

MÁSTER UNIVERSITARIO EN CIENCIA DE DATOS



VNIVERSITAT
DE VALÈNCIA

TRABAJO DE FIN DE MÁSTER

**RECONOCIMIENTO DE LENGUAJE
APLICADO COMO *TINY ML* EN UNA PLACA
CORAL DEV DE GOOGLE**

AUTOR:
CARLOS MARTÍNEZ MINGO

TUTORES:
EMILIO SORIA OLIVAS
REGINO BARRANQUERO CARDEÑOSA

SEPTIEMBRE, 2022

Resumen

Ejecutar modelos complejos de deep learning en dispositivos pequeños, con bajas prestaciones y utilizando pocos recursos y potencia es lo que se conoce como Tiny ML. Este enfoque busca explotar los beneficios de la filosofía del Edge Computing, según la cual los datos no deben salir del dispositivo en el que se generan. Las áreas de aplicación del Tiny ML son muy amplias. En este trabajo se ha estudiado cómo definir, entrenar y preparar diversos modelos de reconocimiento de lenguaje en forma de detección y clasificación de palabras de activación, y cómo adaptarlos cumpliendo los requisitos impuestos por el dispositivo de Tiny ML (en este caso la placa Coral Dev de Google) para que se pueda hacer uso de toda su potencia, pudiendo ejecutar versiones cuantizadas de los modelos en la TPU. Por un lado, utilizando el conjunto de datos conocido como *speech commands*, se han generado y entrenado desde cero diversos modelos basados en redes neuronales. Concretamente, se han entrenado 3 modelos diferentes, cada uno de ellos respondiendo a ciertas características extraídas de los datos. El primer modelo es una red convolucional 2D que utiliza los datos transformados en espectrogramas de Mel, con el que se obtiene un accuracy sobre el conjunto de test del 88.50% en el modelo TFLite normal, y del 85.90% en el modelo cuantizado. El segundo modelo es otra red convolucional 2D diferente a la anterior, que utiliza los datos transformados en MFCCs, con el que se obtiene un accuracy en el conjunto de test del 87.81% con el modelo TFLite normal, y del 86.23% en su versión cuantizada. Por último, el tercer modelo es una red convolucional 1D que utiliza los datos crudos de los audios, obteniendo un accuracy en el conjunto de test del 87.16% en su versión TFLite normal, y del 80.75% en su versión cuantizada. Este último modelo muestra la influencia que puede tener el proceso de cuantización que se lleva a cabo en los modelos, ya que el no cumplir con las limitaciones del dispositivo de Tiny ML dificulta y empeora los resultados. Por otro lado, utilizando un conjunto de datos generado por mí, en español y mucho más reducido, se reentrenó un modelo neuronal complejo aplicando transfer learning con el objetivo de ver si el modelo era capaz de aprender las palabras de activación en un lenguaje diferente al del entrenamiento inicial. En este caso se han obtenido accuracies en el conjunto de test del 93.49% en su versión TFLite normal, y del 90.34% en su versión cuantizada. Con los modelos entrenados y adaptados para su ejecución en la placa, se ha verificado que la ejecución en la TPU es por lo general mucho más rápida, siempre y cuando los modelos se puedan ejecutar en ella.

Palabras clave: **Reconocimiento de lenguaje, Tiny ML, Redes neuronales, Transfer learning, TFLite, Coral Dev Board, Edge TPU**

Abstract

Running complex deep learning models in small devices, with low specifications and using few resources and power is known as Tiny ML. This approach seeks to make the most of the benefits of what is known as Edge Computing, a philosophy in which data should not leave the device where it has been generated. The areas where Tiny ML can be applied are huge. In this work we have studied how to define, train and prepare several speech recognition models, in the form of detection and classification of wake words, and how to adapt them to meet the requirements imposed by the Tiny ML device (in this case, the Coral Dev Board) to use all its power, being able to execute quantized versions of the models in the TPU. On the one hand, using the dataset known as *speech commands*, several models have been generated and trained from the beginning. Specifically, 3 different models have been trained, each one of them reacting to certain features extracted from the data. The first model is a 2D convolutional neural network that uses data transformed into Mel spectrograms, which has obtained an accuracy on the test set of 88.50% with the normal TFLite version, and 85.90% with the quantized version. The second model is a different 2D convolutional neural network that uses data transformed into the so-called MFCCs, which has obtained an accuracy over the test set of 87.81% with the normal TFLite model, and 86.23% with the quantized version. Finally, the third model is a 1D convolutional neural network that uses the raw audio data, obtaining an accuracy over the test set of 87.16% with the normal TFLite model, and 80.75% with the quantized version. This last model shows the influence that the quantization process can have on the models, since not complying with the limitations of the Tiny ML device difficults and worsens the results. On the other hand, using a dataset generated by me, in Spanish and much smaller, a complex neural network was retrained with transfer learning in order to verify if the model was able to learn the wake words in a language different from the initial training language. In this case, the accuracies over the test set are 93.49% with the normal TFLite version, and 90.34% with the quantized version. With the models trained and adapted for executions on the board, it has been verified that the execution in the TPU is in general much faster, as long as the models can be run on it.

Keywords: **Speech recognition, Tiny ML, Neural networks, Transfer learning, TFLite, Coral Dev Board, Edge TPU**

Gracias a todos los que me habéis dedicado unos minutos para ayudarme con la recolección de los datos para enriquecer el contenido y los resultados de este trabajo.

Índice general

Motivación y planteamiento general	1
1. Introducción y marco teórico	4
1.1. Bases del tratamiento y procesamiento de audio	5
1.1.1. Conceptos importantes aplicados en el desarrollo	6
1.2. Reconocimiento de lenguaje o <i>speech recognition</i>	12
1.2.1. Evolución histórica y estado del arte	12
1.2.2. Detección de palabras de activación	14
1.3. Tiny ML	15
1.3.1. ¿Qué es el “Tiny ML”?	15
1.3.2. TPUs para potenciar el Tiny ML	17
2. Herramientas, datos y modelos	18
2.1. Placa Coral Dev, de Google	19
2.1.1. Especificaciones	20
2.1.2. Compatibilidad y requisitos para el correcto funcionamiento . . .	21
2.1.3. Preparación de la placa	25
2.2. Conjuntos de datos y tratamiento	28
2.2.1. Conjunto de datos para el entrenamiento de varias redes neuronales sencillas	29
2.2.2. Conjunto de datos para aplicar transfer learning sobre una red neuronal compleja	40
2.3. Modelos	43
2.3.1. Redes neuronales sencillas	43
2.3.2. Red neuronal compleja sobre la que aplicar transfer learning . . .	48

3. Resultados	50
3.1. Modelos sencillos	50
3.1.1. CNN utilizando los espectrogramas de Mel	51
3.1.2. CNN utilizando los MFCCs	54
3.1.3. CNN utilizando los datos crudos de las señales	57
3.1.4. Comparación entre los diferentes modelos sencillos	60
3.2. Transfer learning en una red compleja	61
3.3. Ejecución de los modelos en la placa Coral Dev	63
4. Conclusiones	65
4.1. Trabajo futuro: mejoras y ampliaciones	66
Bibliografia	69
Anexos	71
A1. Ejemplos de la forma de onda y espectrograma de Mel de varios audios, agrupados por etiqueta	72
A2. Gráficos de las estructura de los modelos utilizados	78
A3. Código para la ejecución de modelos TFLite en la placa Coral	82

Motivación y planteamiento general

“Alexa, enciende las luces del salón.”

“OK Google, dime la mejor ruta para llegar a casa desde mi ubicación.”

Estas dos frases que hoy en día nos parecen tan normales y comunes, hace poco más de 1 década solo se habrían entendido si estuviésemos hablando de una película de ciencia ficción de carácter futurista. Pero parece que ese futuro ya ha llegado porque muchas de estas cosas que considerábamos tan lejanas forman parte de nuestra realidad y de nuestro día a día, de una manera tan rutinaria que casi parece natural. Y esto es así, sin ningún tipo de duda, gracias al increíble y rápido avance que están experimentando campos tan novedosos como la ciencia de datos, el aprendizaje máquina o *machine learning* y la inteligencia artificial o IA, entre otros. Avance que está siendo propiciado por las mejoras tecnológicas y técnicas que han ido apareciendo año tras año, y que han hecho (y están haciendo) posible que modelos con muchos millones de parámetros sean entrenados y mejorados constantemente, ofreciéndonos utilidades y aplicaciones muy diversas que en muchas ocasiones nos hacen la vida un poquito más sencilla. Aplicaciones que van desde el análisis y la extracción de información y conocimiento latente de datos masivos por métodos supervisados, no supervisados o híbridos, con modelos de clasificación o de regresión, hasta el procesamiento de lenguaje natural o la visión artificial por ordenador utilizando modelos de aprendizaje profundo o *deep learning*, basados en redes neuronales.

Así, aunque nos parezca *tan simple* el hecho de que un aparato inteligente nos reproduzca la canción que le hemos pedido, todo lo que está ocurriendo por detrás es altamente complejo y refinado [1]. De manera muy resumida, cada vez que Alexa se activa están sucediendo varios procesos por detrás. Primero, siempre que Alexa tiene el micrófono encendido está procesando todo el audio que recibe esperando la palabra de activación o *wake word* que desencadene el resto de acciones. Cuando la palabra de activación se detecta, se llama a un servicio cloud de Amazon que procesa el resto del audio, aplicando modelos muy precisos de reconocimiento y procesamiento de lenguaje natural para entender qué es lo que se le está pidiendo a Alexa. Cuando el modelo devuelve los resultados se desencadenan las acciones programadas como consecuencia de la respuesta recibida por el modelo y Alexa las lleva a cabo, por ejemplo encendiendo una luz o poniendo una canción. Y a todo este proceso general hay que añadirle la complicación extra de desarrollar y entrenar los modelos correctamente para que funcionen según lo esperado y sean capaces de generalizar su funcionamiento para la gran variedad que puede haber en los datos de entrada, de modo que, por ejemplo, puedan procesar el audio en un entorno ruidoso.

Pero en todo este proceso de reconocimiento de lenguaje que realiza Alexa y que hemos introducido muy brevemente, hay un aspecto que fácilmente se puede pasar por alto: la detección de la palabra de activación se realiza directamente en el dispositivo sin que el audio se envíe a un servicio en la nube para su procesado. Esto es así porque la respuesta que debe dar Alexa debe ser rápida y la latencia debe ser mínima para que el resto del comando que se indique sea procesado en su totalidad, sin que se pierda información relevante. Esta filosofía en la que el procesado del dato se lleva a cabo en el propio dispositivo sin tener que enviarlo a otro lado es lo que se conoce como *Edge Computing*, y es uno de los principales enfoques que se está explotando con el auge del IoT y del conocido como *Tiny ML* que introduciremos posteriormente. Esta filosofía presenta grandes ventajas pero para poder llevarla a cabo en dispositivos de tamaño reducido hay que desarrollar los modelos de una forma determinada, de modo que sean compatibles con las restricciones que imponen los dispositivos.

Es así como surge la idea de llevar a cabo este pequeño trabajo de introducción al Tiny ML. El objetivo final es el desarrollo de varios modelos de reconocimiento de lenguaje, particularizando en reconocimiento de palabras de activación (análogo a lo que hemos comentado sobre Alexa), que puedan ejecutarse en un dispositivo de Tiny ML, en nuestro caso la placa Coral Dev de Google. Para ello, en una primera parte se definen y entrena varías redes neuronales utilizando un conjunto de datos de audios con diversas palabras de activación en inglés. Se entrena varías modelos diferentes porque, aunque los datos brutos son los mismos, se han probado diferentes estructuras y formas de introducir los datos extrayendo de ellos determinada información, con el objetivo de estudiar y analizar cómo de relevante puede ser procesar previamente los datos para mejorar el reconocimiento de lenguaje o adaptarse a diferentes circunstancias en las que, por ejemplo, el poder de computación o la disponibilidad de algoritmos de preprocesado sean limitados. Por otro lado, para enriquecer el trabajo, se ha llevado a cabo una segunda parte en la que se utiliza una red compleja previamente entrenada con un conjunto de datos similar al utilizado en la primera parte, sobre la que se aplica transfer learning con un conjunto de datos reducido y que contiene las mismas frases de activación, pero en este caso en el idioma español. Con ello comprobamos si es posible reentrenar un modelo para que lleve a cabo una tarea similar a la original pero en un idioma diferente, además de verificar si se pueden ejecutar modelos más complejos en dispositivos de Tiny ML de una manera rápida y eficiente, haciendo uso de toda su potencia.

Esta memoria es un resumen del trabajo llevado a cabo para conseguir estos objetivos. En ella, primeramente hacemos una introducción teórica de los conceptos principales que se van a tratar, yendo desde el marco teórico que envuelve el procesamiento de audio y los conceptos y características que se han extraído de los datos para entrenar los diferentes modelos, pasando por los conceptos relacionados con el reconocimiento del lenguaje y sus diferentes ramas, y finalizando con los conceptos relacionados con el Tiny ML y las características que presentan estos dispositivos para que sean tan relevantes en el mundo del machine learning y la inteligencia artificial. Posteriormente se pasa a una descripción más técnica de las herramientas, datos y modelos. Primero se introduce la placa Coral Dev, indicando qué ofrece, qué características y limitaciones presenta, y qué procedimiento se ha seguido para ponerla en funcionamiento desde 0. A continuación se describe todo lo relacionado con los datos utilizados en cada

una de las partes del trabajo, explicando el tratamiento que se ha hecho de ellos, el análisis y su final transformación en información relevante. Y en tercer lugar, se describen los modelos que se han definido y utilizado en cada una de las partes, junto con los procedimientos que se han tenido que llevar a cabo sobre ellos para transformarlos en modelos que se puedan ejecutar en la placa Coral Dev haciendo uso de las máximas prestaciones que ofrece, es decir, modelos que puedan ser ejecutados en la TPU de la placa. Seguidamente, se presentan los resultados obtenidos para cada una de las dos partes del trabajo, mostrando la bondad de los modelos entrenados y comparándolos con sus análogos modificados, con el objetivo de analizar si el proceso requerido para que puedan ser ejecutados en dispositivos de Tiny ML supone una disminución significativa de los rendimientos de los modelos. A todo esto se le añaden los resultados obtenidos de llevar a cabo las ejecuciones en la placa Coral Dev, donde se analiza la velocidad de ejecución que tienen unos modelos u otros, dependiendo de lo adaptados que estén o no para ser ejecutados en el módulo TPU. Por último, y para terminar el trabajo, se presentan las conclusiones obtenidas de manera sintetizada, y también se plantean diferentes pautas de actuación y futuros trabajos y desarrollos que podrían llevarse a cabo para completar o extender lo planteado en esta primera iteración que se ha llevado a cabo para introducirse en el mundo del Tiny ML, de la mano de la placa Coral Dev de Google.

Capítulo 1

Introducción y marco teórico

El reconocimiento de lenguaje es uno de los campos que podríamos considerar *moderno* de la inteligencia artificial, junto con otros como la segmentación automática, los relacionados con vídeo o todos aquellos sobre los que se tengan que aplicar necesariamente redes neuronales, profundas o no, para obtener buenos resultados. El enfoque de las redes neuronales permite tratar datos complejos sobre los que se debe tener en cuenta la evolución temporal intrínseca a ellos, y por lo que en determinados casos puede ser complicado aplicar sobre ellos modelos *clásicos*. Por establecer una comparación que matice lo que se está queriendo decir, un problema clásico de inteligencia artificial utiliza datos organizados en columnas en los que cada una de ellas representa una característica que puede ser más o menos relevante para el conjunto final, mientras que los enfoques modernos buscan que sea la propia red neuronal la que extraiga la mejor información de un conjunto de datos más complejo, que no necesariamente responde a formatos de datos columnares y resumidos.

Por este motivo es necesario tener un conocimiento más profundo de estos datos complejos, para poder saber (o por lo menos tener una idea fundamentada) cómo proceder y cómo enfocar el problema a resolver. Por ello, en este primer capítulo se van a introducir las bases del tratamiento y procesamiento de los datos en formato de audio, que son los que vamos a utilizar a lo largo del trabajo. Partiendo de una base física, entenderemos cómo interpretar los datos, cuáles son los conceptos básicos que es necesario conocer al llevar el sonido a formato digital y, de una manera más específica, cuáles son los conceptos y “características” que vamos a utilizar en el desarrollo.

Posteriormente se pasará a definir de manera más concreta qué entendemos por reconocimiento de lenguaje, en el mundo del aprendizaje máquina y la inteligencia artificial. Se explicará de manera resumida cómo ha sido la evolución histórica del reconocimiento del lenguaje, cuál es el estado del arte y comentaremos la rama particular de detección de palabras, una de las diversas ramas que hay, y de la que hemos realizado este trabajo.

Por último explicaremos también qué entendemos realmente por Tiny ML, indicando tanto sus características como las ventajas e inconvenientes que aparecen con esta nueva filosofía, detallando también el papel fundamental que juegan las TPUs en este campo tan novedoso.

1.1. Bases del tratamiento y procesamiento de audio

Para entender bien los datos de los audios tenemos que entender qué representan en el mundo físico, que no es otra cosa que sonido. El sonido se puede entender como una manifestación auditiva de cualquier fenómeno que involucre la propagación de ondas mecánicas (ondas que necesitan un medio material para propagarse) generadas por el movimiento vibratorio de un cuerpo. Estas vibraciones provocan que las moléculas del aire oscilen, y el cambio en la presión que generan las oscilaciones crea la onda. Por ello, cuando visualizamos datos de tipo audio, vemos la forma de onda que producen las vibraciones que lo originaron.

Las ondas vienen caracterizadas por su frecuencia, su amplitud y su fase. La frecuencia se puede relacionar con la energía y la amplitud con la intensidad. Además, en el caso del sonido aparece otra variable, el timbre, que le aporta “color”, haciendo que cada uno sea diferente de los demás. Esta es una variable difícil de definir, que en ocasiones se explica como la diferencia entre dos sonidos que tienen la misma intensidad, frecuencia y duración, y con características propias como el envolvente acústico, el contenido armónico y las modulaciones en amplitud y frecuencia.

De manera general, el sonido no está formado por una onda periódica concreta, sino que es la suma de muchas ondas diferentes, no periódicas, de manera continua (teniendo en cuenta que las ondas no periódicas son una agregación de ondas periódicas). Este último detalle es importante: en el mundo físico las ondas son analógicas, continuas, pero esa continuidad no se puede llevar de manera totalmente fiel al mundo digital. Es por esto que surge un problema a la hora de intentar llevar a cabo esta conversión, pues se necesitan transformar valores continuos del tiempo y de la amplitud en sus correspondientes valores discretos, finitos. Este proceso de conversión realizado por un ADC (convertidor analógico-digital) se basa en el muestreo y la cuantización de la señal analógica. El muestreo se puede entender como la selección de determinados puntos de la señal analógica continua que se cuantifican de manera discreta y se convierten a formato digital. Dependiendo tanto de la frecuencia de muestreo como de la precisión en la discretización se obtendrán representaciones digitales más o menos fieles de la onda analógica.

Después del proceso de digitalización la onda resultante vendrá caracterizada por una serie de magnitudes:

- frecuencia de muestreo o *sample rate*: número de muestras que se toman por segundo para llevar a cabo la digitalización de la onda analógica. Para entender la importancia final que tiene esta magnitud podemos utilizar el teorema de Nyquist. Resumido de forma sencilla, este teorema establece que solo se podrán reconstruir correctamente frecuencias iguales o inferiores a la mitad de la frecuencia de muestreo. Frecuencias superiores a este valor sufrirán lo que se conoce como *aliasing*, una reconstrucción errónea de dichas frecuencias transformándolas en otras con un valor inferior. Esto supone un grave problema que hay que tener en cuenta a la hora de recopilar datos para resolver problemas basados en audios, ya podría darse el caso de que la información determinante para resolverlo esté en

frecuencias que no se han reconstruido bien, por intentar reducir el volumen de los datos. Por ello, la calidad del audio y las características del mismo son esenciales, motivo por el cual los audios se suelen guardar en formato .wav, un formato de alta calidad y sin compresión

- profundidad de bits o *bit size*: similar a la frecuencia de muestreo, pero en lugar de en el tiempo en las amplitudes. Una profundidad de bits mayor implica un mayor número de divisiones en las que se puede cuantizar la muestra, afectando directamente a la calidad del audio y al rango dinámico, obteniendo finalmente menos ruido
- número de frames: número total de muestras de las que se han guardado los valores. Estos valores representan las amplitudes digitalizadas de la onda analógica en cada punto temporal muestreado. La combinación del número total de frames y el número de frames por segundo nos permite determinar la duración del audio
- número de canales: cantidad de ondas de audio independientes grabadas en el mismo archivo. Se pueden tener 1 (mono) o 2 (estéreo) canales. En algunos problemas puede ser relevante tener 2 canales para que se guarde información relacionada o para tener una mejor reproducción final del audio

1.1.1. Conceptos importantes aplicados en el desarrollo

Teniendo claros los conceptos principales del sonido digital, pasamos a explicar diversos conceptos y otras características del audio en el contexto técnico del aprendizaje máquina y la inteligencia artificial. En el enfoque clásico del machine learning uno de los procedimientos centrales es la extracción de las características más adecuadas para el problema que se quiere resolver. En problemas relacionados con audio sucede lo mismo, ya que hay diferentes características que capturan determinados aspectos del sonido, lo que ayuda a generar mejores modelos inteligentes. Estas características se pueden categorizar de muchas formas:

- según el nivel de abstracción: se distinguen características de bajo (energía, centroide espectral, flujo espectral, ZCR...), medio (tono y tempo, MFCCs...) y alto (melodías, instrumentos, ritmos, género...) nivel
- según el alcance temporal: las características pueden ser instantáneas (del orden de varios milisegundos), segmentadas (del orden de segundos) o globales
- según aspectos musicales: es un caso más concreto, relacionado con los niveles de abstracción altos, donde las características se obtienen en función del timbre, el tono, la armonía, el ritmo...
- según el dominio de la señal: las características se pueden extraer del dominio temporal, del dominio frecuencial o de un dominio que combina la información temporal y frecuencial

Sin embargo, las características que se pueden extraer y utilizar dependen mucho de los datos que estén disponibles, ya que en ocasiones se puede requerir una labor humana para generarlas. Dicho tipo de características son las que se utilizarían siguiendo el enfoque clásico del machine learning. En nuestro caso vamos a seguir un enfoque moderno ya que vamos a utilizar redes neuronales que van a buscar las mejores características con los datos utilizados en su entrenamiento. Pero esto no quiere decir que no podamos llevar a cabo un preprocesado previo de los datos para transformarlos en información de determinado tipo, de modo que ciertas características puedan ser aprovechadas. Y esto es justo lo que vamos a hacer, extrayendo de los datos el espectrograma de Mel y los conocidos como coeficientes cepstrales en las frecuencias de Mel o MFCCs, que proporcionan muy buenos resultados en el reconocimiento de lenguaje [2].

Se han elegido estas dos características concretas por 2 motivos. En primer lugar, la transformación que hacen de los datos se puede interpretar como una imagen, por lo que se pueden entrenar modelos convolucionales con ellos. Y en segundo lugar, porque se basan en la escala de Mel [3], una escala logarítmica de la frecuencia que consigue que sonidos separados una cierta distancia en dicha escala se perciban igualmente separados por los humanos. Esto es lo que hace que la escala de Mel sea fundamental en modelos de machine learning, ya que busca inculcar en ellos nuestra percepción del sonido.

