ventajas para el diseñador, acortando tanto la fase de diseño y pruebas, como la consecución de un producto final.

En definitiva, el diseño de redes neuronales y su implementación en microcontroladores proporciona un gran número de soluciones a situaciones y problemas que existen en nuestro día a día. Con este Trabajo Fin de Grado se pretendía demostrar este tipo de instanciación y que pueda servir como apoyo para alguien que se inicie en este campo.

9. BIBLIOGRAFÍA

Bajo estas líneas se listan los libros consultados para la elaboración del Trabajo Fin de Grado.

- Introducción a la programación en Python. Autores: Andrés Marzal e Isabel García.

ippython.pdf (usc.es)

- Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow. Concepts Tools and Techniques to Build Intelligent Systems. Autor: Aurélien Géron.

Hands-On Machine Learning with Scikit-Learn and TensorFlow (powerunit-ju.com)

- Deep Learning with Python. Autor: François Chollet.

François Chollet - Deep Learning with Python (2018, Manning).pdf (google.com)

- Aprenda a Pensar Como un Programador con Python. Autores: Allen Downey, Jeffrey Elkner, Chris Meyers.

top.dvi (argentinaenpython.com)

- ARM Cortex-M práctico. 1 – Introducción a los microcontroladores STM32 de ST. Autor: Àngel Perles.

libro-ARM-Cortex-M.pdf (upv.es)

- Programming with STM32 Getting Started with the Nucleo Board and C/C++. Autor: Donald Norris

Programming with STM32: Getting Started with the Nucleo Board and C/C++ (electrovolt.ir)

- Aprende Python. Autor: Sergio Delgado Quinteto

Aprende Python

A continuación se listan los enlaces de las páginas webs consultadas. Se dividirá en las diferentes partes del Trabajo Fin de Grado.

- Links empleados para el Marco Teórico:

Redes neuronales: entrenamiento y comportamiento (ucm.es)

Redes neuronales desde cero (I) - Introducción - IArtificial.net

Redes neuronales desde cero (II): algo de matemáticas - IArtificial.net

Error Cuadrático Medio para Regresión - IArtificial.net

Algoritmo backpropagation: funciones y aplicaciones | UNIR

Propagación hacia atrás - Wikipedia, la enciclopedia libre

What is Gradient Descent? | IBM

Overfitting. Qué es, causas, consecuencias y cómo solucionarlo | Grupo Atico34 (protecciondatos-lopd.com)

¿Qué es Underfitting y Overfitting? | by Rubiales Alberto | Medium

¿Qué es el subajuste? | IBM

- Links empleados para el Hardware del Sistema:

Getting started with the STEVAL-STLKT01V1 SensorTile integrated development platform - User manual

Interfacing PDM digital microphones using STM32 MCUs and MPUs - Application note

MP34DT05-A - MEMS audio sensor omnidirectional stereo digital microphone - STMicroelectronics

STM32L476JG - Ultra-low-power with FPU Arm Cortex-M4 MCU 80 MHz with 1 Mbyte of Flash memory, LCD, USB OTG, DFSDM - STMicroelectronics

DFSDM internal peripheral - stm32mpu

STM32L4-System-Digital FIlter for SD Modulators interface (DFSDM) (st-onlinetraining.s3.amazonaws.com)

STM32 Microphone Audio Acquisition: Part 5, STM32 Peripherals for MEMS digital microphones - YouTube

Interfacing PDM digital microphones using STM32 MCUs and MPUs - Application note

STM32L4_System_DFSDM.pdf

- Links empleados para el Diseño de la Red Neuronal:

¿Qué es Keras? Introducción a la biblioteca de redes neuronales - IONOS

¿Qué es TensorFlow? ¿Cómo funciona? - Aprende IA

Descubre Jupyter Notebook: imprescindible para ciencia de datos (aprenderbigdata.com)

Tipos de capas de red neuronal convolucional (keepcoding.io)