Los humanos tenemos un espectro auditivo que se extiende más o menos desde los 20 Hz a los 20 kHz, pero la percepción que tenemos de las frecuencias de dicho espectro es muy particular, pues es logarítmica: podemos percibir de manera más sencilla la diferencia entre un sonido de 100 Hz y otro de 200 Hz, pero es más complicado percibirla entre uno de 1100 Hz y otro de 1200 Hz. Aunque la distancia en Hz es la misma, nuestra percepción de dicha distancia no lo es. Es por este motivo que en muchas ocasiones en lugar de trabajar directamente con las frecuencias trabajamos con los tonos (*pitch* en inglés). Mostramos en la Figura (1.1a) la relación exponencial entre el tono y la frecuencia.

Algo similar sucede con la intensidad de los sonidos, medida en decibelios (dB), otra escala logarítmica que busca relacionar las intensidades según la percepción humana. La intensidad no debe confundirse con la percepción del ruido. El ruido es una percepción subjetiva de la intensidad que depende de la duración y frecuencia del sonido y de la edad de la persona, entre otros factores, y se mide en fonos. Mostramos en la Figura (1.1b) la relación logarítmica de la intensidad, donde también se muestran las curvas de igual ruido y cómo éste depende de la frecuencia del sonido.

La escala tonal sigue una relación logarítmica con la frecuencia pero no está totalmente linealizada con respecto a la percepción humana, ya que se basa en octavas formadas por tonos y semitonos. Avanzando un poco más, la escala de Mel representa una escala que se adecúa a las percepciones de los tonos, lo que la hace mucho más adecuada para ser la base de características de los sonidos que busquen emular la percepción humana. Es por esto que utilizamos los espectrogramas de Mel y los MFCCs.

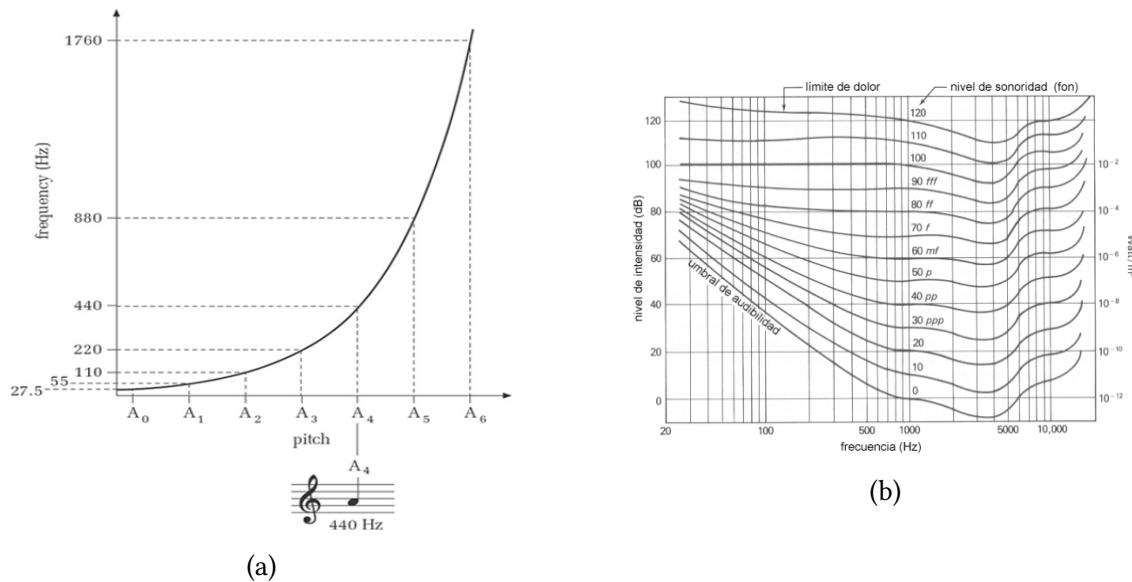


Figura 1.1. Relaciones logarítmicas de las percepciones del sonido. La Figura (1.1a) muestra la relación entre el tono (*pitch*) y la frecuencia, y la Figura (1.1b) la relación entre la intensidad y la frecuencia, mostrando también las curvas de igual ruido.
Imágenes descargadas de: [OpenLearn: the open university](#) (a) y [Blog Audioparamusicos](#) (b).

Espectrograma de Mel

Un espectrograma es la representación gráfica de las frecuencias excitadas en una señal. Se calcula aplicando la transformada de Fourier⁽¹⁾ sobre la señal, que la lleva desde el dominio temporal al dominio frecuencial. Sin embargo, aplicar directamente la transformada de Fourier sobre toda la señal puede ser contraproducente, ya que se pierde el conocimiento temporal de la información: se sabría qué frecuencias se han excitado, pero no cuándo lo han hecho. En su lugar, lo que se hace es aplicar la transformada de Fourier de tiempo reducido (*short-time Fourier transform*, STFT), una variación de la transformada de Fourier original que se aplica sucesivamente sobre la señal enventanada en diversos momentos. De este modo no se pierde el conocimiento temporal de los datos. Las ventanas que se calculen de la señal deben presentar cierto solapamiento entre ellas para evitar lo que se conoce como goteo espectral: una especie de efecto de borde en el inicio y final de la ventana, que se traduce en errores en la detección de las frecuencias excitadas. Graficando la intensidad de las frecuencias excitadas en cada uno de los segmentos temporales se obtiene el espectrograma.

Como hemos dicho que la percepción de la intensidad es logarítmica, podemos convertir el espectrograma original en un espectrograma con la intensidad en decibelios, de modo que la visualización sea más “entendible”, humanamente hablando. Y lo mismo podemos hacer con respecto a las frecuencias. Si las convertimos a la escala de Mel obtenemos un espectrograma de Mel en el que todas las magnitudes son perceptiblemente relevantes. Para realizar la conversión de frecuencias a la escala de Mel se deben seguir los siguientes pasos:

⁽¹⁾Como estamos hablando de señales digitales, cada vez que hablamos de la transformada de Fourier nos referimos a su versión discreta.

1. elegir el número de bandas de Mel. Para cada problema el número de bandas de Mel puede ser uno u otro, por lo que es posible que haya que iterar por prueba y error
2. generar los filtros de Mel. Se generan tantos filtros como bandas de Mel se hayan escogido. Para generar los filtros, primero se convierten las frecuencias inferior y superior a la escala de Mel por medio de la expresión de la Ecuación [1.1], posteriormente se escogen tantos puntos equiespaciados en dicho intervalo generado como bandas de Mel se hayan seleccionado, y por último se convierten dichos a Hz por medio de la Ecuación [1.2], redondeándolas al bin frecuencial más cercano. Con todo ello, se pueden construir los filtros triangulares tal y como se muestra en la Figura (1.2)

$$m = 2595 \cdot \log \left(1 + \frac{f}{500} \right) \quad (1.1)$$

$$f = 700(10^{m/2595} - 1) \quad (1.2)$$

donde m son los mels y f las frecuencias

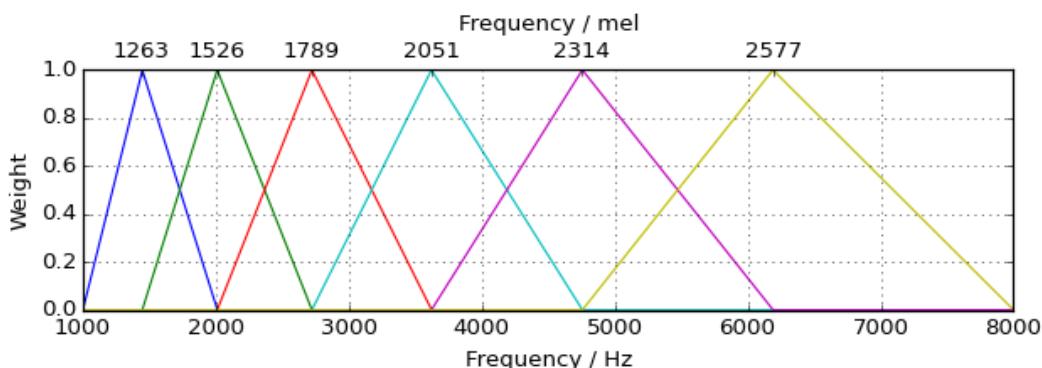


Figura 1.2. Filtros triangulares de Mel generados para 6 bandas de Mel.
Imagen descargada de: [PyFilterbank](#).

3. aplicar los filtros de Mel sobre el espectrograma

De esta manera, se obtendrá finalmente el espectrograma de Mel, cuyas dimensiones se corresponderán con el número de bandas de Mel escogidas y el número de segmentos temporales obtenidos tras aplicar la transformada de Fourier con un tamaño de ventana determinado.

MFCCs

Las otras características que vamos a extraer del conjunto de datos son los MFCCs [4], *Mel-Frequency Cepstral Coefficients*, en inglés. La palabra “cepstral” es un adjetivo que proviene de “cepstrum”. Si analizamos este sustantivo, vemos que es un anagrama de la palabra “spectrum”. Siguiendo este mismo proceso, a medida que entramos en el mundo de los *cepstrums* aparecen otras palabras similares como *quefrequency* (frequency),

liftering (filtering) y *rhamonic* (harmonic), que son los análogos de las frecuencias, los filtros y los armónicos. A nivel teórico, el cepstrum es un espacio al que se llega aplicando la expresión mostrada en la Ecuación [1.3].

$$C(x(t)) = \mathcal{F}(\log(\mathcal{F}(x(t)))) \quad (1.3)$$

donde \mathcal{F} es la transformada de Fourier, $x(t)$ es la señal temporal de audio, y $C(x(t))$ es la señal en el espacio cepstrum

Desglosando la ecuación, vemos que se llega al espacio cepstrum aplicando la transformada de Fourier sobre la señal de audio, aplicando posteriormente el logaritmo, y volviendo a aplicar la transformada de Fourier, como si el propio espectro fuese una señal. Por ello, el cepstrum puede entenderse como un *espectro del espectro*, cuya unidad es la ya mencionada quefrecency (una pseudo-frecuencia medida en milisegundos). En el cepstrum se muestran las quefrecencies excitadas en el espectro anterior, marcando así los rhamonics. Mostramos en la Figura (1.3) un ejemplo de cepstrum generado, donde se ve el primer rhamonic, que es la quefrecency asociada con la frecuencia fundamental de la señal original.

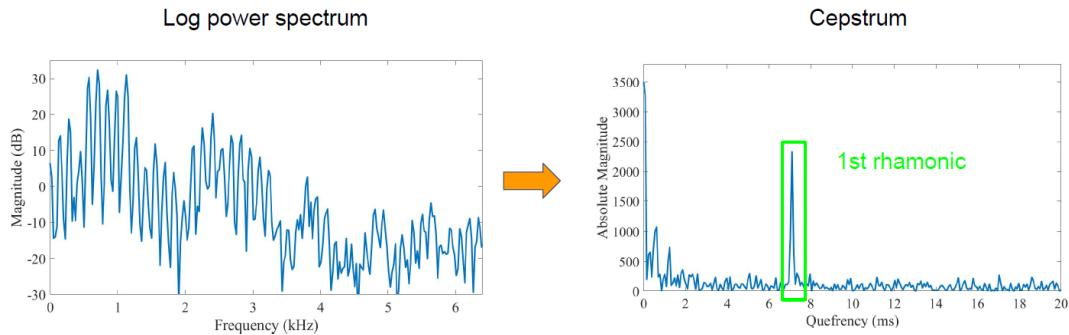


Figura 1.3. Ejemplo de cepstrum, espectro del espectro, cuyas unidades son las quefrecencies. Se muestra el primer rhamonic, o quefrecency asociada a la frecuencia fundamental de la señal. Imagen descargada de: [Github: AudioSignalProcessingForML](#).

Entendido qué es el cepstrum, pasamos a explicar por qué juega un papel esencial. Los sonidos del habla se generan cuando el tracto vocal altera los pulsos glotales de las cuerdas vocales. Si sobre el logaritmo del espectro se calcula el envolvente espectral (que no es más que un suavizado del propio espectro), se obtienen unos picos que se conocen como formantes o portadores de la identidad del sonido, y que contienen información muy profunda relacionada con el timbre o incluso los fonemas del lenguaje. Por ello, es interesante aislarlos y tratarlos como características del audio. Este envolvente espectral se puede entender como la frecuencia de respuesta del tracto vocal, generadora de los diferentes formantes. El ruido resultante de restar el envolvente espectral al logaritmo del espectro se puede relacionar con los pulsos glotales, por lo que el habla se puede entender como la convolución de la frecuencia de respuesta del tracto vocal con el pulso glotal. Mostramos en la Figura (1.4) un resumen de estos conceptos.

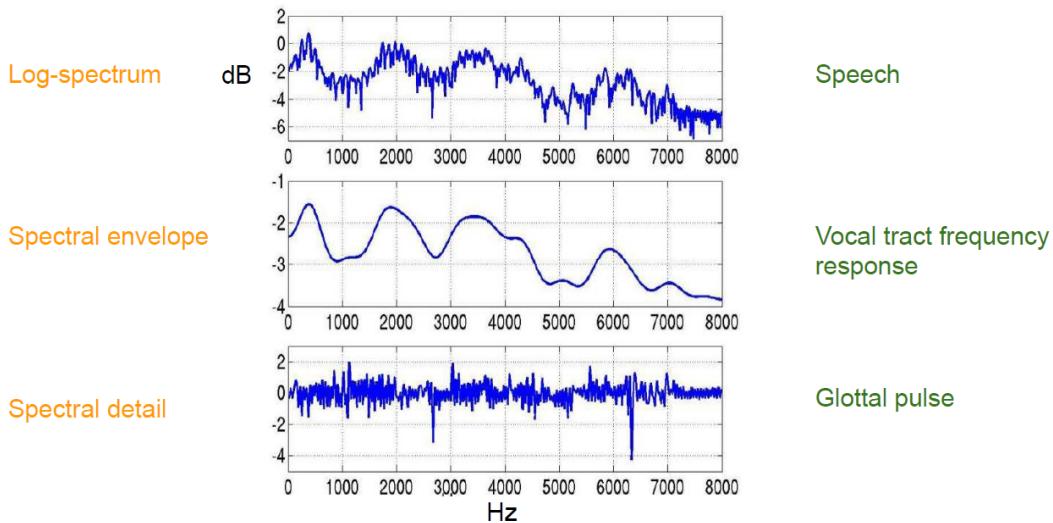


Figura 1.4. Resumen para entender la importancia del cepstrum, donde se ve la relación de los derivados obtenidos del logaritmo del espectro con la generación del habla en el tracto vocal.
Imagen descargada de: [Github: AudioSignalProcessingForML](#).

Si entendemos el habla como la convolución de la frecuencia de respuesta del tracto vocal con el pulso glotal, aplicando la ecuación [1.3] para obtener el cepstrum se pueden separar las 2 componentes del habla y calcular su espectro de manera bien diferenciada:

$$\begin{aligned} x(t) &= e(t) \cdot h(t) \rightarrow \mathcal{F}(x(t)) = X(t) = E(t) \cdot H(t) \rightarrow \\ &\rightarrow \log(X(t)) = \log(E(t)) + \log(H(t)) \end{aligned}$$

donde $x(t)$ es el habla, $e(t)$ es la frecuencia de respuesta del tracto vocal y $h(t)$ son los pulsos glotales

El cepstrum resultante se traduce en unas quefrecuencias bien diferenciadas: las bajas serán de la frecuencia de respuesta del tracto vocal y las altas del pulso glotal. Quedándonos solo con las quefrecuencias bajas estamos extrayendo la información de la frecuencia de respuesta del tracto vocal, que contiene los formantes que caracterizan bien cada fonema. Así vemos la potencia del cepstrum.

Con toda esta explicación se pueden explicar de dónde salen los MFCCs. Aplicando algunos cambios a la ecuación del cepstrum, se obtiene la Ecuación [1.4], que devuelve los MFCCs.

$$C(x(t)) = \mathcal{C}(\log(mels[\mathcal{F}(x(t))])) \quad (1.4)$$

donde \mathcal{F} es la transformada de Fourier, \mathcal{C} es la transformada coseno discreta, $mels$ hace referencia a la transformación de las frecuencias a la escala de Mel, $x(t)$ es la señal temporal de audio, y $C(x(t))$ son los MFCCs

Vemos que las únicas diferencias son la transformación de las frecuencias a la escala de Mel y el uso de la transformada coseno discreta. Se utiliza esta transformada porque es una versión simplificada de la transformada de Fourier que devuelve coeficientes reales, decorrelaciona la energía en las diferentes bandas de Mel y reduce el número de dimensiones para representar el espectro. De esta operación se pueden extraer un máximo

de 39 coeficientes de cada segmento temporal enventanado que se esté procesando. En el reconocimiento de lenguaje se suelen utilizar los primeros 12-20 coeficientes, ya que son los que guardan la información más relevante, relacionada con los formantes. Al final, se obtienen unos datos transformados, cuyas dimensiones son el número de MFCCs escogidos y el número de segmentos temporales procesados.

1.2. Reconocimiento de lenguaje o *speech recognition*

Entendidas las bases fundamentales del audio digital, de los datos de audio y de los conceptos y magnitudes que vamos a utilizar en este trabajo, pasamos a describir su campo de aplicación: el reconocimiento de lenguaje.

De manera concisa y directa, el reconocimiento de lenguaje es una disciplina de la inteligencia artificial que tiene como objetivo permitir la comunicación de forma hablada entre los seres humanos y los ordenadores o dispositivos digitales. Es un campo en auge y con gran importancia ya que, debido a la inmersión digital que estamos experimentando desde hace poco más de una década, la interacción humano-máquina tiene lugar (o tendrá en poco tiempo) en un gran número ámbitos, situaciones, contextos y formas diferentes, de manera cotidiana. Es por este motivo que generar modelos y sistemas que faciliten la comunicación y el entendimiento de una manera fluida y realista en todos los aspectos, puede suponer un gran avance tecnológico con aplicaciones que pueden mejorar mucho la vida de las personas.

Para entender la amplitud e importancia de este campo, hacemos un breve resumen de la evolución histórica que ha tenido desde sus inicios en los años 70 llegando hasta la década del 2010 donde se han producido los mayores avances gracias al desarrollo tecnológico y el crecimiento acelerado de campos como la ciencia de datos, el aprendizaje máquina o la inteligencia artificial. También comentaremos cuál es el estado del arte y se mencionarán algunas de las muchas ramas que abarca este extenso campo que han ido apareciendo hace relativamente pocos años.

1.2.1. Evolución histórica y estado del arte

El reconocimiento de lenguaje tal y como lo hemos definido tiene sus orígenes en la década de los 70 [5,6]. Antes de eso se llevaban a cabo investigaciones relacionadas con el lenguaje, pero iban sobre todo enfocadas a la síntesis y la reproducción de audio hablado. Sin embargo, el campo ya se iba preparando porque se crearon sistemas que eran capaces de reconocer una única voz que dijese números en alto, e incluso IBM desarrolló “Shoebox”, un sistema capaz de responder a 16 palabras en inglés.

Pero en los años 70 se inventaron los modelos ocultos de Markov (HMM, por sus siglas en inglés), modelos estadísticos que devolvían una secuencia de símbolos o valores y era capaz de detectar patrones. Viendo la gran potencia que esto podría tener, se apostó mucho por esta tecnología y hubo grandes inversiones y muchas líneas de investigación que provocaron grandes avances en la década. Por ejemplo, el sistema “Carnegie Mellon’s

Harpy” fue capaz de entender unas 1000 palabras, y se crearon sistemas capaces de interpretar diferentes voces.

En los años 80 fue cuando se empezaron a utilizar los modelos ocultos de Markov, por lo que los sistemas de reconocimiento de lenguaje pasaron de algunos cientos a varios miles de palabras. En lugar de utilizar palabras y buscar patrones en el sonido, estos modelos estimaban la probabilidad de que sonidos desconocidos fuesen palabras. También en esta década se fundaron empresas como Dragon Systems o SpeechWorks, que empezaron a llevar el reconocimiento de lenguaje al ámbito comercial.

En la década de los 90 el reconocimiento de lenguaje sufrió un impulso muy grande debido sobre todo a la aparición del ordenador personal, junto con el desarrollo de procesadores más potentes. La empresa BellSouth introdujo el portal de voz VAL, que era un sistema de reconocimiento de voz interactivo de acceso telefónico. De la mano de Dragon Systems fueron apareciendo nuevos sistemas cada vez más refinados.

A inicios del siglo XXI la tecnología de reconocimiento del lenguaje había alcanzado precisiones del 80%, pero parecía que estaba estancada ya que los planteamientos que se seguían para la detección de palabras no habían evolucionado mucho. Sin embargo, por el 2001 llegó Google y lanzó el Google Voice Search, que puso el poder del reconocimiento del lenguaje en manos de millones de personas y que el propio Google utilizó para recopilar datos de muchas búsquedas que ayudaban a predecir qué es lo que estaba diciendo la persona. A nivel teórico se utilizaron métodos de procesamiento en paralelo haciendo uso de combinaciones de modelos ocultos de Markov y enfoques acústicos y fonéticos para aumentar la robustez de los modelos de reconocimiento de lenguaje en entornos ruidosos.

Pero es en la década pasada cuando se experimenta el mayor crecimiento gracias a la explosión de Internet y de los dispositivos inteligentes personales. Aparecen muchos programas y aplicaciones que acercan el reconocimiento de voz al grueso de la población de una manera muy directa, por ejemplo con Siri por parte de Apple y Alexa por parte de Amazon, que además recopilan una gran cantidad de datos con los que trabajar y mejorar. De manera paralela, el desarrollo tecnológico hace posible el procesado y la extracción de información de estos datos masivos que se generan en el mundo digital continuamente conectado. Esto permite que se desarrollen nuevos modelos entrenados con un conjunto de datos cada vez más grande y diverso, y se empiezan a aplicar modelos con un enfoque basado en inteligencia artificial como lo entendemos actualmente.

Es así como se llega al enfoque basado en redes neuronales profundas, que constituye el estado del arte hoy en día. Debido a la gran capacidad de procesamiento y de computación en general se han podido desarrollar y entrenar redes de muchos millones de parámetros, con arquitecturas y enfoques altamente complejos, que ya están superando el límite de precisión humana. Por poner algunos ejemplos, Wav2Letter++ [7], Wav2Vec 2.0 [8] y DeepSpeech [9] son modelos desarrollados por gigantes tecnológicos como Facebook o empresas del Silicon Valley AI Lab que proponen enfoques particulares y mejoras o nuevas características que enriquecen mucho el problema, y que sientan la base para futuros desarrollos todavía más complejos que poco a poco vayan mejorando los resultados, la velocidad de entrenamiento y ejecución, o los recursos necesarios.

1.2.2. Detección de palabras de activación

El reconocimiento de lenguaje es un campo muy amplio que, debido a los grandes avances que hemos comentado, ha derivado en diversas ramas o problemas concretos que requieren ser enfocados de una forma particular para obtener buenos resultados. Aunque todas ellas tienen puntos comunes, cada rama es particular y requiere un enfoque concreto, con unos datos adecuados.

Un mismo conjunto de datos tratado y procesado de diferentes formas puede ser utilizado para abordar diferentes aspectos del reconocimiento de lenguaje. Por ejemplo, asignando diferentes etiquetas a los audios podemos entrenar un modelo que sea capaz de detectar unas cosas u otras. O por ejemplo, dividiendo los audios de determinadas formas, podemos conseguir que el modelo final actúe o esté preparado para diversas acciones. Teniendo esto en cuenta, algunas de las ramas que podemos encontrar dentro del reconocimiento de lenguaje son:

- reconocimiento de lenguaje de carácter general, para mantener conversaciones humano-máquina
- detección de palabras de activación
- transcripción o *speech to text* (STT)
- reconocimiento de la persona que habla
- detección de sentimientos y emociones
- ...

En este trabajo nos centramos en la rama de detección de palabras de activación. Puede parecer que no es un campo en sí mismo ya que en cierto modo puede ser una particularización del reconocimiento de lenguaje de carácter general. Sin embargo, la forma de entrenar un modelo para que lleve a cabo esta acción requiere un planteamiento determinado para obtener buenos resultados. Un ejemplo muy claro es la diferente forma de enfocar los datos de audio para entrenar el modelo: un modelo de reconocimiento de lenguaje de carácter general se debería entrenar con audios de frases largas, mientras que en el caso de las palabras de activación la unidad a reconocer es la palabra concreta, no una frase entera. Es por ello que el planteamiento de ambos problemas presenta diferencias radicales. Además, en el caso de detección de palabras de activación la gran mayoría de los datos de entrada que recibirá el modelo serán de silencio o de ruido de fondo, no de lenguaje en sí, por lo que los falsos positivos se deben minimizar. Por otro lado, prácticamente todos los datos de entrada de lenguaje en sí no estarán centrados en la palabra de activación a detectar, por lo que el modelo no debería activarse en lenguaje común y arbitrario.

Del mismo modo, en el caso de las palabras de activación y teniendo en cuenta la finalidad del trabajo de poder llevarlo a cabo en un dispositivo de Tiny ML, el modelo debe tener unas características determinadas para que la función se lleve a cabo correctamente. Por ejemplo, la respuesta debe ser rápida y sin latencia, ya que la activación debe preparar

al dispositivo para una interacción posterior. Y añadido a esto, el modelo debe ser pequeño y la cantidad de computación debe ser reducida para que pueda ser desplegado en los dispositivos y la ejecución sea rápida.

Teniendo todo en cuenta, a lo largo del trabajo se han intentado desarrollar diversos modelos que cumpla estas características, utilizando dos conjuntos de datos adecuados para que el entrenamiento sea efectivo y los resultados obtenidos sean aceptables.

1.3. Tiny ML

Para cerrar este capítulo de introducción teórica nos vamos a centrar en el Tiny ML, ya que recordemos que el objetivo final del trabajo no es solo el desarrollo de los modelos de reconocimiento de lenguaje, sino también el ser capaces de ponerlos en funcionamiento en la placa Coral Dev de Google.

De manera muy resumida, vamos a ver qué se entiende realmente por Tiny ML y qué relación tiene esta pequeña revolución tecnológica, que ha aparecido de forma muy discreta, con otros aspectos relacionados como son el IoT o el Edge Computing. Por último, veremos qué papel juegan las llamadas TPUs en estos pequeños dispositivos y qué ventajas ofrecen en comparación con las clásicas CPUs y GPUs.

1.3.1. ¿Qué es el “Tiny ML”?

El Tiny ML se define en términos generales como un campo dentro de las tecnologías y aplicaciones del aprendizaje automático que se centra tanto en el hardware (circuitos integrados dedicados y microcontroladores) como en el software, y que busca que algoritmos y modelos se puedan ejecutar en los propios dispositivos, con un consumo muy bajo de energía, permitiendo que estén siempre activos y listos para funcionar. Los llamados dispositivos de Tiny ML ofrecen unas prestaciones mucho más reducidas en comparación con grandes ordenadores y servidores, por lo que es necesario una preparación previa de los modelos para poder ejecutarlos de la manera más eficiente posible.

Se trata de un campo que está en crecimiento debido al auge del IoT o Internet de las cosas, un paradigma reciente según el que prácticamente cualquier dispositivo electrónico está conectado a Internet y puede enviar información y datos que recoge por medio de sensores. En sus inicios, el dispositivo IoT solo actuaba como sensor y se centraba en el envío de los datos a un servidor u ordenador donde eran procesados, pero el Tiny ML supone un cambio radical ya que los propios dispositivos de IoT podrían llevar a cabo el procesado y la ejecución de los modelos. Esto va en consonancia con lo que se conoce como Edge Computing, una filosofía según la que los datos no deben abandonar el dispositivo donde se han generado, ya que todo se debería llevar a cabo a nivel local. Vemos así como el Tiny ML une el mundo de los dispositivos de bajo coste y bajo consumo con el mundo del aprendizaje máquina, proporcionando un gran número de aplicaciones en muchos campos diferentes.

El Tiny ML proporciona así una serie de ventajas muy interesantes [10]:

- baja latencia: como los modelos de inteligencia artificial se ejecutan localmente, sin necesidad de enviar nada a servidores externos, el tiempo de espera se reduce de manera considerable. Solo esta característica abre las puertas a multitud de aplicaciones en las que el tiempo de respuesta juega un papel fundamental
- bajo consumo de energía: el Tiny ML se lleva a cabo en dispositivos gobernados por microcontroladores, pequeños ordenadores diseñado par realizar una tarea determinada, y que tienen consumos de energía del orden de los milivatios. Esto abre la puerta a aplicaciones en las que los dispositivos no sean accesibles de manera sencilla, ya que su autonomía les permite estar en funcionamiento mucho tiempo
- bajo ancho de banda: como los datos no necesitan ser enviados al servidor de manera constante, se puede disponer de un menor ancho de banda en la conexión. Esto es muy positivo en el caso de problemas en los que se generen grandes volúmenes de datos muy rápidamente, ya que no es necesario enviarlos todos, sino que se pueden tratar en el propio dispositivo y, si es necesario, enviar solo una parte
- privacidad: como los datos no salen en ningún momento del dispositivo (e incluso es posible que el dispositivo no esté conectado a Internet o que la conexión sea muy limitada) no hay problemas de ciberseguridad
- independencia sin interrupciones: como no es necesario que los datos se envíen a un servidor donde van a ser procesados, no se depende de ningún servicio o conexión a Internet que pueda detener el proceso en caso de que dichos servicios se encuentren caídos o colapsados

Pero por otro lado también aparecen algunos problemas que se deben tener en cuenta:

- los dispositivos de Tiny ML pueden estar en lugar remotos y de difícil acceso, por lo que actualizar el software o el modelo puede ser costoso
- la cantidad de memoria para almacenar datos es limitada, por lo que pueden aparecer problemas en casos en que la volumetría sea significativa
- los modelos que se pueden ejecutar en los dispositivos no son los modelos originales, ya que es necesario reducirlos y convertirlos convenientemente para que puedan ser ejecutados con todas las prestaciones que ofrece el dispositivo. Esto puede traducirse en modelos con peores resultados

A pesar de todo esto, el IoT sigue creciendo y eso hace que las perspectivas para el Tiny ML también lo hagan. Ya hay empresas que ofrecen servicios de TinyML a demanda, TinyML-as-a-service. Las ventajas que ofrece este tipo de tecnología son muy potentes y, como se ha ido diciendo, las aplicaciones que puede tener en multitud de campos diferentes son muy amplias.

1.3.2. TPUs para potenciar el Tiny ML

Además de todas las ventajas que se han indicado previamente, hay un factor extra que se suma a todas ellas para enriquecer todavía más el valor que proporcionan los dispositivos de Tiny ML. Esta característica exclusiva es la unidad de procesamiento tensorial, TPU (por sus siglas en inglés).

La TPU es un circuito integrado de aplicación específica (ASIC) en aceleradores de inteligencia artificial, desarrollado por Google, para llevar a cabo tareas de inferencia de redes neuronales. Es una unidad de procesamiento adaptada y optimizada para TensorFlow, que ha sido creada específicamente para el aprendizaje automático. En comparación con las unidades de procesamiento gráfico o GPUs, que se usan con frecuencia y cuyo diseño es específico para realizar cálculos con número en coma flotante, las TPUs están diseñadas para llevar a cabo un mayor volumen de cálculo en operaciones de precisión reducida, por ejemplo con una precisión de tan solo 8 bits.

A pesar de que el término TPU fue acuñado para este circuito específico diseñado por Google para el framework de TensorFlow, también han aparecido otros diseños de aceleradores de inteligencia artificial en otros proveedores, de manera que se está convirtiendo en un estándar plenamente reconocido.

La primera TPU se anunció en 2016, pero hoy en día ya hay 5 generaciones diferentes, cada una con mejoras significativas con respecto a la anterior y siendo la última de 2021, mostrando que es un mercado que está vivo. La revolución para el Tiny ML vino con la 3^a generación, ya que hasta ese momento las TPUs eran dispositivos muy potentes pero destinados a grandes servidores y ordenadores potentes. Sin embargo, el año 2018 se anunció la primer versión de la Edge TPU, una TPU de tan solo 5x5 mm que se ponía a disposición de los desarrolladores del mundo de la inteligencia artificial en una línea de productos bajo la marca Coral.

La placa Coral Dev es el dispositivo de Tiny ML que hemos utilizado para llevar a cabo este trabajo, y es una placa que incorpora la Edge TPU. En el capítulo siguiente explicaremos de manera más detallada las especificaciones propias tanto de la placa como de la TPU, gracias a la cual hemos podido introducirnos, aunque sea de manera superficial, en el mundo del Tiny ML.

Capítulo 2

Herramientas, datos y modelos

Una vez introducidos los conceptos teóricos que se van a ir utilizando a lo largo del desarrollo, pasamos a explicar de modo general cuál ha sido el procedimiento seguido. Para ello vamos a detallar cuáles han sido los conjunto de datos utilizados para las diferentes partes del trabajo, así como los modelos planteados y aplicados. Pero previamente es necesario explicar cuál ha sido la herramienta principal que ha guiado el planteamiento del trabajo.

Es cierto que todo el código ha sido escrito en Python y que los frameworks que principalmente se han utilizado para la definición de los modelos han sido TensorFlow 2.0 [11] y Keras [12], pero ha sido la placa Coral Dev de Google la que desde el principio y de manera general ha ido condicionando los pasos a seguir en cada uno de los procesos, ya que el objetivo final del trabajo es la ejecución de los modelos de reconocimiento de lenguaje en un dispositivo de Tiny ML. Por ello, primero explicaremos a fondo qué es la placa Coral Dev, cuáles son sus especificaciones, qué requisitos tiene a la hora de ejecutar los modelos y el procedimiento paso a paso que se ha seguido para poder utilizarla.

Posteriormente se explicarán cuáles son los conjuntos de datos que se han utilizado en las diferentes partes del trabajo. En la parte del entrenamiento desde cero de modelos de reconocimiento de lenguaje se explicará qué conjunto de datos se ha escogido y por qué, así como el tratamiento completo que ha requerido el conjunto para enriquecerlo y adecuarlo, junto con la transformación que se ha hecho de los datos para convertirlos en la información relevante que se ha utilizado para entrenar los modelos. Del mismo modo, en la parte del transfer learning sobre un modelo complejo previamente entrenado para ver si es posible reentrenar un modelo con datos en otro idioma, se explicará el procedimiento que se ha seguido para la recolección, preprocessado y estandarización de los datos.

Finalmente, mostraremos los modelos que se han utilizado para cada uno de los casos, así como los procesos que se han tenido que seguir y realizar sobre ellos para transformarlos y que puedan ser ejecutados correctamente en la placa Coral Dev, utilizando todos los recursos y el potencial que ofrece.

2.1. Placa Coral Dev, de Google

La placa Coral Dev de Google [13] es el dispositivo de Tiny ML que se ha utilizado para ejecutar los modelos de reconocimiento de lenguaje que se han creado, entrenado y preparado, y que ha servido para mostrar y verificar qué es lo que hay que hacer para desplegar modelos complejos de machine learning en dispositivos de este tipo, haciendo uso de toda su potencia.

De modo simplificado, este dispositivo es una computadora de una sola placa, es decir, un ordenador completo en un solo circuito. La peculiaridad de esta placa, y lo que la diferencia de otras computadoras de una sola placa como pueden ser algunos modelos de Raspberry Pi o Arduino, es que está diseñada para ejecutar modelos de machine learning, ya que dispone de un acelerador de inteligencia artificial, el Edge TPU, cuyo objetivo es llevar a cabo tareas de inferencia para permitir el funcionamiento de algoritmos de machine learning de forma local, sin necesidad de que los datos necesiten ser enviados por Internet para ser procesados. Todo esto se consigue gracias a la tecnología Coral, una plataforma de hardware y software para construir modelos neuronales de rápida inferencia y que ofrece diversos dispositivos (la placa Dev, la placa Dev-mini, un acelerador de USB y en un futuro próximo la placa Dev-micro) para realizar Tiny ML. Mostramos en la Figura (2.1) las partes delantera y trasera de la placa Coral Dev.

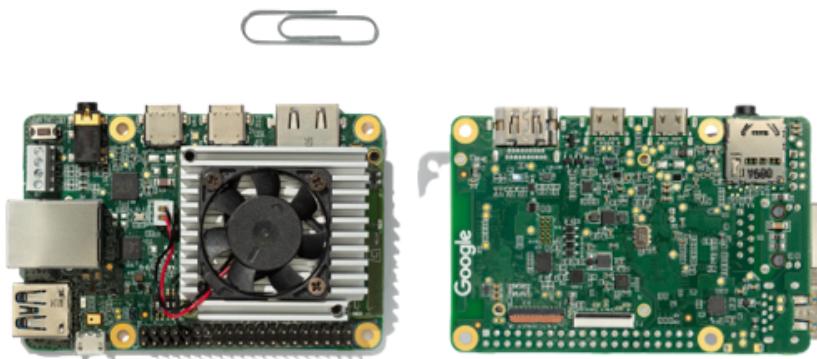


Figura 2.1. Partes delantera y trasera de la placa Coral Dev, donde se puede apreciar su tamaño comparado con el de un clip.

La placa Coral Dev que utilizamos en este trabajo está enfocada a desarrolladores que buscan crear prototipos de una manera más o menos sencilla y directa, para luego llevarlos a producción combinando el SoM (*System-on-Module*) con su propio hardware. Ofrece buenas prestaciones para llevar a cabo Tiny ML a un precio muy asequible y tiene las características propias de la tecnología Coral. Algunas de ellas son:

- rendimiento: la Edge TPU (de ahora en adelante TPU) es capaz de ejecutar 4×10^{12} operaciones por segundo (TOPS), utilizando solo 0.5 vatios (W) por cada TOP
- compatibilidad de modelos: la TPU soporta una gran variedad de arquitecturas generadas con TensorFlow, incluyendo aquellas creadas con Keras. El flujo de trabajo

general para crear modelos se basa exclusivamente en TensorFlow, haciendo que no sean necesarias otras APIs, aunque es necesario transformar los modelos a TensorFlow Lite (TFLite)

- modelos pre-compilados: se han verificado muchas arquitecturas de modelos que son conocidos, por lo que se pueden llevar a cabo tareas de clasificación de imágenes, detección de objetos, detección de palabras, etc. de manera sencilla
- Mendel Linux: para facilitar el desarrollo se ha creado un derivado de Debian conocido como Mendel, que ha sido optimizado para sistemas embebidos, haciendo que sea muy ligero. Además, viene instalado con herramientas típicas en la creación de modelos como son Python, C++, algunas librerías para la TPU y el MDT (*Mendel Development Tool*) para facilitar la conexión con la placa
- permite la inferencia simultánea de diferentes modelos en la misma TPU, siempre que los modelos se hayan compilado conjuntamente, y el reentrenamiento de modelos en el propio dispositivo llevando a cabo transfer learning

Ahora que tenemos una idea de qué es la placa Coral Dev y qué es lo que ofrece, pasamos a ver en más detalles las especificaciones, los requisitos que se deben cumplir con los modelos para que se ejecuten haciendo uso de toda la potencia disponible y la preparación que se hizo de la placa para poder utilizarla.

2.1.1. Especificaciones

Es interesante conocer de manera un poco más técnica el hardware que estamos manipulando, por lo que mostramos en la Tabla (2.1) algunas de las especificaciones más importantes de la placa⁽²⁾.

Tabla 2.1. Especificaciones técnicas principales de la placa Coral Dev.

CPU	NXP i.MX 8M SoC (quad Cortex-A53, Cortex-M4F)
GPU	GC7000 Lite Graphics Integrado
Acelerador ML (TPU)	Google Edge TPU co-procesador: 4 TOPS (int8); 2 TOPS/W
RAM	1, 2 o 4 GB LPDDR4
Memoria flash	8 GB eMMC, MicroSD slot
Wireless	Wi-Fi 2x2 MIMO (802.11b/g/n/ac 2.4/5GHz) y Bluetooth 4.2
LAN	Puerto Gigabit Ethernet
USB	Type-C OTG y alimentación, Type-A 3.0 host, micro-B serial console
Audio	jack de 3.5 mm, micrófono digital y terminal para altavoces
Video	HDMI 2.0a, conector a monitor y conector para cámara
Alimentación	5 V DC (USB Type-C)
Dimensiones	88x60x24 mm

⁽²⁾Para ver las especificaciones al completo, con toda la información detallada, es recomendable visitar la web de la Coral Dev Board [13] donde se detallan a nivel técnico todos los componentes que la componen.

2.1.2. Compatibilidad y requisitos para el correcto funcionamiento

Para que la TPU de la placa puede proporcionar altas velocidades de ejecución de los modelos neuronales con un bajo consumo energético se tienen que cumplir una serie de requisitos, que van desde el uso exclusivo de ciertas arquitecturas y operaciones en la red neuronal hasta la correcta cuantización y compilación del modelo.

De manera breve y general, la TPU solo soporta modelos de tipo TFLite que han sido cuantizados a enteros de 8 bits y compilados específicamente para su ejecución en la TPU. TFLite es una versión ligera de TensorFlow, destinada precisamente para dispositivos móviles y edge, que ofrece una muy baja latencia al realizar inferencia, y con el que tanto los modelos como el propio kernel son mucho más pequeños en tamaño. No se puede entrenar directamente un modelo TFLite, sino que se tiene que convertir un modelo normal previamente entrenado. Mostramos en la Figura (2.2) el flujo general que deben seguir los modelos para que sean compatibles y puedan ejecutarse en la TPU de la placa.

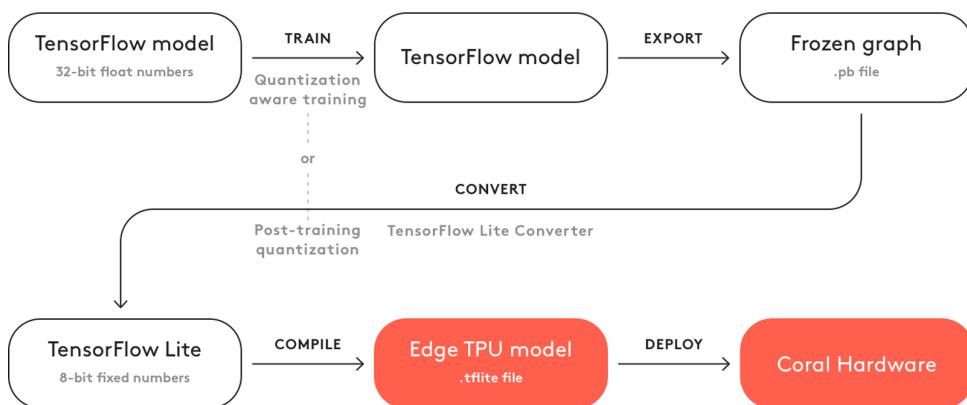


Figura 2.2. Resumen del flujo que deben seguir los modelos para ser compatibles y poder ser ejecutados en la TPU de la placa Coral Dev.

En la figura se observan las partes más relevantes del flujo de los modelos. Se parte de un modelo normal generado con TensorFlow que se debe cuantizar, ya sea realizando un entrenamiento extra que lo prepare para la cuantización final (pre-cuantización) o directamente cuantizando el modelo entrenado (post-cuantización). De este modo, se obtiene un modelo cuantizado a enteros de 8 bits en formato .tflite. Este modelo se debe compilar de una manera concreta para preparar su ejecución en la TPU, y así poder proceder con su despliegue en la placa.

A continuación se describen un poco más en detalle algunos aspectos fundamentales que se deben tener en cuenta para que la transformación y adaptación del modelo se pueda llevar a cabo sin problemas. Concretamente, vamos a comentar los requisitos generales que debe tener el modelo, las operaciones permitidas en las estructuras neuronales y cómo llevar a cabo los procesos de cuantización y compilación para la TPU.

Requisitos generales del modelo

Para construir un modelo en TensorFlow que sea capaz de explotar toda la potencia que ofrece la TPU para llevar a cabo la inferencia, se deben cumplir los siguientes requisitos generales:

- los parámetros de los tensores deben estar cuantizados a enteros de 8 bits, en formato *int8* o *uint8*
- los tamaños de los tensores deben ser constantes durante el tiempo de ejecución, o lo que es lo mismo, no se permiten tensores de tamaños dinámicos
- los parámetros de los modelos (como los bias de los tensores) deben ser constantes durante el tiempo de ejecución
- los tensores deben ser 1D, 2D o 3D. Si un tensor tiene más de 3 dimensiones, solo las 3 dimensiones internas pueden tener un tamaño superior a 1
- el modelo debe utilizar exclusivamente las operaciones soportadas por la TPU (se muestran más adelante cuáles son estas operaciones)

A efectos prácticos estos requisitos se traducen en: tener que llevar a cabo la cuantización del modelo, indicar de manera explícita los tamaños de los tensores sin permitir que varíen, tener en cuenta la limitación de las dimensiones de los tensores, y utilizar o definir capas con operaciones permitidas.

No cumplir con alguno de estos requisitos se puede traducir en que el modelo no se puede compilar para ser ejecutado en la TPU o, en el mejor de los casos, que solo una parte de las operaciones se puedan llevar a cabo en ella.

Operaciones permitidas

Mostramos en la Tabla (2.2) el conjunto de operaciones (con sus correspondientes limitaciones) que se pueden utilizar para que el modelo se ejecute completamente en la TPU.

Vemos que el número de operaciones es bastante elevado, aunque por ejemplo se echan de menos algunas tan importantes como la de BatchNormalization, muy común y muy utilizada en redes convolucionales.

Como se ha comentado en el apartado anterior, si se utilizan operaciones no permitidas solo aquellas que sí que lo sean podrán ser ejecutadas en la TPU, pudiendo afectar negativamente de manera significativa a la ejecución completa del modelo.

Tabla 2.2. Conjunto de operaciones, y sus correspondientes limitaciones, permitidas en la arquitectura de los modelos para que se puedan ejecutar en la TPU.
El símbolo (*) indica que hay matices a consultar en la web oficial.