Redes Neuronales Convolucionales, la Guía Definitiva - Frogames

Convolutional Neural Network : definición y funcionamiento (datascientest.com)

Convolutional Neural Networks: La Teoría explicada en Español | Aprende Machine Learning

Función SoftMax: Activación para la clasificación - Jacar

How To Calculate the Mean and Standard Deviation — Normalizing Datasets in Pytorch | by Jorrit Willaert | Towards Data Science

Matriz de confusión - Wikipedia, la enciclopedia libre

How does ReLU activation work?. How ReLU works is a mysterious thing to… | by nipun deelaka | Analytics Vidhya | Medium

Getting started with X-CUBE-AI Expansion Package for Artificial Intelligence (AI) - User manual

AI_WS: Workshop AI - Google Drive

- Links empleados para Configuración y Validación:

Getting started with X-CUBE-AI Expansion Package for Artificial Intelligence (AI) - User manual

Introduction to STM32Cube.AI - 4 NN Model creation using Keras - YouTube

- Links empleados para la obtención de figuras del Trabajo Fin de Grado:

[1] Artificial Intelligence, Machine Learning, and Deep Learning: Same context, Different concepts - Master Intelligence Economique et Stratégies Compétitives (master-iesc-angers.com)
[2] Redes neuronales desde cero (I) - Introducción - IArtificial.net
[3] content (ucm.es)
[4] SoftMax Activation Function: Everything You Need To Know (insideaiml.com)
[5] Convolutional Neural Networks: La Teoría explicada en Español | Aprende Machine Learning
[6] Error Cuadrático Medio para Regresión - IArtificial.net
[7] Redes neuronales desde cero (II): algo de matemáticas - IArtificial.net
[8] Gradiente Descendiente para aprendizaje automático - IArtificial.net
[9] ¿Qué es Underfitting y Overfitting? | by Rubiales Alberto | Medium
[10] Understanding the Mel Spectrogram | by Leland Roberts | Analytics Vidhya | Medium
[11] Getting started with the STEVAL-STLKT01V1 SensorTile integrated development platform - User manual
[12] Interfacing PDM digital microphones using STM32 MCUs and MPUs - Application note
[13] Datasheet - MP34DT05-A - MEMS audio sensor omnidirectional digital microphone (st.com)
[14] STM32L47xxx, STM32L48xxx, STM32L49xxx and STM32L4Axxx advanced Arm®-based 32-bit MCUs - Reference manual
[15] Captura de pantalla de: STM32 Microphone Audio Acquisition: Part 5, STM32 Peripherals for MEMS digital microphones - YouTube
[16] Captura de pantalla de: STM32Cube.AI workshop - 4 Five steps behind Neural Network (NN) - YouTube
[17] Captura de pantalla de: STM32Cube.AI workshop - 6 Neural Network (NN) Model creation - YouTube
[18] How To Calculate the Mean and Standard Deviation — Normalizing Datasets in Pytorch | by Jorrit Willaert | Towards Data Science
[19] Redes Neuronales Convolucionales, la Guía Definitiva - Frogames
[20] Convolutional Neural Network : definición y funcionamiento (datascientest.com)
[21] How does ReLU activation work?. How ReLU works is a mysterious thing to… | by nipun deelaka | Analytics Vidhya | Medium
[22] Getting started with X-CUBE-AI Expansion Package for Artificial Intelligence (AI) - User manual

[23] STM32Cube.AI workshop - 9 STM32CubeMX and X-Cube-AI lab - YouTube

Por último, se listan los video tutoriales consultados:

STM32CubeMX basics: 01.1 Introduction - Session introduction - YouTube

STM32 USB training - 01 Introduction - YouTube

STM32L4 training: 01. Introduction - YouTube

STM32Cube.AI workshop - 1 Introduction to Artificial Intelligence (AI) - YouTube

STM32 Microphone Audio Acquisition: Part 1, Microphone Basics - YouTube