Operación	Versión runtime	Limitaciones
Add	Todas	
AveragePool2d	Todas	Sin funciones de activación fusionadas
Concatenation	Todas	Sin funciones de activación fusionadas (*)
Conv2d	Todas	Misma dilatación en las dimensiones x, y
DepthwiseConv2d	≥ 13	Misma dilatación en las dimensiones x, y
ExpandDims	≥ 13	
FullyConnected	Todas	Formato por defecto de los pesos. Tensor de salida 1D
L2Normalization	Todas	
Logistic	Todas	
LSTM	≥ 14	LSTS unidireccionales
Maximum	Todas	
MaxPool2d	Todas	Sin funciones de activación fusionadas
Mean	≥ 13	Sin reducción en la dimensión de batch (*)
Minimum	Todas	
Mul	Todas	
Pack	≥ 13	Sin packing en la dimensión batch
Pad	≥ 13	Sin packing en la dimensión batch
PReLU	≥ 13	Alfa debe ser 1D (solo la dimensión interna puede ser >1) (*)
Quantize	≥ 13	
ReduceMax	≥ 14	No puede operar en la dimensión batch
ReduceMin	≥ 14	No puede operar en la dimensión batch
ReLU	Todas	
ReLUN1To1	Todas	
Reshape	Todas	Ciertos reshapes no mapeados si los tensores son grandes
ResizeBilinear	Todas	Input/Output es 3D (según el tamaño se ejecuta o no en TPU)
ResizeNearestNeighbor	Todas	Input/Output es 3D (según el tamaño se ejecuta o no en TPU)
Rsqrt	≥ 14	
Slice	Todas	
Softmax	Todas	Tensor de entrada 1D con máximo 16 000 elementos
SpaceToDepth	Todas	
Split	Todas	No splits en la dimensión batch
Squeeze	≥ 13	
StridedSlice	Todas	strides = 1, ellipsis-axis-mask = 0, new-axis-max = 0
Sub	Todas	
Sum	≥ 13	No puede operar en la dimensión batch
Squared-difference	≥ 14	
Tanh	Todas	
Transpose	≥ 14	
TransposeConv	≥ 13	

Cuantización y compilación para la TPU

Cuantizar el modelo significa convertir todos los números en coma flotante de 32 bits en los correspondientes enteros de 8 bits más cercanos⁽³⁾. Esto hace que el modelo sea mucho más pequeño y rápido, aunque las representaciones en 8 bits puede suponer una disminución en la precisión y accuracy del modelo. Los modelos se pueden cuantizar total o parcialmente. Una cuantización total cuantiza los pesos y las activaciones de las capas, mientras que una cuantización parcial o híbrida solo cuantiza los pesos.

La cuantización del modelo se puede llevar a cabo de 2 formas diferentes:

- pre-cuantización: se lleva a cabo un entrenamiento preparado para la cuantización, en el que se utilizan nodos de cuantización falsos en los nodos de la red para simular el efecto de utilizar enteros de 8 bits. Este proceso requiere modificar la red antes del entrenamiento, pero generalmente resulta en un modelo cuantizado con mejores resultados que si se lleva a cabo la post-cuantización ya que hace que el modelo sea más tolerante a precisiones menores por el ajuste que se lleva a cabo en los pesos, en lugar de transformarlos directamente
- post-cuantización: no requiere que se hagan modificaciones sobre la red, por lo que permite cuantizar cualquier modelo que ya ha sido previamente entrenado. Este proceso requiere un conjunto de datos representativo, un conjunto de datos con la misma estructura y que más o menos abarque el espacio del problema, para poder llevar a cabo una correcta conversión teniendo en cuenta los rangos dinámicos de las activaciones y los inputs. Algo importante a tener en cuenta es que no todas las operaciones permitidas en TFLite son compatibles con este tipo de cuantización, por lo que es posible que ciertas operaciones no se conviertan y no puedan ser ejecutadas en la TPU

A pesar de que la TPU requiere que el modelo esté cuantizado, se permite que tanto las entradas como las salidas del modelo estén en coma flotante. Internamente se lleva a cabo una operación de cuantización en la CPU en la entrada y la salida, pero esto se puede traducir en pequeñas latencias (generalmente despreciables para la mayoría de modelos) si el tensor de entrada es muy grande.

Una vez el modelo está completamente cuantizado, solo falta compilarlo para su ejecución en la TPU. Este proceso se lleva a cabo con el *edge TPU compiler*. Este compilador genera un grafo que se procesa en la propia placa, en el que se indican las operaciones que se pueden llevar a cabo en la TPU. Si no se cumplen con los requisitos que se han comentado, el compilador es posible que no dé fallos, pero indicará que solo una parte del modelo puede ejecutarse en la TPU. En el momento en el que se detecte una operación no soportada, el compilador dividirá el grafo en 2 partes: una con las primeras operaciones que sí se pueden ejecutar en la TPU, y otra con el resto de operaciones a partir de la

⁽³⁾En general el proceso de cuantizar se refiere a transformar los números en un formato más simple. Nosotros nos centramos exclusivamente en el caso de cuantización a enteros de 8 bits porque es lo requerido para que el modelo se pueda ejecutar en la TPU de la placa Coral Dev, pero por ejemplo también se podrían llevar a cabo cuantizaciones a enteros de 32 bits.

primera no válida que tendrá que ser ejecutada en la CPU. Mostramos un resumen de este proceso en la Figura (2.3).

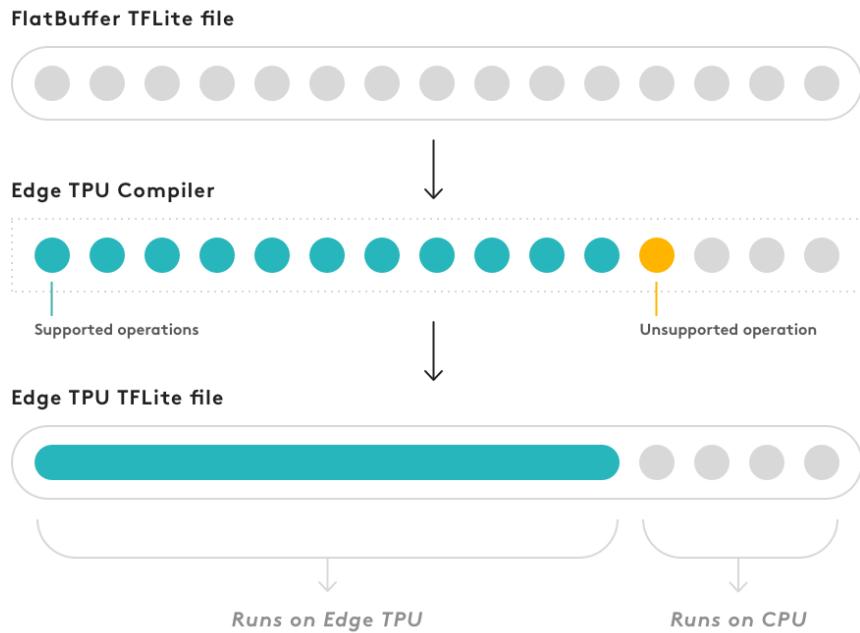


Figura 2.3. Proceso seguido por el compilador para crear los grafos de ejecución de la TPU.

Si una parte del modelo no puede ejecutarse en la TPU y por ello debe ejecutarse en la CPU, es normal y esperable obtener rendimientos significativamente inferiores en cuanto a la velocidad de inferencia. Puede ser interesante probar diferentes estructuras, incluso incluyendo operaciones no permitidas en posiciones clave del modelo, para en cierto modo lograr un balance entre el rendimiento y los resultados del modelo.

2.1.3. Preparación de la placa

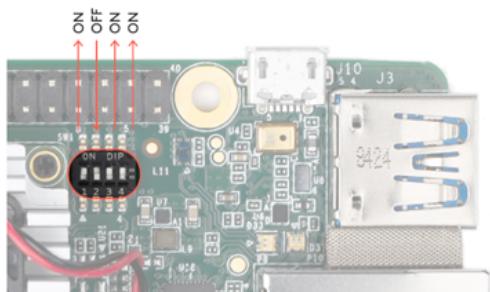
Para cerrar esta parte de descripción y funcionamiento de la placa Coral Dev, explicamos de manera breve el proceso que se siguió para prepararla para llevar a cabo la ejecución de modelos TFLite en ella. El hardware necesario para poder utilizar la placa es:

- un ordenador que sirva de host, con Python3 instalado. Como en nuestro caso se utilizaba un ordenador con Windows 10 instalado, fue necesario descargarse también “Git for Windows”, un programa que proporciona acceso a la terminal Git Bash en la que poder ejecutar los comandos necesarios
- una tarjeta microSD con al menos 8 gigas de capacidad y un adaptador para conectarlo al ordenador host
- una fuente de alimentación (2-3 A, 5 V) con USB-C, como un cargador de móvil
- un cable USB-C a USB-A para conectar la placa al ordenador
- una conexión Wi-Fi

Una vez se dispone del material necesario, es posible configurar la placa para su funcionamiento. Indicamos a continuación los pasos que hay que seguir para ello, **con las particularidades de utilizar un ordenador con Windows 10 como host**:

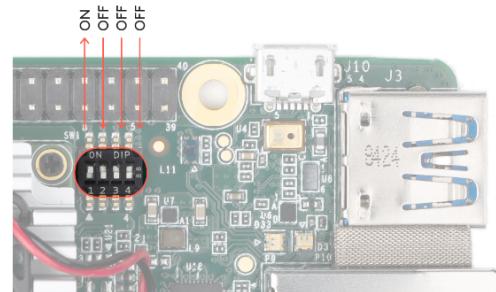
1. flasheo de la placa con Mendel Linux. Por defecto la placa viene solo con un U-Boot bootloader instalado, por lo que lo primero que hacer que hacer es instalar el sistema operativo Mendel Linux. Esto es muy sencillo ya que hay disponible imágenes que se encargan de ello, utilizando la tarjeta SD, aunque también se puede flashear el sistema manualmente. En nuestro caso utilizamos la tarjeta SD:
 - a) primero nos descargamos la última versión de la imagen de Mendel Linux (en nuestro caso: 5.3 Eagle, noviembre 2021)
 - b) utilizando un programa como BalenaEtcher [14] se flashea la imagen en la SD
 - c) con la placa totalmente desconectada, se cambia el modo de inicio de la placa, fijando los interruptores como se muestra en la Figura (2.4a)
 - d) se introduce la tarjeta SD en la placa
 - e) se conecta el cable de alimentación de tipo USB-C al puerto PWR. La placa debería encenderse (indicado por un LED rojo) y el proceso de flasheo de Mendel Linux se debería llevar a cabo en 5-10 minutos. Se puede conectar un cable HDMI a una pantalla para ver qué procesos se están ejecutando
 - f) cuando el LED rojo se apague, el proceso ha finalizado. Se desconectan los cables y se retira la tarjeta SD
 - g) se vuelve a cambiar el modo de inicio de la placa, fijando los interruptores como se muestra en la Figura (2.4b)
 - h) se conecta el cable de alimentación y se conecta la placa al ordenador host con el cable USB-C a USB-A, y se esperan unos minutos a que todo esté preparado

Boot mode	Switch 1	Switch 2	Switch 3	Switch 4
SD card	ON	OFF	ON	ON



(a)

Boot mode	Switch 1	Switch 2	Switch 3	Switch 4
eMMC	ON	OFF	OFF	OFF



(b)

Figura 2.4. Posiciones de los interruptores para los diferentes tipos de inicio de la placa. La Figura (2.4a) muestra el inicio cargando la SD, y la Figura (2.4b) el inicio normal.

2. mientras se flashea la placa, se puede ir instalando el MDT en el ordenador host. MDT es un paquete de Debian, por lo que se instala lanzando el siguiente comando desde la terminal Bash⁽⁴⁾: `python3 -m pip install mendel-development-tool`. Después se deben ejecutar los comandos `alias python3='winpty python3.exe'` y `alias mdt='winpty mdt'` para configurar el path en Windows 10. Los comandos esenciales que se han utilizado del MDT son⁽⁵⁾
 - *MDT devices*: devuelve una lista de dispositivos y direcciones IP en la red local
 - *MDT wait-for-device*: espera a que se descubra un dispositivo en la red
 - *MDT shell [device]*: abre una shell interactiva en el dispositivo por defecto o el indicado
 - *MDT push <path or file> [remote path]*: crea una copia en la placa de los archivos o directorios locales indicados
 - *MDT pull <remote path or file> <local path>*: crea copias en local de los archivos o directorios de la placa indicados
 - *MDT genkey*: genera una clave SSH y la guarda en el ordenador local
 - *MDT pushkey <public key>*: copia la clave pública en la placa, para poder acceder por SSH sin necesitar el cable conectado al ordenador host
 - *MDT reboot*: reinicia el dispositivo
 - *MDT version*: muestra la versión del MDT
3. se establece la conexión con la shell de la placa utilizando MDT. Primero se verifica que el dispositivo se detecta por medio del comando *MDT wait-for-device*, y después se establece la conexión por medio del comando *MDT shell*. Si no aparece la placa porque no se detecta, hay que ir al “Panel de Control” → “Redes e Internet” → “Redes y Dispositivos Compartidos”, y ahí debería haber una red que no se identifica. Haciendo click en ella y yendo a “Propiedades” → “Compartir”, se debe marcar la opción que dice “Permitir que otros usuarios se conecten a través de este ordenador”
4. se establece la conexión a Internet, ejecutando este comando con las opciones propias del Wi-Fi escogido: `nmcli dev wifi connect <nombre de la red> password <contraseña> ifname wlan0`. Se puede verificar la conexión con el comando `nmcli connection show`
5. actualización del software de Mendel, ejecutando los comandos de Debian `sudo apt-get update` y `sudo apt-get dist-upgrade`. También se pueden actualizar el resto de paquetes con el comando `sudo apt-get upgrade`

Si se han realizado los pasos correctamente, ya se dispone de la placa totalmente instalada y funcional. Conectándola a la alimentación, y conectándose a ella por medio

⁽⁴⁾Este comando se ha adaptado ligeramente a Windows 10 para que no aparezcan posteriormente fallos sobre el path. Para ver los comandos originales se recomienda seguir la guía general de la página oficial de Coral [13].

⁽⁵⁾Se puede consultar una lista completa de los comandos que ofrece el MDT en la página oficial de Coral [13].

del cable USB-C a USB-A o utilizando el protocolo SSH, podemos interaccionar con ella y ejecutar los modelos que queramos. En la página oficial se ofrecen ejemplos de modelos pre-instalados en la imagen que se pueden ejecutar directamente. Para desconectar y apagar la placa de manera segura, se debe ejecutar el comando ***sudo shutdown now*** desde la shell de la placa.

El siguiente y último paso sería copiar a la placa los modelos y todo lo que haga falta (datos, etiquetas, etc.), y ejecutar desde su shell un script que realice todo lo necesario para leer y procesar los datos, y que llame a los modelos para llevar a cabo la inferencia. Durante la preparación que se llevó a cabo para poder realizar inferencia con los modelos generados, se encontraron un par de inconvenientes importantes. Por un lado, se es muy dependiente de las actualizaciones que se hagan del software de la placa, ya que hay herramientas y paquetes que se pueden quedar obsoletas. Y siguiendo en esta misma línea, no es posible actualizar Python ni descargarse las últimas versiones de los paquetes y librerías por fallos e incompatibilidades en las dependencias. Esto puede suponer un gran problema, ya que es posible que no todas las herramientas que se hayan utilizado fuera de la placa se puedan utilizar en ella, limitando mucho el procesado que se pueda hacer de los datos. Para solucionar este problema se pueden investigar combinaciones de versiones antiguas que sí que funcionen o, en el peor de los casos, reprogramar de una manera más simple y utilizando solo los paquetes disponibles las funcionalidades que se necesiten. En cualquier caso, se dispone de una placa totalmente funcional y lista para ejecutar en la TPU los modelos que se hayan compilado correctamente para ello.

2.2. Conjuntos de datos y tratamiento

Una vez presentada la placa Coral Dev, pasamos a describir los conjuntos de datos que se han utilizado a lo largo del trabajo, así como el tratamiento y preprocesado que han recibido para transformarlos en información útil y específica con la que entrenar los diferentes modelos.

Como ya hemos comentado, el trabajo tiene 2 partes: en la primera el objetivo es ser capaz de diseñar y entrenar diferentes modelos basados en redes neuronales simples que sean capaces de llevar a cabo reconocimiento de lenguaje para clasificar diferentes frases de activación. En la segunda parte seguimos con un enfoque similar, pero en lugar de diseñar y entrenar los modelos desde 0, aplicamos técnicas de transfer learning sobre un modelo mucho más complejo que previamente se ha entrenado con un conjunto de datos similar al que utilizamos en la primera parte, para ver si el modelo es capaz de aprender a llevar a cabo reconocimiento de lenguaje para clasificar las frases de activación, pero en este caso en un idioma diferente al idioma de entrenamiento de origen. Para llevar a cabo estas dos tareas han sido necesarios dos conjuntos de datos diferentes. Cada uno de ellos presenta una volumetría concreta, acorde a lo que es necesario para el correcto entrenamiento del modelo correspondiente. Además, cada conjunto de datos ha necesitado un diferente tratamiento para hacerlos apropiados para los modelos. Detallamos a continuación todo lo relacionado con estos conjuntos de datos utilizados.

2.2.1. Conjunto de datos para el entrenamiento de varias redes neuronales sencillas

El conjunto de datos utilizado para entrenar diferentes modelos basados en redes neuronales sencillas, con el objetivo llevar a cabo reconocimiento de lenguaje por medio de la clasificación de palabras de activación, es el conjunto de datos conocido como *speech commands* [15]. Este conjunto de datos es muy particular y es idóneo para entrenar modelos que detecten palabras o frases de activación, ya que a diferencia de otros conjuntos de datos para el reconocimiento de lenguaje, no dispone de audios de media o larga duración con su correspondiente transcripción, si no que los audios son breves y contienen exactamente las palabras que se busca detectar. Esto es muy importante porque un modelo de reconocimiento de lenguaje que busque detectar palabras de activación debe tener en cuenta que la unidad a reconocer es una única palabra (o conjunto de palabras) pero no una frase completa.

El conjunto de datos *speech commands* busca cubrir esas necesidades específicas para que el entrenamiento del modelo sea lo más acertado posible. Del mismo modo, la recolección de los datos que se hizo para componer el conjunto de datos buscaba ser lo más realista posible, por lo que muchas muestras presentan ruido de fondo y en general no están grabadas con la mejor calidad, ya que se obtuvieron desde los micrófonos de portátiles y móviles de las personas que decidieron participar en la recolección de los datos. Esto hace que el futuro modelo entrenado pueda estar más preparado para situaciones cotidianas de ruido, por ejemplo.

A efectos prácticos, el conjunto de datos final contiene más de 100 000 audios de 1 segundo de duración, de 35 palabras en inglés, recogidos de más de 2500 participantes. Sin embargo, esto es para la versión 2 del conjunto de datos. En nuestro caso hemos utilizado la versión 1, una versión un poco más reducida, con menos muestras y menos palabras, y con una calidad inferior. Esto ha sido así debido a limitaciones en memoria y potencia a la hora de procesar y utilizar los audios. Finalmente, los datos utilizados en el trabajo han sido un subconjunto de los datos de la versión 1 original, que contiene audios de las siguientes palabras, todas ellas en inglés:⁽⁶⁾

- números del 0 al 9
- on, off
- up, down, left, right
- yes, no

Tratamiento y exploración de los datos

Una vez teníamos el conjunto de datos, se realizó el tratamiento previo para verificar que todos presentan las mismas características básicas. Para ello, se procesaron todos los audios y se analizaron el número de canales, la profundidad de bits, el número de frames por segundo o frecuencia de muestreo y el número de frames totales.

⁽⁶⁾La selección final de las palabras se ha hecho de modo que el modelo final pueda detectar palabras que se puedan utilizar en una gran variedad de casos y aplicaciones diferentes.

Con respecto al número de canales, se verificó que todos los audios están en mono, es decir, tienen un único canal. Del mismo modo, todos los audios presentaban una profundidad de 2 bytes y una frecuencia de muestreo de 16 kHz. Hasta aquí todo era lo esperado, ya que era lo que se había detallado en la descripción del conjunto de datos. Sin embargo, al analizar el número de frames totales, se obtuvo que 3997 audios no tenían el número de frames esperado, lo que se traduce en una duración diferente a la esperada. Esto es crítico, ya que si la duración de los audios no está estandarizada, las características que se extraigan de ellos no serán homogéneas y los modelos no podrán ser entrenados correctamente.

Analizando más en profundidad los casi 4000 audios, vimos que todos ellos tenían una duración menor de 1 segundo. Este era el mejor de los dos casos posibles, ya que no requería tener que recortar el audio, con la consiguiente posible pérdida de información relevante. Justo al contrario, se hizo padding a los audios añadiendo fragmentos de silencio a los audios cortos hasta que la duración de todos ellos fuese la misma. Sin embargo, antes de llevar esto a cabo, se analizaron individualmente los 19 audios que presentaban una duración menor que la mitad de la duración esperada, por ser una duración demasiado corta. De esos 19 audios, solo 7 tuvieron que ser descartados por no entenderse la palabra que contenían. Pero esto nos ayudó a confirmar que audios con una duración menor de la esperada no implicaba que los audios fuesen incorrectos o no válidos.

Una vez teníamos todos los audios con las mismas características y duración, se pasó a analizar otra posible fuente de error más problemática que se identificó durante el proceso de homogeneización. En este caso, el problema estaba relacionado con el volumen y la intensidad del sonido grabado. Mostramos en la Figura (2.5) la muestra particular que nos hizo darnos cuenta de la posible fuente de error.

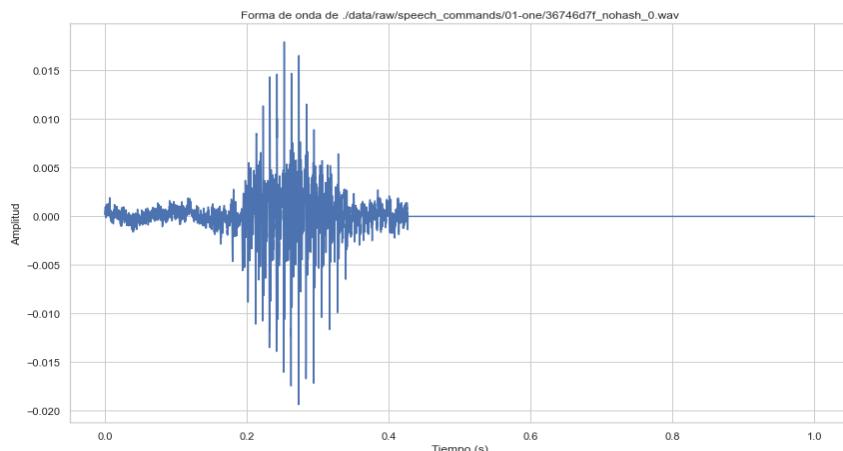


Figura 2.5. Ejemplo de una muestra aparentemente válida que presenta una amplitud de sonido extremadamente baja.

Aparentemente, si nos fijamos en la forma de la onda no percibimos nada extraño. Pero si nos fijamos en la escala en la que están las amplitudes podemos intuir el posible error, que se confirma una vez reproducimos el audio: el sonido es demasiado bajo. Tanto, que apenas se puede distinguir la palabra. Esto se puede traducir en muestras erróneas en las que el sonido de fondo se confunda con la propia frase de activación, generando malos resultados.

Como no era viable realizar un análisis individual de todas las muestras, se llevó a cabo un simple análisis estadístico de la amplitud de onda de los audios con el objetivo de identificar posibles outliers o patrones inesperados. Para ello, se agruparon las muestras por etiquetas, ya que determinados sonidos pueden aumentar de manera sistemática la amplitud de la onda, y se analizaron las amplitudes medias, mínimas y máximas de cada tipo de audio. Para los 3 casos se calculó el valor absoluto de los frames y poder así fijar un límite arbitrario pero adecuado de validez, según la distribución obtenida en los datos. De este modo, se analizó la amplitud media absoluta, la amplitud máxima absoluta y la amplitud mínima absoluta, pero en el caso de la amplitud media también se estudió sin aplicar el valor absoluto en cada frame. Mostramos en la Figura (2.6) el resumen de este análisis estadístico plasmado gráficamente como los boxplots con la distribución de los puntos de cada muestra de cada etiqueta.

Del gráfico de boxplots de la media de las amplitudes no se obtuvo información relevante. La amplitud media es 0 en la gran mayoría los casos, como se debería esperar, ya que las amplitudes positivas deberían cancelarse con las negativas. Sí que se detectó algún posible outlier en el caso en el que la amplitud media está muy alejada de 0, pero después de una investigación individual se confirmó si el audio era válido o no. La gran mayoría de audios catalogados como posibles outliers sí que eran válidos, y dicho valor de la media se debía únicamente a que la forma de la onda está desplazada en el eje y/o que la forma de la onda era extraña, pero el audio seguía siendo válido. Mostramos en la Figura (2.7) dos ejemplos de audios que presentaban una amplitud media muy diferente a la esperada. La de la Figura (2.7a) resultó ser no válida mientras que la de la Figura (2.7b) solo presentaba la amplitud desplazada, pero sí que era válida.

Del gráfico de boxplots de la media absoluta de las amplitudes tampoco se extrajo mucha información relevante. Como era de esperar, la amplitud media había aumentado un poco, pero seguía manteniéndose en límites válidos que no presentaban indicios de error. Se inspeccionaron algunos audios que presentaban amplitudes medias un poco más elevadas, pero se verificó que todos ellos eran válidos aunque la calidad de los audios en algunos casos presentaba cierta distorsión o se percibía cierto ruido constante de fondo. En la Figura (2.8) vemos un ejemplo de onda con una media absoluta muy elevada, que se corresponde con una muestra muy saturada pero en la que la palabra se distingue sin problemas.

Del gráfico de los boxplots de los máximos absolutos sí que pudo extraer información importante y útil para la detección de audios no válidos de una forma sistemática. Intuitivamente se habría esperado que la amplitud máxima detectada fuese un valor alejado de cero. Sin embargo, se observó que la distribución de los valores máximos absolutos era bastante homogénea entre 0 y 1. Los valores superiores no suponían ningún problema, ya que era entendible que algún frame del audio presentase un amplitud elevada. En cambio, los valores muy de amplitudes máximas absolutas próximos a 0 podían indicar que el audio no presentaba sonido o que era imperceptible lo que se estaba diciendo en él. Se inspeccionaron los resultados de los audios por etiqueta, y se fijó un límite inferior a partir del que el valor máximo absoluto implicaba una muestra errónea (viendo que la secuencia se repetía, se decidió fijar un límite común en una amplitud máxima de 0.06). De manera general se permitieron audios que tenían una intensidad baja, pensando que el modelo de reconocimiento de lenguaje por clasificación de palabras

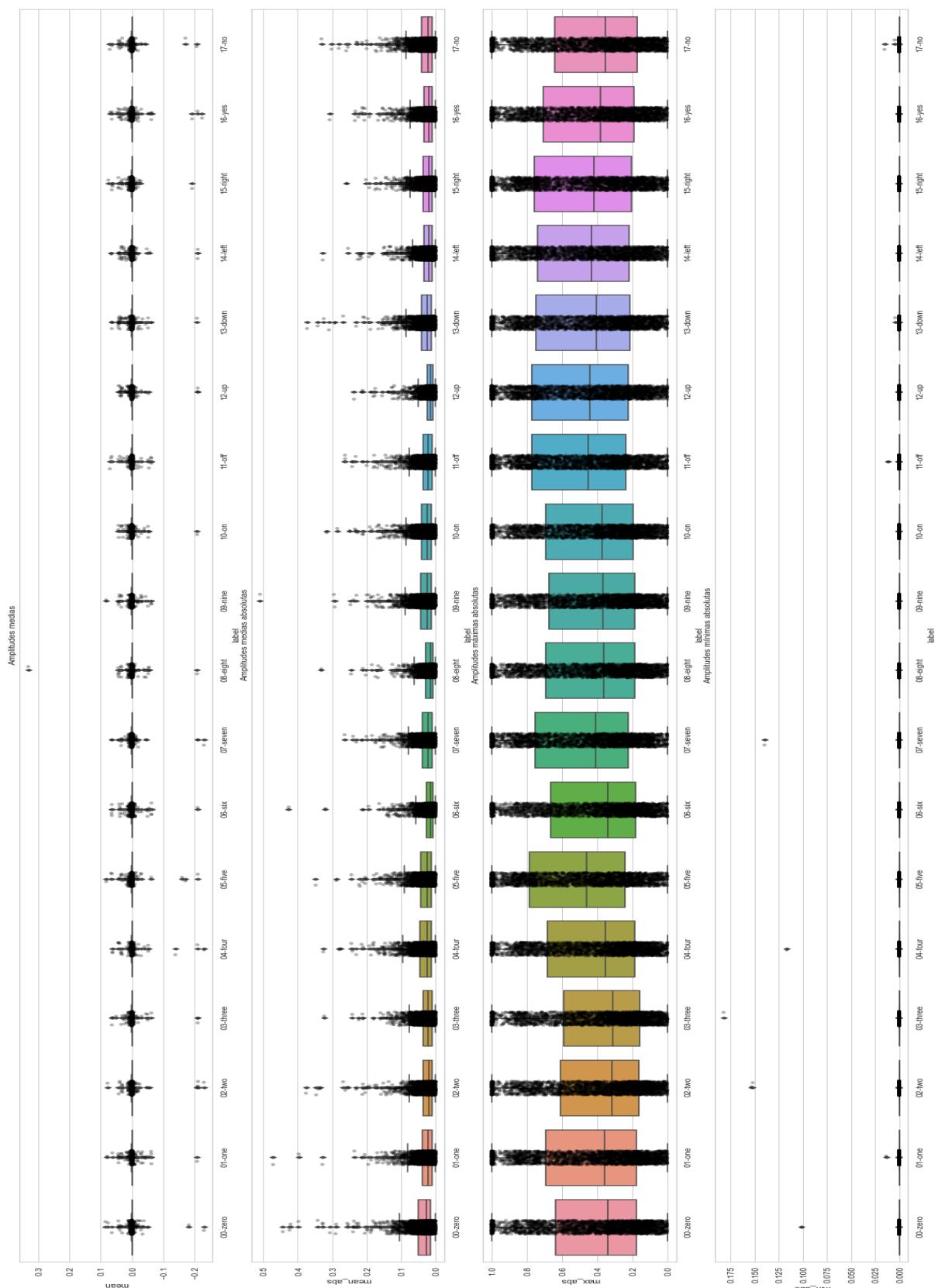


Figura 2.6. Resumen del análisis estadístico de las amplitudes de los audios. Por orden, se muestran los boxplots con la distribución de los puntos de las amplitudes medias, amplitudes medias absolutas, amplitudes máximas absolutas y amplitudes mínimas absolutas, agrupadas por etiquetas.

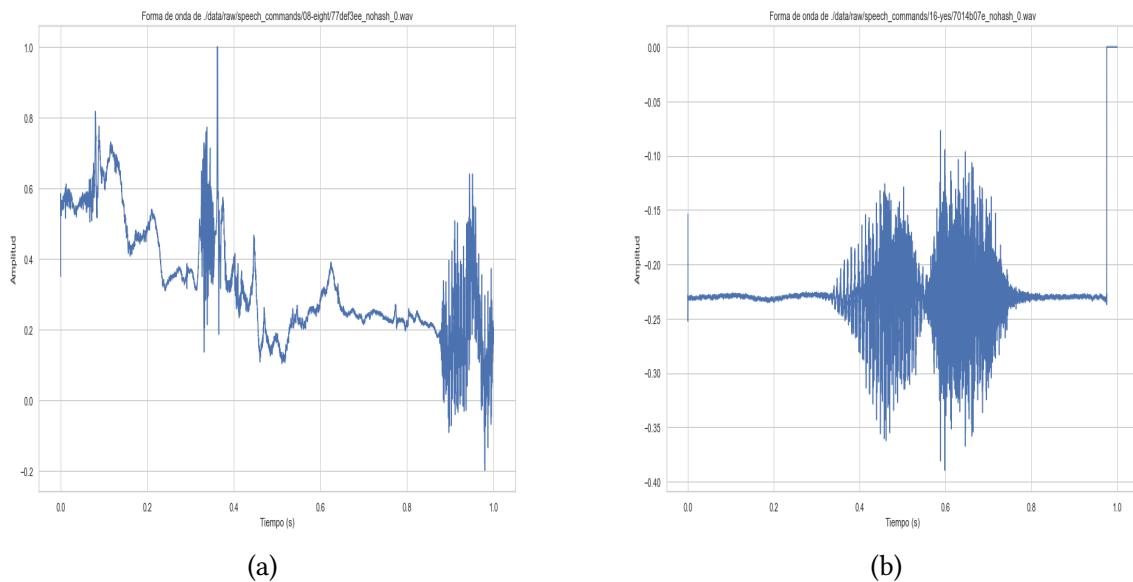


Figura 2.7. Ejemplo de muestras que presentan una amplitud media muy diferente a la esperada. La de la Figura (2.7a) resultó ser no válida mientras que la de la Figura (2.7b) solo presentaba la amplitud desplazada, pero sí que era válida.

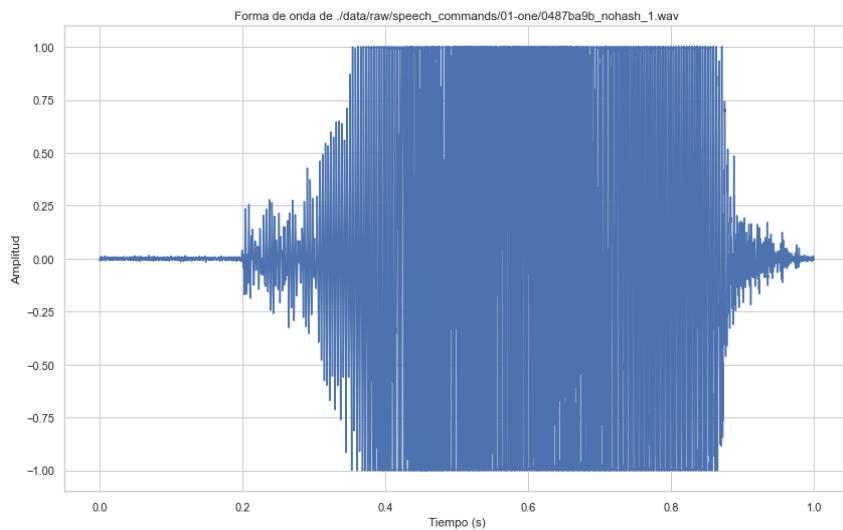


Figura 2.8. Ejemplo de un audio con una amplitud media absoluta muy elevada, que se corresponde con una muestra saturada pero válida, porque la palabra se distingue sin problemas.

de activación también debería detectar palabras con un volumen reducido (por ejemplo si la persona está lejos del dispositivo de captura de audio), pero era necesario fijar un límite para evitar muestras inválidas. Siguiendo esta idea, se eliminaron una gran cantidad de muestras por etiqueta.

Por último, del gráfico de los boxplots de los mínimos absolutos tampoco se obtuvo información relevante. En prácticamente todos los casos el mínimo era 0, y los valores diferentes eran prácticamente 0 también.

Data augmentation

Si recordamos, el conjunto de datos que se escogió para entrenar los modelos simples era el de la versión 1, una versión reducida del conjunto completo. Además de esto, de entre todas las palabras disponibles se hizo una selección de un subconjunto final de palabras. Y si a todo esto le añadimos que al realizar la limpieza del conjunto de datos se eliminaron un número considerable de muestras por etiqueta, es posible que el conjunto de datos en este momento no sea suficiente para entrenar los modelos decentemente.

Por ello, se decidió ampliar el conjunto de datos llevando a cabo un procedimiento sencillo de data augmentation. El data augmentation es un proceso que consiste en generar artificialmente nuevas muestras a partir de los datos reales con el objetivo de abarcar una mayor parte del espacio del problema, de modo que el modelo creado se entrene con un mayor número de muestras. Por ejemplo, en el caso del lenguaje hablado el data augmentation ha proporcionado buenos resultados para mejorar el reconocimiento de lenguaje con acento extranjero [16], por lo que además de ampliar el conjunto de datos puede enriquecer el resultado final. Sin embargo, son necesarios un par de apuntes sobre esta técnica:

- primero, las nuevas muestras que genere este procedimiento deberán pertenecer al conjunto de entrenamiento. No tiene sentido que las nuevas muestras pertenezcan al conjunto de validación o de test porque el objetivo principal es expandir el espacio de las muestras para que el modelo generalice mejor, no para testear sobre dichas nuevas muestras.
- segundo, hay que tener en cuenta que el data augmentation puede servir para mejorar los resultados del modelo que se entrena, pero siempre se conseguirá un resultado más generalizado si el conjunto de datos inicial es más extenso y diverso. En nuestro caso disponemos de un conjunto de datos limitado, así que vamos a aplicar data augmentation para intentar equilibrar el haber eliminado datos erróneos.

Los procesos de data augmentation se pueden aplicar prácticamente en cualquier conjunto de datos, pero se llevarán a cabo de una forma o de otra dependiendo de la naturaleza de los mismos. Particularizando en nuestro caso, para datos de audio podemos aplicar ciertas operaciones directamente sobre la forma de onda o sobre el espectro calculado previamente. Es importante remarcar que los datos aumentados deben ser “creíbles”, es decir, en el caso del audio los resultados deben ser realistas: por ejemplo, si el audio es de lenguaje hablado las frecuencias que lo formen deben estar en el espectro de la voz humana, por lo que no todas las transformaciones que se podrían aplicar son válidas. Algunas de las técnicas para aplicar data augmentation en datos de audio son:

- sobre la forma de onda:
 - desplazamiento temporal (*time shifting*): se desplaza la onda en el tiempo. En audios en los que la temporalidad es importante (por ejemplo en el caso del lenguaje) esta transformación no tiene sentido

- estiramiento/acortamiento temporal (*time stretching*): se modifica la velocidad del sonido sin cambiar el tono del audio
 - escalado del tono (*pitch scaling*): se modifican los tonos (frecuencias) del audio sin cambiar la velocidad del sonido
 - añadir ruido: se añade ruido blanco (o de otros tipos) de fondo
 - *impulse response addition*: se añaden reverberaciones a los datos (bueno para generalizar la detección de audio en diferentes medios o espacios)
 - filtrados (*low/high pass-band*): se aplican filtros para eliminar ciertas frecuencias
 - inversión de la polaridad: se invierte la forma de la onda
 - ganancia aleatoria: se aumentan las amplitudes para simular una mayor intensidad del sonido
- sobre el espectrograma: [17]
 - *time masking*: se enmascaran ciertas franjas temporales en el espectrograma (muy útil para generalizar el caso de que el audio llegue cortado o con interferencias)
 - *frequency masking*: se enmascaran ciertas frecuencias en determinadas franjas temporales
 - estiramiento/acortamiento temporal (*time stretching*)
 - escalado del tono (*pitch scaling*)

En nuestro caso se decidió llevar a cabo el data augmentation aplicando algunas de las técnicas directamente sobre la forma de la onda⁽⁷⁾. En primer lugar, se aplicó una ganancia aleatoria sobre las muestras que previamente habíamos detectado que tenían una baja amplitud máxima absoluta y que habíamos conservado. Concretamente, se aplicó sobre todas las muestras que tenían una amplitud máxima inferior a 0.15. En la Figura (2.9) se muestra un ejemplo de cómo la ganancia aleatoria ha generado una nueva muestra.

Posteriormente, se añadió ruido blanco a las muestras cuyo valor de la amplitud máxima absoluta fuese elevado. Concretamente, se aplicó sobre todas aquellas muestras cuyo valor máximo absoluto de la amplitud fuese superior a 0.75. En la Figura (2.10) se muestra un ejemplo de cómo el ruido blanco ha generado una nueva muestra.

Por último, se aplicó un escalado del tono sobre algunas muestras aleatorias de cada etiqueta. En este caso no mostramos ninguna imagen comparativa ya que visualmente la diferencia entre los dos espectros generados para comprobarlo no es apreciable. Aún así, sí que se verificó que la muestra generada era diferente de la original, escuchando ambos audios.

Con los datos analizados, tratados y aumentados ya disponemos del conjunto final que deberemos transformar en información útil con la que entrenar los modelos neuronales. En la Figura (2.11) se muestra cómo queda finalmente la distribución de los datos agrupados por etiquetas, donde se ve claramente que tenemos un conjunto altamente balanceado y válido, Enriquecido y listo para ser procesado.

⁽⁷⁾En todo momento se mantuvo orden y coherencia con las nuevas muestras generadas, para asegurar que estas estuviesen posteriormente en el conjunto de entrenamiento exclusivamente.

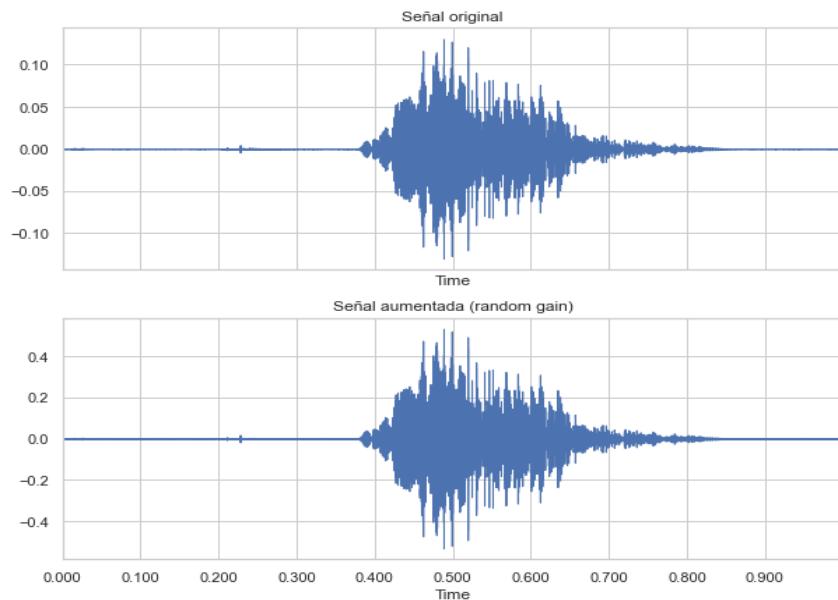


Figura 2.9. Ejemplo de una muestra sobre la que se ha aplicado ganancia aleatoria. Se ve cómo la escala del eje y ha aumentado en la segunda imagen.

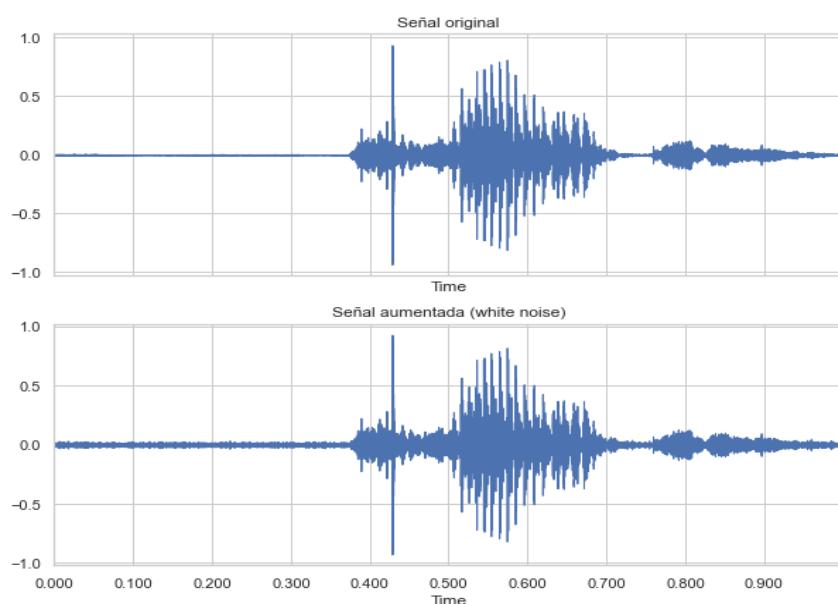


Figura 2.10. Ejemplo de una muestra sobre la que se ha añadido ruido blanco. El ruido blanco se percibe sin problemas en las zonas silenciosas.

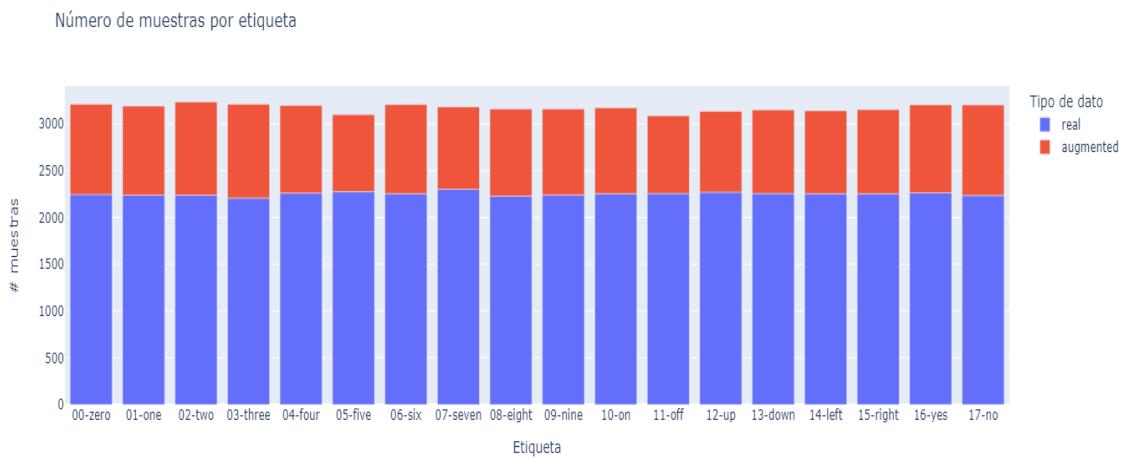


Figura 2.11. Distribución final de los datos, agrupados por etiquetas. Se ve que se dispone de un conjunto de datos balanceado y enriquecido, listo para ser procesado.

En el Anexo (A1) se muestran algunos ejemplos de las formas de onda y los espectrogramas de Mel de varios audios de cada una de las etiquetas, generados a partir del conjunto de datos final, totalmente analizado y preparado.

Transformación en información útil para los modelos neuronales

Con el tratamiento y análisis que se ha realizado hasta ahora hemos conseguido homogeneizar, limpiar y ampliar el conjunto inicial. Sin embargo, debido a cómo hemos decidido abordar el problema, no podemos introducir los datos de los audios directamente en los modelos para entrenarlos: necesitamos convertirlos en información útil.

Lo comentaremos con mayor detalle en la sección de los modelos pero, de manera muy resumida, se decidió llevar a cabo el entrenamiento de 3 tipos de modelos diferentes, dependiendo de la transformación que se hizo sobre el conjunto final de los datos. Cada una de las transformaciones buscaba centrarse en una característica diferente de los audios, con el objetivo de estudiar si un preprocesado determinado proporcionaba mejores resultados en el reconocimiento de lenguaje por medio de la detección de palabras de activación, con modelos más o menos sencillos.

Las transformaciones de los datos se llevaron a cabo utilizando la librería de Python Librosa [18], una librería muy completa para el análisis y el procesado de audio⁽⁸⁾.

⁽⁸⁾Hasta este momento se había decidido procesar los datos manteniendo sus características originales con el objetivo de estandarizar y verificar que todos los audios eran correctos y válidos. A partir de ahora, siempre que utilicemos la librería Librosa vamos a forzar la frecuencia de muestreo a 22050 Hz (frente a los 16 kHz originales), que es la frecuencia de muestreo con la que trabaja por defecto la librería. De este modo, obtendremos la información de una manera totalmente adecuada y homogénea, ya que los parámetros para llevar a cabo los cálculos en la librería están optimizados para esta frecuencia de muestreo, sin aumentar en exceso el tamaño de las muestras o los resultados. Más información sobre esto puede encontrarse en [este enlace](#), donde los autores del paquete explican los motivos por los que fijan la frecuencia de muestreo por defecto a 22050 Hz.

Concretamente, las transformaciones que se llevaron a cabo fueron las siguientes:

- en primer lugar, se extrajeron de los audios sus correspondientes spectrogramas de Mel que, como se ha explicado en la introducción teórica, son una representación particular del spectrograma clásico en la escala de Mel, una escala linealizada para el espectro auditivo humano. Se considera que estos spectrogramas son uno de los mejores tipos de información que proporcionar a un modelo basado en redes neuronales para el reconocimiento de lenguaje, ya que al final transformamos los audios en imágenes sobre las que aplicar redes convolucionales
- en segundo lugar, se extrajeron los MFCCs (coeficientes cepstrales en las frecuencias de Mel), otro tipo de información intrínseca a los audios que se había utilizado a lo largo de la historia para el reconocimiento del lenguaje ya que, debido a su justificación y demostración teórica, mapean la información del audio directamente con el timbre de la voz, que se puede entender como la característica diferenciadora de los diferentes fonemas del lenguaje
- por último, se mantuvo la información cruda de la señal de audio sin centrarse en ninguna característica en particular. Con este planteamiento se busca que sea la propia red neuronal la que se encargue de encontrar los mejores patrones y la información intrínseca de los audios para llevar a cabo la detección y clasificación de las palabras de activación. Como hemos comentado previamente, este parece ser el enfoque que se está explotando actualmente en el estado del arte del reconocimiento de lenguaje, ya que es el método que menos procesamiento previo requiere, con las implicaciones y la mejora en la velocidad que ello supone

Para la extracción de los spectrogramas de Mel se fijó en 80 el número de bandas de Mel a obtener de cada audio. El tamaño de la ventana sobre la que se aplicaba la FFT era de 2048 frames y la distancia en frames entre cada ventana era de 512 frames (estos valores son los que Librosa establece por defecto para una frecuencia de muestreo de 22050 Hz). De este modo, obtenemos imágenes de los spectrogramas con una dimensión de 44x80, siendo 44 el número de segmentos temporales resultantes y 80 el número de bandas de Mel generadas. En el Anexo (A1) podemos ver ejemplos de spectrogramas de Mel generados para diversas muestras aleatorias de cada una de las etiquetas. En la Figura (2.12) mostramos un caso concreto de spectrogramas de Mel generados, donde claramente se aprecian diferencias muy notables para muestras de etiquetas diferentes mientras que los spectrogramas son similares para las muestras de la misma etiqueta.

Para los MFCCs se siguió un planteamiento similar. Se mantuvieron los mismos valores para el tamaño de la ventana y la distancia en frames entre cada ventana, pero en este caso en lugar de fijar las bandas de Mel se fijaba el número de coeficientes a extraer de cada segmento temporal. Como se ha comentado en la introducción teórica, de manera general pueden extraerse hasta 39 coeficientes por cada segmento temporal. Sin embargo, para tareas de reconocimiento de lenguaje suelen utilizarse entre 12 y 20 coeficientes ya que estos contienen la gran mayoría de información relevante para el lenguaje. En nuestro caso, se fijó a 12 el número de coeficientes que se extrajeron de cada segmento temporal para mantener al mínimo la complejidad y la volumetría de datos a entrenar

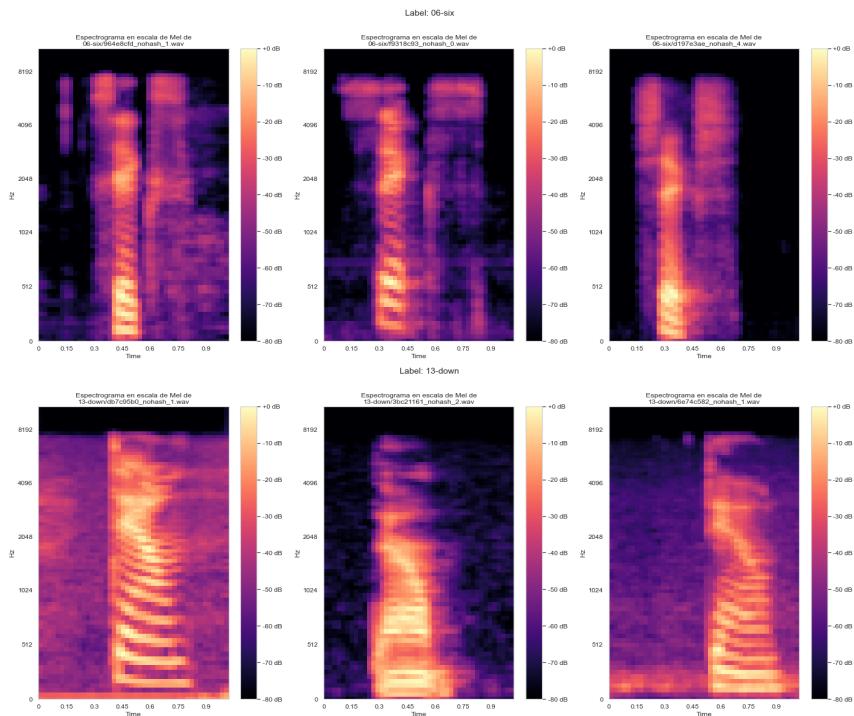


Figura 2.12. Ejemplo de espectrogramas de Mel generados para 3 muestras aleatorias de las etiquetas “six” (arriba) y “down” (abajo). Se aprecian semejanzas entre los espectrogramas de las mismas etiquetas, y claras diferencias entre las muestras de etiquetas diferentes.

posteriormente, sin empeorar a priori la bondad de los resultados obtenidos. De este modo, al final cada audio se traduce en una matriz de 44x12, siendo 44 el número de segmentos temporales resultantes y 12 el número de MFCCs extraídos en cada segmento. Aunque estos coeficientes se pueden considerar características individuales, también podemos entenderlos como si se tratase de una variación de un espectrograma, por lo que podemos visualizarlos gráficamente. En la Figura (2.13) mostramos un ejemplo concreto de los gráficos generados con los MFCCs para diferentes muestras de dos etiquetas diferentes. Se pueden apreciar similitudes entre las muestras de las mismas etiquetas y claras diferencias si se comparan las muestras de las diferentes etiquetas.

Por último, para el caso de la señal cruda de audio no fue necesario realizar ninguna transformación en particular. Lo único que se hizo fue guardar la propia señal reduciendo mucho la frecuencia de muestreo para poder para que la información resultante ocupase mucho menos espacio y facilitase el entrenamiento del modelo y en general la gestión de los datos. Concretamente, se redujo la frecuencia de muestreo a una 5^a parte de la original: se estaba utilizando la frecuencia que utiliza Librosa por defecto (22050 Hz), y se redujo a 4410 Hz.

En cada caso, una vez llevada a cabo la transformación de los datos, la información se guardó apropiadamente en archivos JSON diferentes (separando claramente los audios originales y las muestras generadas por data augmentation) junto con las etiquetas de los audios, para que no fuese necesario repetir el largo proceso. Con todo esto, ya se disponía de los datos finales completamente procesados y transformados para entrenar los diferentes modelos.

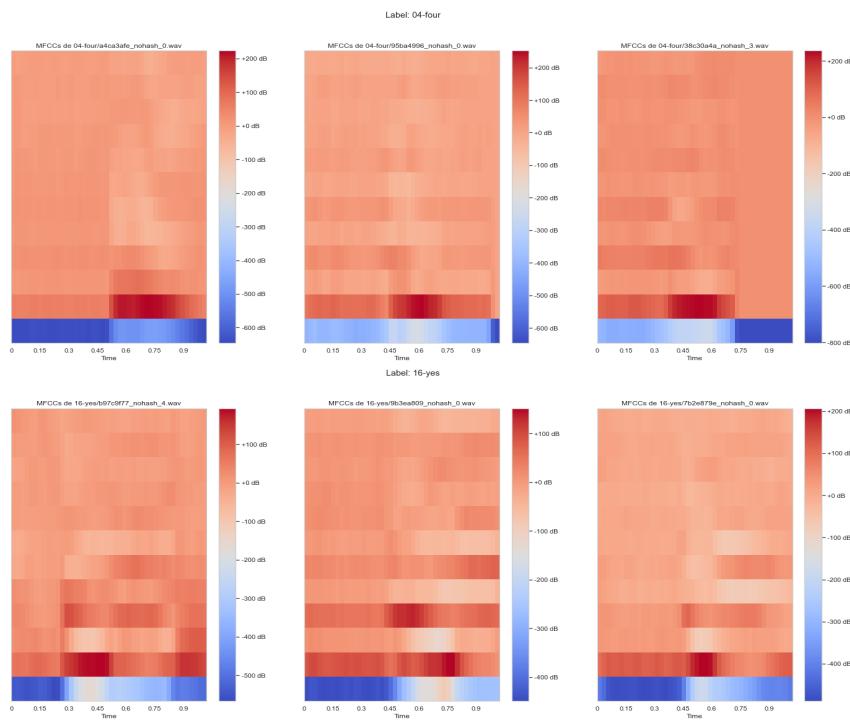


Figura 2.13. Ejemplo de representación gráfica de los MFCCs generados para 3 muestras aleatorias de las etiquetas “four” (arriba) y “yes” (abajo). Se aprecian semejanzas entre los gráficos de las mismas etiquetas, y ciertas diferencias entre las muestras de etiquetas diferentes.

2.2.2. Conjunto de datos para aplicar transfer learning sobre una red neuronal compleja

Para la segunda parte del trabajo, en la que aplicamos transfer learning sobre un modelo previamente entrenado con la versión 2 del conjunto de datos *speech commands* que hemos utilizado en la primera parte, era necesario un conjunto de datos que también tuviese audios cortos en los que se decían las palabras que se querían detectar en el modelo final. En un primer momento se buscaron conjuntos de datos en español que más o menos tuviesen las mismas características que el conjunto previo, de modo que se adaptasen a la tarea de reconocimiento de lenguaje por medio de la detección y clasificación de palabras de activación. Sin embargo, los conjuntos de datos para el reconocimiento de lenguaje en español no son tan abundantes, y los que hay no poseen las características para entrenar un modelo así (los que se encontraron fueron conjuntos de audios de media y larga duración en los que las etiquetas eran la transcripción de los propios audios).

Una de las grandes ventajas que ofrece el transfer learning es que, como el modelo ya ha sido previamente entrenado y lo que se busca es realizar *fine tuning* sobre unas pocas capas finales de la red, el conjunto de datos necesario no tiene que ser extremadamente grande. Por ello, en lugar de buscar conjuntos de datos en español se decidió generar un conjunto propio, recopilando audios de diferentes personas en los que se dijesen las palabras que se quieren detectar. Para ello, se solicitó a varias personas que enviasen por WhatsApp audios cortos (de entre 1 y 3 segundos) en los que dijesen individualmente las palabras a detectar, en un entorno no excesivamente ruidoso (la única condición era

que la palabra se pudiese escuchar sin problemas). Por seguir con la lógica del problema, las palabras escogidas fueron la traducción al español de las palabras utilizadas en el conjunto de la primera parte:

- números del 0 al 9
- aceptar, rechazar
- arriba, abajo, izquierda, derecha
- sí, no

Este planteamiento buscaba estudiar si el transfer learning permite reentrenar un modelo para que detecte las palabras en un lenguaje diferente. Se consiguieron recopilar audios de 102 personas diferentes⁽⁹⁾, todas ellas enviando un audio de cada una de las palabras, por lo que al final se obtuvo un conjunto de 1836 audios en español. No fue necesario aplicar técnicas de data augmentation en este caso, ya que el conjunto de datos recopilado presentaba un volumen suficiente.

Tratamiento y exploración de los datos

Durante la etapa de recolección de los datos en español cada uno de los audios recibidos se analizó de manera individual para verificar que era válido, por lo que no fue necesaria una exploración posterior. La tarea que sí que fue necesaria realizar fue la de estandarizar todos los audios, ya que no eran en absoluto homogéneos. Aunque todos presentaban las mismas características porque procedían de la misma fuente, WhatsApp, la duración de los audios era muy irregular, extendiéndose entre los 1 y los 3 segundos, bien por la longitud de la palabra o por la dificultad que supone grabar con el móvil de manera exacta. De manera general, el procesado que tuvieron los audios fue el siguiente:

1. lectura del archivo .ogg (propio de las grabaciones de WhatsApp) utilizando la frecuencia de muestreo por defecto de Librosa, 22050 Hz
2. estandarización individual de los audios para que todos ellos duren un segundo exacto (medido en número de frames), asegurándose de que dicho segundo se corresponde con el momento en el que se dice la palabra
3. exportación ordenada del audio final estandarizado, en formato .wav

El proceso de estandarización fue lo más complicado de llevar a cabo, ya que durante la recolección de los datos se pudo observar que no había un patrón concreto para las grabaciones. Más específicamente, se podían clasificar los audios en 3 perfiles de grabación: el primer grababa la palabra justo al principio del audio y luego dejaba un poco de silencio, el segundo dejaba un poco de silencio al principio y luego grababa la palabra justo al final, y el tercero que grababa la palabra en medio y dejaba silencio al

⁽⁹⁾Los audios se obtuvieron en su mayoría de personas entre los 18 y los 30 años (aunque con un cierto porcentaje de personas mayores de 30 años) y con una distribución más o menos del 50% entre audios obtenidos de hombres y mujeres. No se han obtenido audios ni de niños ni de personas mayores de 70 años.

principio y al final. Dándose cuenta de esto, no había forma sistematizada y directa de recortar la ventana concreta de 1 segundo de duración que contenía la palabra. Por ello, el proceso de estandarización comprendía dos funciones definidas por mí mismo, con la intención de extraer la mejor ventana exacta de un segundo de duración con la palabra:

- la primera función se programó por precaución. En principio WhatsApp no permite enviar audios inferiores a 1 segundo, pero por completitud y por contemplar todos los casos, esta función realizaba padding a los audios cortos, añadiendo silencio hasta completar el segundo
- la segunda función es la que se encargaba de tomar la mejor ventana de 1 segundo de duración. Por medio de la librería Librosa, para cada audio se iban analizando las sucesivas ventanas de 1 segundo de duración de manera ordenada, separadas una cantidad de frames igual a la 15^a parte de la frecuencia de muestreo ($22050/15 = 1470$ frames), extrayendo de cada ventana la métrica conocida como RMS, *root mean squared*, de todos los valores que conforman la ventana. Finalmente, se escogía la ventana que presentaba un mayor RMS, ya que como condición se había solicitado que el audio fuese en entornos no muy ruidosos, de modo que el mayor RMS se correspondería con la ventana donde se encontraba el habla humana

Una vez se tenían todos los audios estandarizados, se escogieron algunos de manera aleatoria de cada etiqueta para verificar que eran correctos. Todos ellos resultaron válidos aunque en algunos pocos casos, para las palabras de más de 2 sílabas, era posible que el inicio o el final de la palabra estuviese ligeramente recortado. Por último, se llevó a cabo una rápida inspección para verificar de nuevo que todos los audios presentaban las mismas características de número de canales (siendo todos mono), profundidad de bits (siendo todos 2 bytes), frecuencia de muestreo (siendo todos 22050 Hz) y número de frames (siendo todos 22050 frames, verificando que todos los audios duraban 1 segundo). No hubo ningún fallo ni nada fuera de lo esperado. Con todo esto, ya se disponía del conjunto final de datos para aplicar transfer learning, totalmente estandarizado y preparado. Mostramos en la Figura (2.14) la distribución final del conjunto de datos para el transfer learning, donde se ve que disponemos de un conjunto de datos perfectamente balanceado, con 102 muestras por etiqueta.

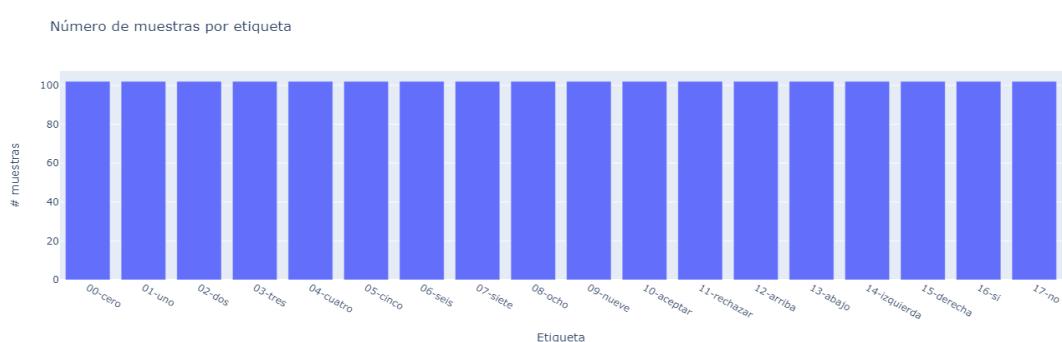


Figura 2.14. Distribución final de los datos para realizar transfer learning, agrupados por etiquetas. Se dispone de un conjunto de datos perfectamente balanceado, con 102 muestras por etiqueta.

2.3. Modelos

Para finalizar este capítulo, pasamos a explicar los diferentes modelos que se han desarrollado o utilizado para cada una de las partes del trabajo. En todos los casos se han utilizado redes neuronales ya que son los modelos que mejor permiten utilizar datos complejos, como lo son datos en forma de imagen o audios, y además permiten extraer información intrínseca de una manera más elaborada utilizando estructuras particulares destinadas a ello. Se ha dejado de lado la idea clásica de realizar el entrenamiento con un número reducido de variables físicas pre-generadas⁽¹⁰⁾ y en su lugar se opta por un enfoque moderno basado en deep learning y redes neuronales. Por otro lado, también se han utilizado exclusivamente redes neuronales desarrolladas con TensorFlow 2.0 porque, como ya hemos comentado, la TPU de la placa Coral Dev solo está preparada para ejecutar modelos de tipo TFLite (por lo que necesariamente deben ser generados con TensorFlow) que cumplan una serie de características impuestas por la propia placa.

De manera general, para cada uno de los casos explicaremos la estructura del modelo creado o descargado, y también explicaremos cómo se llevaron a cabo los procedimientos de conversión de los modelos a formato .tflite y los procesos de cuantización, así como la consiguiente preparación para la ejecución de los modelos en el la TPU de la placa. Por ello, para cada caso tendremos 3 o 4 modelos diferentes, aunque todos ellos tengan la misma estructura:

- un modelo normal creado con TensorFlow 2.0
- un modelo normal en formato .tflite preparado para ser ejecutado en la placa. Solo los modelos que tengan operadores compatibles (u operadores que se pueden transformar en operadores compatibles con la misma funcionalidad) se pueden convertir a este formato
- un modelo cuantizado y en formato .tflite preparado para ser ejecutado en la placa. En principio, si se dispone de una muestra amplia de los datos esperados por el modelo, cualquier modelo de tipo TFLite se puede cuantizar total o parcialmente
- un modelo cuantizado y en formato .tflite preparado para ser ejecutado en la TPU de la placa. En el caso del modelo complejo sobre el que se aplica transfer learning no dispondremos de este modelo ya que no cumple con los requisitos establecidos por la placa

2.3.1. Redes neuronales sencillas

Para la primera parte del trabajo se ha generado una red neuronal sencilla para cada tipo de representación de los datos. Cada una tiene una estructura particular obtenida por prueba y error, viendo cómo era la respuesta de la red en cada caso.

Como las redes se han ido generando y entrenado desde 0, el proceso de cuantización se ha podido llevar a cabo previamente en lo que se conoce como pre-cuantización del

⁽¹⁰⁾En cierto modo los MFCCs se pueden considerar variables físicas pre-generadas. Sin embargo no los consideramos así porque estas características no se extraen de manera particular por audio sino que al final se dispone de un conjunto extenso de dichas variables en cada muestra.

modelo, llevando a cabo un entrenamiento extra previo a la cuantización pero que está preparado para llevarla a cabo. Este procedimiento prepara el modelo para su posterior cuantización, haciendo que el modelo final cuantizado tenga un comportamiento muy similar al del modelo original (entendiendo comportamiento como bondad del modelo, medido con el accuracy). A continuación detallamos cada caso individual, con las peculiaridades encontradas en cada uno.

Red CNN para los datos de los espectrogramas de Mel

El primer modelo que se desarrolló fue el asociado a los datos presentados como espectrogramas de Mel. Los espectrogramas son imágenes, así que a priori una red neuronal basada en capas convolucionales 2D era la mejor forma de afrontar el problema ya que las capas convolucionales 2D están optimizadas para las imágenes (en cuanto a número de parámetros a entrenar) y por lo general ofrecen muy buenos resultados. Además, después de analizar los requisitos y limitaciones de la placa Coral Dev, se verificó que este tipo de capas se pueden ejecutar en la TPU de la placa.

Antes de definir el modelo se importaron los datos asociados a los espectrogramas de Mel con sus correspondientes etiquetas y se realizó la división en los conjuntos de entrenamiento, validación y test. A la hora de hacer esta división se tuvieron en cuenta las muestras que se habían generado por data augmentation, para que solo estuviesen presentes en el conjunto de entrenamiento. Aproximadamente un 30% de los datos conformaron el conjunto de test y un 10% el de validación. Estos porcentajes tan elevados no suponían un problema ya que se disponía de un conjunto grande de datos. Una vez estaban los datos divididos en los 3 conjuntos, se llevó a cabo un simple ajuste de dimensiones con el objetivo de que los datos se ajustaran a las dimensiones esperadas por la red convolucional. Inicialmente, los datos tenían una dimensión de $nx44x80$, siendo n el número de muestras en cada conjunto, 44 el número de segmentos temporales resultantes y 80 el número de bandas de Mel seleccionado previamente. A este tensor de 3 dimensiones se le tuvo que añadir una 4^a dimensión por el final que representase el número de canales que tenía la imagen. En nuestro caso, el número de canales era de 1, por lo que las dimensiones finales de los datos eran de $nx44x80x1$, cumpliendo las restricciones de la placa sobre las dimensiones de los tensores.

Con los datos disponibles, se procedió a crear la red basada en capas convolucionales. Después de ir probando muchas estructuras diferentes y diversos hiperparámetros de las capas convolucionales, la red que nos proporcionó mejores resultados fue una red de 3 capas convolucionales 2D con kernels de 3x3 y diversos filtros de salida, cada una de ellas seguida por una capa de max pooling 2D con kernels 2x2 y strides de 1x1. Cada capa convolucional 2D reducía a la mitad el número de filtros que generaba. Después de las capas convolucionales se utilizó una capa flatten para aplanar todos los datos, seguida de una capa densa con 18 neuronas (una para cada etiqueta) y función de activación softmax para que los resultados obtenidos de la última capa se puedan entender como las probabilidades de pertenecer a cada una de las etiquetas. En el Anexo (A2), en la Figura (A2.1a) se muestra un gráfico con la estructura de esta red.

Una vez se disponía de la estructura final del modelo, se procedió con su compilación,

entrenamiento y análisis de su bondad por medio del accuracy. También se preparó el proceso para realizar inferencia con este modelo. Después de todo esto, se pasó a la generación del modelo cuantizado. Debido a la estructura del modelo y al hecho de que se ha generado desde 0, se pudo llevar a cabo el proceso de pre-cuantización, realizando un entrenamiento extra que preparaba al modelo para ser cuantizado, utilizando funciones específicas y preparadas para generar modelos cuantizados. Finalizado dicho nuevo entrenamiento, se pasó a cuantizar como tal el modelo, transformándolo en un modelo TFLite totalmente preparado para ser ejecutado en la placa Coral Dev. Recordemos que hemos comentado que para que los modelo puedan estar ejecutados en la TPU de la placa, estos tienen que estar completamente cuantizados (es decir, se cuantizan tanto los pesos como las activaciones) y esta cuantización debe ser completamente a enteros. Para llevar a cabo este tipo de cuantización fue necesario utilizar datos del entrenamiento para generar un conjunto representativo que permita establecer los límites y las distancias entre enteros a la hora de cuantizar el modelo. En el Anexo (A2), en la Figura (A2.1b) se muestra un gráfico con la estructura del modelo cuantizado y convertido a TFLite, generado con la herramienta Netron [19], un programa de código abierto dedicado a la visualización de modelos aprendizaje máquina y deep learning.

Al modelo completamente cuantizado a enteros se le añadió un archivo extra con el conjunto de metadatos asociados a las etiquetas y parámetros importantes de los datos para que el modelo en sí estuviese completo. Seguidamente, se preparó otro proceso diferente para realizar inferencia sobre nuevos datos utilizando el modelo cuantizado, en cierto modo emulando su ejecución en la placa Coral Dev, ya que para ello fue necesario definir un intérprete del modelo, preparar y cuantizar correctamente los datos de entrada, preparar y asignar los tensores correctamente, e invocar al intérprete del modelo cuantizado para realizar la inferencia.

Por último, este modelo cuantizado en formato .tflite se compiló de nuevo utilizando el *Edge TPU Compiler* [20] en formato online, con el objetivo de preparar finalmente el modelo para que se pueda ejecutar en la TPU de la placa Coral Dev, generando así el modelo TFLite preparado para la TPU.

Red CNN para los datos de los MFCCs

El segundo modelo que se desarrolló fue el asociado a los datos presentados como MFCCs. En este caso se siguió un procedimiento completamente análogo al realizado previamente, para el caso de los espectrogramas de Mel. Como hemos mostrado, los MFCCs se pueden graficar y entender como imágenes, así que una red neuronal basada en capas convolucionales 2D vuelve a ser la mejor forma de enfocar el problema del diseño de la red.

Igual que se hizo en el caso de los espectrogramas de Mel, en primer lugar se importaron los datos, se hizo la división en los conjuntos de entrenamiento, validación y test siguiendo los mismos porcentajes y el mismo procedimiento para asegurar que las muestras aumentadas solo estaban en el conjunto de entrenamiento, y se llevó a cabo el ajuste de las dimensiones de los datos. En este caso, las dimensiones iniciales eran de $n \times 44 \times 12$, siendo n el número de muestras de cada conjunto, 44 el número de segmentos

temporales resultantes y 12 el número de MFCCs calculados. A este tensor se le añadió también una cuarta dimensión que representaba los canales, que en este caso era solo 1, igual que en el caso anterior. De este modo, las dimensiones finales de los datos eran de $nx44x12x1$.

Posteriormente se pasó a crear la red basada en capas convolucionales. De nuevo, el proceso se llevó a cabo por prueba y error, probando muchas estructuras diferentes y modificando los hiperparámetros de las capas. En este caso, la red que proporcionó unos mejores resultados fue de nuevo una red de 3 capas convolucionales 2D con kernels de 3x3 y diversos filtros de salida, cada una de ellas seguida por una capa de max pooling 2D con kernels 2x2 y strides de 1x1. Los filtros generados en cada capa convolucional no seguían un patrón claro. Después de estas capas convolucionales se utilizó una capa flatten para aplanar todos los datos, seguida de una capa densa con 25 neuronas, y finalmente seguida de otra capa densa con 18 neuronas (una para cada etiqueta) y función de activación softmax por el mismo motivo que en el caso de los espectrogramas de Mel. Vemos que la estructura de la red es muy similar a la del caso anterior, siendo la mayor diferencia el hecho de que se añadió una capa densa extra antes de la capa final de salida. En el Anexo (A2), en la Figura (A2.2a) se muestra un gráfico con la estructura de esta red.

Una vez se disponía de la estructura final del modelo, los pasos que se siguieron fueron exactamente los mismos que en el caso anterior. Primero se realizó la compilación, entrenamiento y análisis de la bondad por medio del accuracy, y se preparó un proceso para realizar inferencia. A continuación se pasó a la generación del modelo cuantizado, con todo idéntico al caso de los espectrogramas de Mel, a excepción de que el conjunto representativo de datos para llevar a cabo la cuantización se generó a partir del conjunto de entrenamiento de los MFCCs. Posteriormente se generó el modelo TFLite cuantizado final, que se compiló de nuevo para poder ser ejecutado en la TPU de la placa. De este modo, en este caso también se dispone de un modelo normal, un modelo cuantizado en formato .tflite y un modelo cuantizado y compilado para que se ejecute en la TPU. En el Anexo (A2), en la Figura (A2.2b) se muestra un gráfico con la estructura del modelo cuantizado y convertido a TFLite.

Red CNN para los datos crudos de la señal

Por último, se desarrolló el modelo asociado a los datos crudos de la señal. Este caso fue un poco particular, ya que en un primer momento la idea era desarrollar un modelo basado en redes densas con la posibilidad de incorporar alguna capa recurrente usando neuronas LSTM. Sin embargo, después de probar muchas estructuras diferentes se obtuvieron muy malos resultados: o bien el modelo sobreajustaba de manera exagerada con un accuracy en validación y test muy bajo, o bien todo el entrenamiento requería demasiado tiempo y en general los resultados no eran buenos. Por ello se decidió investigar otra forma de proceder, y se descubrió que una buena alternativa era la de aplicar de nuevo capas convolucionales, en este caso 1D [21]. Esto suponía un problema, porque las capas convolucionales 1D no aparecían en la lista de capas permitidas por la placa Coral Dev. Sin embargo, supusimos que si las capas 2D sí que eran válidas, de algún modo las capas 1D también lo serían o se podrían adaptar. Esto se verificó porque se encontraron casos en los que modelos con capas convolucionales 1D sí habían podido convertirse en

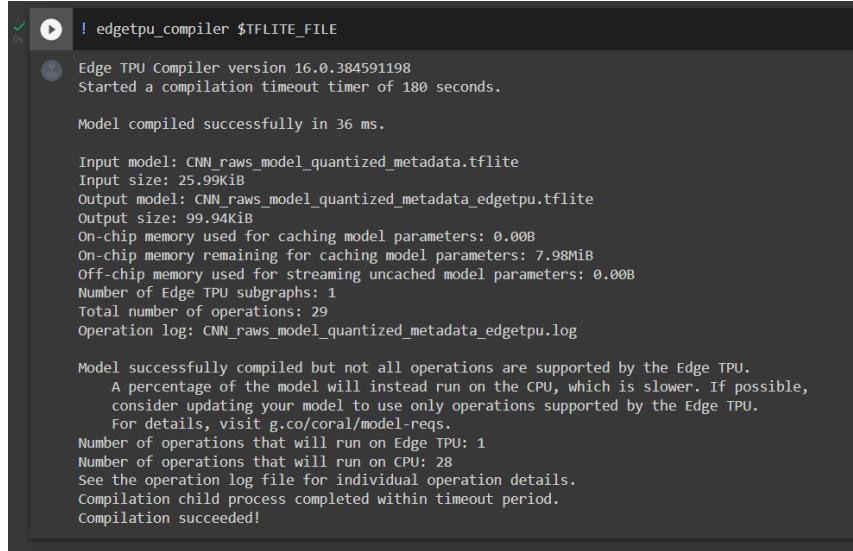
modelos TFLite.

Teniendo clara esta hipótesis, se pasó a proceder como en los casos anteriores. En primer lugar se importaron los datos, se dividieron en los conjuntos de entrenamiento, validación y test con los mismos porcentajes y manteniendo la coherencia con las muestras aumentadas. Y de nuevo, como la red iba a ser convolucional, se tuvieron que adaptar las dimensiones de los datos. Inicialmente las dimensiones eran de $nx4410$, siendo n el número de muestras para cada conjunto y 4410 el número de frames con información por audio. A este tensor se le añadió una tercera dimensión indicando el número de canales, que igual que en los casos anteriores era solo 1, por lo que las dimensiones finales de los datos eran de $nx4410x1$.

Con los datos preparados se pasó a crear la red basada en capas convolucionales 1D. Procediendo del mismo modo que con los espectrogramas de Mel y los MFCCs, se probaron diversas estructuras e hiperparámetros por prueba y error. En este caso, la red que ofreció unos mejores resultados era una red de 4 capas convolucionales 1D con kernels de diferentes tamaños y diferente número de filtros, seguidas por capas de max pooling 1D con kernels y strides de diferentes tamaños. Lo único a comentar aquí es que el tamaño de los kernels y las strides se iniciaron en valores superiores y fueron disminuyendo en cada capa. Los primeros valores de los kernels estaban calculados de modo que la capa extrajese teóricamente información relevante, ya que no todos los frames tienen valores útiles, así que un kernel de un tamaño adecuado podría condensar la información correspondiente a las mínimas unidades de audio que son percibidas por las personas (aproximadamente de unos 10 ms). Siguiendo a las capas convolucionales se utilizó una capa flatten para aplanar todos los datos, seguida de una capa densa con 30 neuronas y otra capa densa con 18 neuronas (una para cada etiqueta) y función de activación softmax, igual que en los otros casos. Por ello, para este enfoque disponemos de una red muy diferente a las anteriores, más compleja por tener una capa extra. En el Anexo (A2), en la Figura (A2.3a) se muestra un gráfico con la estructura de esta red.

Con la estructura del modelo fijada se pasó a su compilación, entrenamiento y verificación de su bondad por medio del accuracy. Del mismo modo, se preparó un proceso para realizar inferencia. Pero a la hora de intentar generar el nuevo modelo preparado para la posterior cuantización, surgió un error porque las capas convolucionales 1D y las capas max pooling 1D no estaban preparadas para ello. Debido a esto, en lugar de llevar a cabo una pre-cuantización por medio de un nuevo entrenamiento preparado para cuantizar como en los casos anteriores, se tuvo que llevar a cabo el proceso mediante una post-cuantización. El procedimiento es el mismo, pero se parte directamente del modelo normal, por lo que es posible que la cuantización modifique el comportamiento en el modelo final cuantizado. De este modo, se obtuvo un modelo final en formato .tflite cuantizado, utilizando el conjunto representativo correspondiente. Por el mismo motivo, a la hora de intentar compilar el modelo TFLite cuantizado para poder ser ejecutado en la TPU de la placa, apareció un aviso informando de que ciertas operaciones no iban a poder ser ejecutadas en ella, y que el resto de operaciones (a partir de la primera operación que no pueda ejecutarse en la TPU) se ejecutarían en la CPU. Mostramos en la Figura (2.15) este aviso. Debido a estos problemas, para este caso disponemos de un modelo normal, un modelo TFLite cuantizado y un modelo TFLite cuantizado pero que solo puede ejecutarse parcialmente en la TPU. En el Anexo (A2), en la Figura (A2.3b) se muestra

un gráfico con la estructura del modelo cuantizado y convertido a TFLite, donde se ve cómo las operaciones no permitidas se han transformado, generando un nuevo modelo aparentemente más complejo.



```
! edgetpu_compiler $TFLITE_FILE
Edge TPU Compiler version 16.0.384591198
Started a compilation timeout timer of 180 seconds.

Model compiled successfully in 36 ms.

Input model: CNN_raws_model_quantized_metadata.tflite
Input size: 25.99Kib
Output model: CNN_raws_model_quantized_metadata_edgetpu.tflite
Output size: 99.94Kib
On-chip memory used for caching model parameters: 0.00B
On-chip memory remaining for caching model parameters: 7.98Mib
Off-chip memory used for streaming uncached model parameters: 0.00B
Number of Edge TPU subgraphs: 1
Total number of operations: 29
Operation log: CNN_raws_model_quantized_metadata_edgetpu.log

Model successfully compiled but not all operations are supported by the Edge TPU.
A percentage of the model will instead run on the CPU, which is slower. If possible,
consider updating your model to use only operations supported by the Edge TPU.
For details, visit g.co/coral/model-reqs.

Number of operations that will run on Edge TPU: 1
Number of operations that will run on CPU: 28
See the operation log file for individual operation details.
Compilation child process completed within timeout period.
Compilation succeeded!
```

Figura 2.15. Aviso obtenido al intentar compilar el modelo TFLite para los datos crudos de la señal para que se pueda ejecutar en la TPU de la placa Coral Dev. Se indica que solo la 1^a de las 29 operaciones se puede ejecutar en la TPU, por lo que las 28 operaciones siguientes se ejecutarán en la CPU.

2.3.2. Red neuronal compleja sobre la que aplicar transfer learning

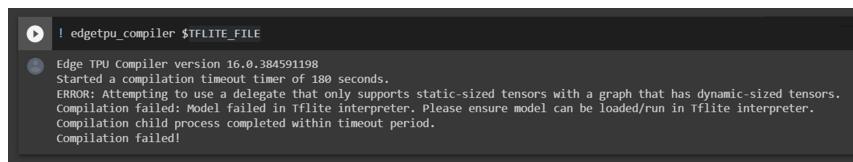
Para la segunda parte del trabajo se utilizó un script disponible públicamente en la página de TensorFlow para dispositivos móviles y edge [22] como base para llevar a cabo el transfer learning sobre el conjunto de datos generado en español. Este script utiliza un paquete de TensorFlow llamado *TensorFlow Lite Model Maker* para reentrenar un modelo de reconocimiento de lenguaje que clasifica palabras de activación. El modelo que se reentrena se conoce como *BrowserFft*, un modelo generado en JavaScript y entrenado con la versión 2 del mismo conjunto de datos que hemos utilizado en la primera parte, *Speech commands*. Este script, que puede ejecutarse directamente en Google Colab, se encarga de aplicar transfer learning sobre el conjunto de datos que el usuario decida subir. Se utilizan muchas clases y objetos propios de Model Maker, por lo que no se permiten realizar modificaciones de manera sencilla.

Explicamos el script de manera muy resumida. En primer lugar divide el conjunto de datos subido en los conjuntos de entrenamiento, validación y test, añadiendo en todos los casos muestras de audios con silencio o sonido de fondo para entrenar una nueva categoría, de modo que el modelo final no salte si no detecta claramente una de las palabras de activación. A continuación carga el modelo previamente entrenado utilizando clases específicas de Model Maker, dificultando mucho que se pueda acceder directamente a él. En el Anexo (A2), en la Figura (A2.4a) se muestra un gráfico con la estructura de la red neuronal definida en el modelo, y donde se indica con un recuadro rojo cuál es

la capa que se va a reentrenar al realizar transfer learning. Analizando dicho gráfico vemos que en principio el modelo no es excesivamente complejo, pero suponemos que la estructura ha sido optimizada para este problema. Siguiendo con el script, después de cargar el modelo, lo compila y lleva a cabo el entrenamiento. Posteriormente analiza su bondad utilizando el accuracy, convierte el modelo normal a formato .tflite y establece un proceso para realizar inferencia sobre muestras aleatorias del conjunto.

Para poder realizar de manera completa esta parte del trabajo, se tuvieron que hacer pequeñas modificaciones al script base y se tuvo que añadir toda la parte de cuantización del modelo. Como en este caso no disponíamos del modelo base sin entrenar, se tuvo que aplicar el proceso de post-cuantización de manera similar a lo hecho con el modelo convolucional de los datos crudos de los audios. Siguiendo ese mismo planteamiento y utilizando un conjunto de datos representativo válido, se obtuvo un modelo TFLite completamente cuantizado a enteros. Debido a todo esto, en este caso no disponemos del modelo normal, pero disponemos del modelo normal en formato TFLite y el modelo TFLite cuantizado. En el Anexo (A2), en la Figura (A2.4b) se muestra un gráfico con la estructura del modelo cuantizado y convertido a TFLite, donde se observa un modelo altamente complejo, con muchas operaciones y capas introducidas antes del modelo convolucional como tal, probablemente debido al intento de transformar las operaciones no permitidas en otras que sí que lo son, para llevar a cabo la conversión del modelo normal en TFLite.

Por último, mencionar que en este caso tampoco fue posible obtener un modelo TFLite cuantizado y preparado para ser ejecutado en la TPU de la placa Coral Dev, ya que al intentar compilar el modelo para la TPU saltó un error de incompatibilidad sobre la naturaleza dinámica de los tensores del modelo. Como no tenemos acceso al modelo base, no es algo que se pudiese modificar. Mostramos en la Figura (2.16) el error que aparecía al intentar compilar el modelo para la TPU.



```
! edgetpu_compiler $TFLITE_FILE
Edge TPU Compiler version 16.0.384591198
Started a compilation timeout timer of 180 seconds.
ERROR: Attempting to use a delegate that only supports static-sized tensors with a graph that has dynamic-sized tensors.
Compilation failed: Model failed in tflite interpreter. Please ensure model can be loaded/run in Tflite interpreter.
Compilation child process completed within timeout period.
Compilation failed!
```

Figura 2.16. Error obtenido al intentar compilar el modelo TFLite cuantizado de la red neuronal compleja sobre la que se ha aplicado transfer learning, debido a incompatibilidades por la naturaleza dinámica de los tensores.

Capítulo 3

Resultados

Entendido el enfoque que se ha seguido para el desarrollo del trabajo y teniendo claras las herramientas, los datos y los modelos utilizados, pasamos a describir los resultados que hemos obtenido en cada una de las partes.

Empezaremos detallando los resultados obtenidos en la primera parte del trabajo, comentando cómo han funcionado los diferentes modelos entrenados. Para ello se mostrará cómo se llevó a cabo el entrenamiento de cada modelo, analizando su bondad y rendimiento por medio de los diferentes valores del accuracy obtenidos sobre los diferentes conjuntos de entrenamiento, validación y test. También se compararán los modelos normales con sus correspondientes versiones cuantizadas con la finalidad de estudiar si la cuantización influye en el modelo, y en caso afirmativo, de qué manera lo hace.

Posteriormente se seguirá un planteamiento similar con el modelo complejo sobre el que se ha aplicado transfer learning. Veremos la bondad del modelo por medio del accuracy en cada conjunto de datos y compararemos el modelo normal con la versión cuantizada.

Por último, y para terminar el capítulo, se hará un pequeño análisis sobre las ejecuciones de cada modelo en la placa Coral Dev, con el objetivo de verificar que los modelos pueden ser ejecutados sin problemas en la placa, y para analizar si la ejecución en la TPU supone una mejora significativa en los tiempos de ejecución, y ver así si los requisitos que hay que cumplir a la hora de definir los modelos merecen realmente la pena.

3.1. Modelos sencillos

Empezamos mostrando y analizando los resultados de la primera parte del trabajo. Primero analizaremos el caso de los espectrogramas de Mel, seguiremos con el de los MFCCs y terminaremos con el de los datos crudos de los audios. Por último compararemos de manera general todos los modelos simples entre sí, para ver de entre todos ellos cuál podría ser considerado el mejor teniendo en cuenta diferentes variables.

3.1.1. CNN utilizando los espectrogramas de Mel

El modelo basado en redes convolucionales 2D para tratar los datos en forma de espectrogramas de Mel se entrenó a lo largo de 25 épocas con un batch size de 50 muestras, utilizando el optimizador *adam* y la función de coste *sparse categorical crossentropy*. Mostramos en la Figura (3.1) la evolución del accuracy y del loss para los conjuntos de entrenamiento y validación durante el propio entrenamiento del modelo.

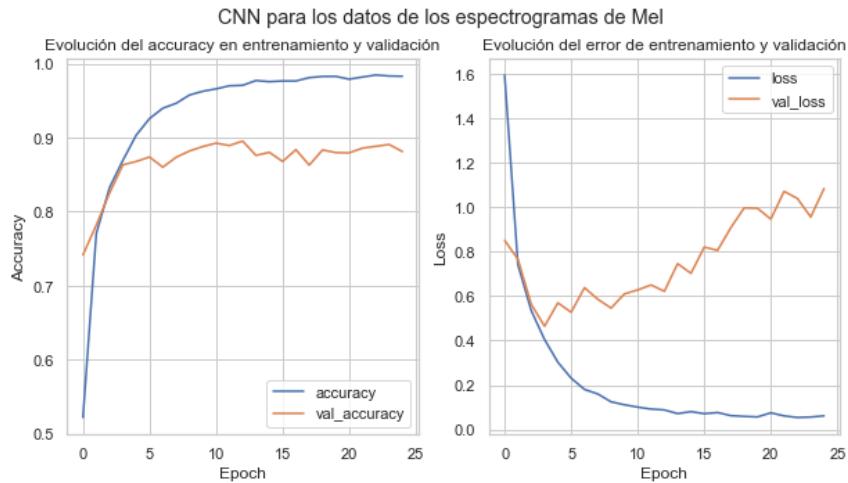


Figura 3.1. Evolución del accuracy y del loss para los conjuntos de entrenamiento y validación durante el entrenamiento del modelo convolucional para los datos en forma de espectrogramas de Mel.

En la figura vemos que el comportamiento del conjunto de entrenamiento es el esperado, con el accuracy aumentando lentamente y el loss disminuyendo en consecuencia. En cambio, si nos fijamos en el conjunto de validación, vemos que desde épocas muy tempranas el accuracy alcanza valores cercanos al 90%, oscilando levemente, pero la evolución del loss es muy irregular ya que en las primeras épocas sí que disminuye pero a partir de la 4^a comienza a aumentar de manera constante. Esto podría indicar que el modelo está sobreajustando, pero no se puede confirmar ya que el accuracy se mantiene estable. La única conclusión que sacamos es que el modelo se podría haber entrenado completamente con unas 5-10 épocas.

Con el modelo completamente entrenado, se calculó el accuracy final sobre los conjuntos de entrenamiento y de test. Obtuvimos que el accuracy en entrenamiento fue del 98.82% y en test fue del 88.50%. Hay una diferencia del 10%, similar a la que se observa en la figura entre los conjuntos de entrenamiento y validación. Esta diferencia puede considerarse un poco grande, pero el valor del accuracy obtenido en test no es malo en absoluto y nos vale como resultado final. Mostramos en las Figuras (3.2) y (3.3) las matrices de confusión sobre los conjuntos de entrenamiento y de test para analizar dónde está fallando más el modelo.

En la matriz de confusión sobre el conjunto de entrenamiento vemos que todas las clasificaciones están por encima del 90%, presentando una clasificación perfecta para la etiqueta "six", y siendo la etiqueta "on" donde hay una mayor cantidad de fallos, con un accuracy del 94.48%. Estos fallos vienen principalmente por errores con respecto a

		Matriz de confusión para el conjunto de train (CNN para los espectrogramas de Mel)																	
		00-zero	01-one	02-two	03-three	04-four	05-five	06-six	07-seven	08-eight	09-nine	10-on	11-off	12-up	13-down	14-left	15-right	16-yes	17-no
Etiquetas reales	00-zero	98.95	0.00	0.04	0.13	0.00	0.00	0.00	0.00	0.08	0.00	0.00	0.00	0.00	0.25	0.00	0.42	0.13	
	01-one	0.00	99.15	0.00	0.00	0.08	0.00	0.00	0.00	0.00	0.04	0.00	0.00	0.00	0.51	0.17	0.04	0.00	
	02-two	0.08	0.08	98.75	0.12	0.04	0.00	0.17	0.04	0.12	0.00	0.04	0.00	0.00	0.17	0.00	0.00	0.29	0.08
	03-three	0.04	0.00	0.04	99.71	0.00	0.00	0.04	0.04	0.00	0.04	0.00	0.00	0.00	0.00	0.00	0.08	0.00	0.00
	04-four	0.00	0.72	0.00	0.00	98.43	0.13	0.04	0.00	0.08	0.00	0.00	0.04	0.04	0.00	0.42	0.04	0.04	0.00
	05-five	0.00	0.22	0.00	0.00	0.00	98.18	0.04	0.00	0.00	0.49	0.09	0.00	0.04	0.00	0.18	0.75	0.00	0.00
	06-six	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	07-seven	0.04	0.09	0.00	0.09	0.00	0.09	0.00	98.63	0.00	0.13	0.00	0.00	0.04	0.04	0.73	0.04	0.09	0.00
	08-eight	0.04	0.00	0.00	0.09	0.00	0.00	0.39	0.13	99.02	0.00	0.04	0.00	0.00	0.00	0.09	0.04	0.17	0.00
	09-nine	0.04	0.00	0.00	0.04	0.00	0.04	0.00	0.04	0.00	99.31	0.04	0.00	0.04	0.13	0.09	0.21	0.00	0.00
	10-on	0.00	2.05	0.00	0.04	0.26	1.03	0.00	0.09	0.00	0.26	94.48	0.94	0.09	0.04	0.51	0.17	0.00	0.04
	11-off	0.00	0.00	0.00	0.00	0.00	0.04	0.09	0.00	0.00	0.00	0.04	98.98	0.62	0.00	0.18	0.00	0.04	0.00
	12-up	0.00	0.04	0.00	0.00	0.00	0.00	0.09	0.00	0.00	0.00	0.04	0.26	99.43	0.00	0.13	0.00	0.00	0.00
	13-down	0.09	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.09	0.00	0.00	0.00	98.92	0.13	0.00	0.39	0.35
	14-left	0.00	0.22	0.00	0.00	0.00	0.00	0.04	0.00	0.00	0.00	0.00	0.00	0.04	99.57	0.04	0.09	0.00	0.00
	15-right	0.00	0.26	0.00	0.00	0.00	0.04	0.00	0.00	0.00	0.09	0.04	0.00	0.00	0.22	99.35	0.00	0.00	0.00
	16-yes	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.13	0.00	99.87	0.00	0.00
	17-no	0.17	0.04	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.13	0.00	0.00	0.04	0.46	0.00	0.00	1.18	97.94

Figura 3.2. Matriz de confusión para el conjunto de entrenamiento del modelo convolucional para los datos en forma de espectrograma de Mel.

la etiqueta “one”, ya que ambas palabras tiene una pronunciación que puede ser muy similar.

En la matriz de confusión sobre el conjunto de test vemos que todas las etiquetas salvo la de “on” presentan un accuracy superior al 80%, llegando incluso a superar el 94% en etiquetas como “three”, “six” y “yes”. Igual que sucede con el conjunto de entrenamiento, la etiqueta “on” es la que presenta un accuracy más bajo, en este caso del 75.92%, y la mayoría de fallos vienen de errores con respecto a la etiqueta “one”, con una tasa de error cercana al 5%.

Como ya hemos comentado en el capítulo anterior, después del entrenamiento del modelo normal se pasó a cuantizar el modelo. Para el caso de los espectrogramas de Mel, el proceso se pudo llevar a cabo aplicando una pre-cuantización y entrenando un modelo preparado para dicha cuantización. Posteriormente, haciendo uso de un conjunto representativo de los datos, se cuantificó completamente el modelo a enteros. En la Tabla (3.1) mostramos un resumen de las accuracies que se obtuvieron sobre cada uno de los conjuntos para cada uno de los 3 modelos generados en el proceso: el modelo normal (ya comentado), el pre-cuantizado y el cuantizado.

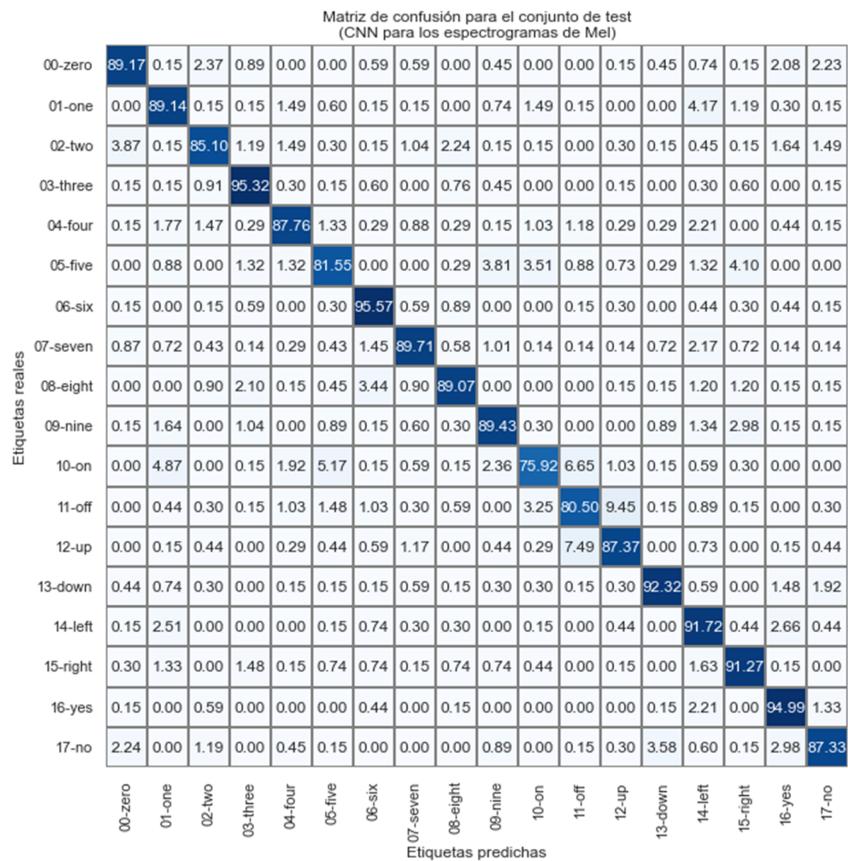


Figura 3.3. Matriz de confusión para el conjunto de test del modelo convolucional para los datos en forma de espectrograma de Mel.

Tabla 3.1. Resumen de los accuracies obtenidos sobre cada uno de los conjuntos para cada uno de los 3 modelos convolucionales para los datos en forma de espectrograma de Mel generados durante el proceso completo.

Modelo	Accuracy en entrenamiento	Accuracy en validación	Accuracy en test
normal	98.82%	88.12%	88.50%
pre-cuantizado	95.52%	85.48%	86.05%
cuantizado	95.46%		85.90%

Analizando la tabla, vemos que para este caso obtuvimos que las accuracies de los modelos pre-cuantizado y cuantizado son prácticamente idénticas, pero ambas son ligeramente inferiores a las obtenidas con el modelo normal. De este modo, vemos como la cuantización completa de los pesos y las activaciones a enteros influye negativamente en el rendimiento del modelo, pero como la disminución del rendimiento no es muy grande, se puede asumir con tal de poder ejecutar el modelo en la TPU de la placa Coral Dev.

3.1.2. CNN utilizando los MFCCs

El modelo basado en redes convolucionales 2D para tratar los datos en forma de MFCCs tuvo un tratamiento muy similar al modelo de los espectrogramas de Mel. Igual que para el caso anterior, el modelo se entrenó a lo largo de 25 épocas con un batch size de 50 muestras, utilizando el optimizador *adam* y la función de coste *sparse categorical crossentropy*. Mostramos en la Figura (3.4) la evolución del accuracy y del loss para los conjuntos de entrenamiento y validación durante el propio entrenamiento del modelo.

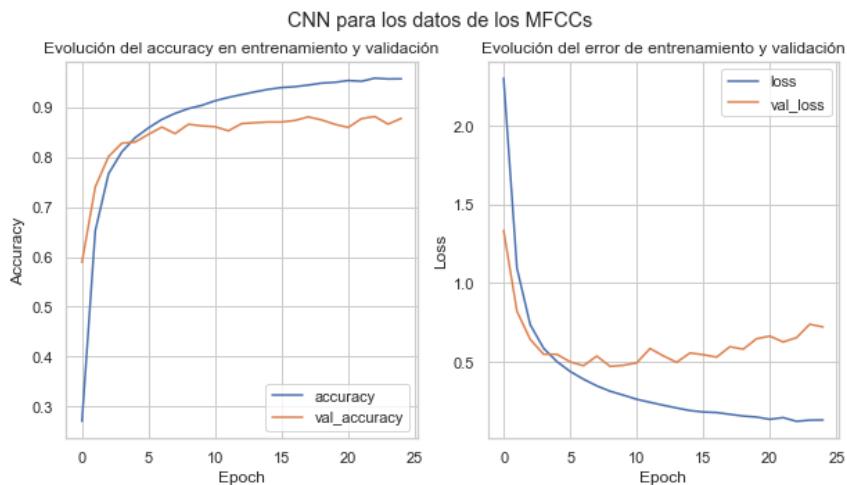


Figura 3.4. Evolución del accuracy y del loss para los conjuntos de entrenamiento y validación durante el entrenamiento del modelo convolucional para los datos en forma de MFCCs.

En la figura vemos que obtenemos un resultado muy parecido al del modelo de los espectrogramas de Mel. De nuevo vemos que en el conjunto de entrenamiento el accuracy aumentando lentamente y el loss disminuyendo en consecuencia, tal y como debería ser. En cambio, si nos fijamos en el conjunto de validación, vemos que desde épocas muy tempranas el accuracy alcanza valores ligeramente superiores al 85%, oscilando levemente, pero la evolución del loss vuelve a ser irregular. Igual que sucedía en el caso anterior, en las primeras épocas el loss disminuye pero a partir de la 10^a comienza a aumentar ligeramente (de forma mucho menos agresiva que en el modelo anterior). Las conclusiones que extraemos son las mismas: es posible que el modelo esté levemente sobreajustado, pero el accuracy se mantiene estable. Para este modelo, es posible que hubiesen bastado 10 épocas para completar el entrenamiento.

Con el modelo completamente entrenado, se calculó el accuracy final sobre los conjuntos de entrenamiento y de test. Obtuvimos que el accuracy en entrenamiento fue del 97.14% y en test fue del 87.81%. Vuelve a haber una diferencia del 10%, idéntica a la observada en la figura entre los conjuntos de entrenamiento y validación. La diferencia vuelve a ser elevada, pero los resultados obtenidos en el conjunto de test son muy buenos y nos valen como resultado final, siendo ligeramente mejores a los del modelo anterior. Mostramos en las Figuras (3.5) y (3.6) las matrices de confusión sobre los conjuntos de entrenamiento y de test para analizar dónde está fallando más el modelo.

En la matriz de confusión sobre el conjunto de entrenamiento vemos que todas las clasificaciones están por encima del 94% a excepción de la etiqueta “on”, que vuelve a

		Matriz de confusión para el conjunto de train (CNN para los MFCCs)																		
		00-zero	01-one	02-two	03-three	04-four	05-five	06-six	07-seven	08-eight	09-nine	10-on	11-off	12-up	13-down	14-left	15-right	16-yes	17-no	
Etiquetas reales	00-zero	98.53	0.00	0.63	0.00	0.04	0.04	0.04	0.08	0.00	0.00	0.04	0.00	0.04	0.04	0.04	0.08	0.00	0.21	0.21
	01-one	0.00	94.92	0.08	0.08	0.76	0.34	0.00	0.04	0.00	0.21	1.40	0.13	0.04	0.04	0.21	1.61	0.00	0.13	
	02-two	0.54	0.00	97.09	0.46	0.58	0.04	0.33	0.25	0.00	0.00	0.08	0.04	0.08	0.29	0.00	0.00	0.00	0.21	
	03-three	0.00	0.00	0.13	98.71	0.17	0.21	0.04	0.17	0.08	0.00	0.04	0.00	0.00	0.00	0.00	0.42	0.04	0.00	
	04-four	0.04	0.21	0.04	0.13	97.63	0.34	0.04	0.30	0.00	0.00	0.51	0.47	0.13	0.00	0.00	0.13	0.04	0.00	
	05-five	0.00	0.04	0.00	0.00	0.09	98.27	0.00	0.09	0.09	0.09	0.53	0.09	0.18	0.00	0.00	0.53	0.00	0.00	
	06-six	0.08	0.00	0.17	0.17	0.04	0.08	98.48	0.17	0.30	0.00	0.00	0.08	0.00	0.08	0.30	0.00	0.00	0.04	
	07-seven	0.00	0.04	0.09	0.26	0.13	0.47	0.43	97.68	0.09	0.04	0.04	0.00	0.43	0.04	0.04	0.17	0.00	0.04	
	08-eight	0.00	0.00	0.09	0.56	0.00	0.13	0.04	0.00	98.72	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.21	0.17	0.04
	09-nine	0.00	0.64	0.00	0.43	0.00	1.59	0.00	0.17	0.04	94.29	0.26	0.00	0.00	0.21	0.13	2.15	0.00	0.09	
	10-on	0.00	0.39	0.04	0.04	0.30	3.47	0.04	0.04	0.00	0.17	93.84	0.98	0.43	0.00	0.21	0.04	0.00	0.00	
	11-off	0.04	0.00	0.00	0.00	0.36	0.49	0.00	0.00	0.09	0.00	1.02	96.01	2.00	0.00	0.00	0.00	0.00	0.00	
	12-up	0.00	0.00	0.00	0.00	0.09	0.09	0.17	0.09	0.00	0.00	0.17	1.31	97.86	0.09	0.04	0.00	0.00	0.09	
	13-down	0.13	0.26	0.30	0.04	0.00	0.13	0.00	1.12	0.13	0.26	0.30	0.00	0.13	95.29	0.26	0.04	0.13	1.47	
	14-left	0.04	0.04	0.00	0.00	0.09	0.00	0.22	0.04	0.04	0.00	0.00	0.17	0.17	0.04	98.01	0.52	0.61	0.00	
	15-right	0.00	0.04	0.00	0.35	0.00	0.82	0.04	0.04	0.22	0.00	0.00	0.00	0.00	0.30	98.19	0.00	0.00	0.00	
	16-yes	0.08	0.00	0.00	0.00	0.00	0.00	0.34	0.00	0.17	0.04	0.00	0.13	0.00	0.13	1.10	0.00	98.01	0.00	
	17-no	0.04	0.08	0.21	0.00	0.21	0.00	0.00	0.17	0.00	0.42	0.04	0.04	0.34	0.88	0.25	0.00	0.29	97.01	

Figura 3.5. Matriz de confusión para el conjunto de entrenamiento del modelo convolucional para los datos en forma de MFCCs.

presentar el resultado más bajo, pero en este casi no destaca tanto ya que el accuracy es del 93.84%. La mayor diferencia en esta vez es que los fallos para la etiqueta vienen principalmente de errores con respecto a la etiqueta “five”. Estos fallos no tienen una justificación tan clara como la del caso anterior, ya que la pronunciación ni siquiera es parecida.

En la matriz de confusión sobre el conjunto de test vemos que todas las etiquetas presentan un accuracy superior al 80%, sin excepciones, muchas de ellas llegando a superar el 90%. De nuevo vuelve a ser la etiqueta “on” la que presenta un menor accuracy, seguido de cerca por las etiquetas “off” y “nine”. Para la etiqueta “on” la mayor tasa de error, superior al 7%, viene con respecto a la etiqueta “five” (igual que en el entrenamiento), para la etiqueta “off” es con respecto a la etiqueta “up” con una tasa de error de casi el 8%, y para la etiqueta “nine” es con respecto a la etiqueta “right”, con una tasa del 5%.

Para el caso de los MFCCs el proceso de cuantización fue idéntico al del caso de los espectrogramas de Mel, con la única diferencia de que el conjunto representativo era el correspondiente a los MFCCs. Esto se traduce en que primero se generó un modelo pre-cuantizado sobre el que se llevó a cabo la cuantización completa a enteros. En la Tabla (3.2) mostramos el resumen de los accuracies que se obtuvieron sobre cada uno de los conjuntos para cada uno de los 3 modelos diferentes.

		Matriz de confusión para el conjunto de test (CNN para los MFCCs)																	
		00-zero	00-one	02-two	03-three	04-four	05-five	06-six	07-seven	08-eight	09-nine	10-on	11-off	12-up	13-down	14-left	15-right	16-yes	17-no
Etiquetas reales	00-zero	91.39	0.00	3.56	0.15	0.30	0.15	0.59	1.19	0.15	0.00	0.00	0.00	0.74	0.15	0.15	0.45	1.04	
	00-one	0.00	84.67	0.15	0.30	2.23	1.04	0.00	0.00	1.79	3.42	0.15	0.30	0.60	2.23	2.38	0.00	0.74	
	02-two	1.94	0.45	86.74	1.19	1.64	0.15	1.19	1.19	0.89	0.15	0.60	0.15	0.89	1.19	0.30	0.15	0.00	1.19
	03-three	0.15	0.30	0.15	93.96	0.15	0.30	0.30	1.06	1.06	0.45	0.15	0.00	0.00	0.15	0.00	1.66	0.00	0.15
	04-four	0.29	1.18	0.44	0.15	89.38	1.33	0.15	1.03	0.15	0.00	3.10	1.33	0.44	0.15	0.88	0.00	0.00	0.00
	05-five	0.15	0.44	0.00	0.88	0.73	87.70	0.00	0.88	0.15	1.46	2.49	1.17	0.73	0.29	0.44	2.49	0.00	0.00
	06-six	0.00	0.00	0.15	0.30	0.30	0.44	93.80	2.51	1.33	0.00	0.00	0.30	0.00	0.00	0.30	0.15	0.44	0.00
	07-seven	0.29	0.00	0.87	0.72	0.58	1.01	3.19	89.57	0.14	0.14	0.29	0.14	1.16	0.87	0.00	0.43	0.00	0.58
	08-eight	0.00	0.00	0.90	2.99	0.30	0.45	1.20	0.15	90.72	0.45	0.00	0.75	0.15	0.00	0.15	1.50	0.30	0.00
	09-nine	0.00	2.98	0.00	0.60	0.00	2.53	0.15	0.74	0.74	82.89	1.34	0.15	0.30	1.04	0.74	5.36	0.15	0.30
	10-on	0.15	2.22	0.30	0.15	1.62	7.39	0.00	0.30	0.00	0.30	81.39	3.55	1.03	0.74	0.15	0.74	0.00	0.00
	11-off	0.00	0.74	0.00	0.00	1.33	2.22	0.15	0.00	0.30	0.00	3.25	82.72	7.98	0.30	0.74	0.15	0.15	0.00
	12-up	0.00	0.29	0.59	0.00	0.29	1.17	0.15	0.73	0.44	0.00	1.17	6.61	85.02	1.32	1.17	0.15	0.29	0.59
	13-down	0.74	0.74	0.59	0.00	0.15	0.89	0.30	1.92	0.30	1.92	0.74	0.00	0.74	84.34	1.03	0.00	0.15	5.47
	14-left	0.59	0.59	0.15	0.15	0.15	0.30	0.89	0.44	0.44	0.59	0.00	1.18	1.33	0.15	88.17	2.96	1.63	0.30
	15-right	0.00	1.18	0.00	2.22	0.00	2.37	0.59	0.30	1.18	0.44	0.00	0.15	0.00	0.15	1.92	89.35	0.15	0.00
	16-yes	0.15	0.29	0.15	0.00	0.15	0.00	0.88	0.00	0.74	0.00	0.00	0.15	0.15	0.15	2.65	0.74	93.67	0.15
	17-no	1.34	0.89	1.19	0.00	0.30	0.45	0.00	0.15	0.00	1.04	0.00	0.00	0.60	6.41	1.49	0.00	1.04	85.10

Figura 3.6. Matriz de confusión para el conjunto de test del modelo convolucional para los datos en forma de MFCCs.

Tabla 3.2. Resumen de los accuracies obtenidos sobre cada uno de los conjuntos para cada uno de los 3 modelos convolucionales para los datos en forma de MFCCs generados durante el proceso completo.

Modelo	Accuracy en entrenamiento	Accuracy en validación	Accuracy en test
normal	97.14%	87.74%	87.81%
pre-cuantizado	96.02%	86.08%	86.65%
cuantizado	93.74%		86.23%

Analizando la tabla, vemos que para este caso obtuvimos unas accuracies muy similares para todos los casos, a excepción del conjunto de entrenamiento para el modelo cuantizado, que presenta una diferencia del 4% y el 3% con respecto a los modelos normal y pre-cuantizado, respectivamente. Sin embargo, esta es una diferencia sobre el conjunto de entrenamiento, por lo que tampoco implica nada importante. Fijándonos en las accuracies en test vemos que todas están comprendidas en un intervalo del 1%, lo cuál es muy positivo. En este caso también vemos como la cuantización del modelo influye negativamente, pero en este caso la disminución del rendimiento puede considerarse despreciable, por lo que las modificaciones con tal de ejecutar los modelos en la TPU de la placa Coral Dev son perfectamente asumibles.

3.1.3. CNN utilizando los datos crudos de las señales

El modelo basado en redes convolucionales 1D para tratar los datos crudos de los audios fue el que más diferencias presentó en el tratamiento. En este caso, el modelo se entrenó con un batch size de 50 muestras, utilizando el optimizador *adam* y la función de coste *sparse categorical crossentropy*, pero durante 75 épocas, ya que la evolución del accuracy fue mucho más lenta y progresiva. Mostramos en la Figura (3.7) la evolución del accuracy y del loss para los conjuntos de entrenamiento y validación durante el propio entrenamiento del modelo.

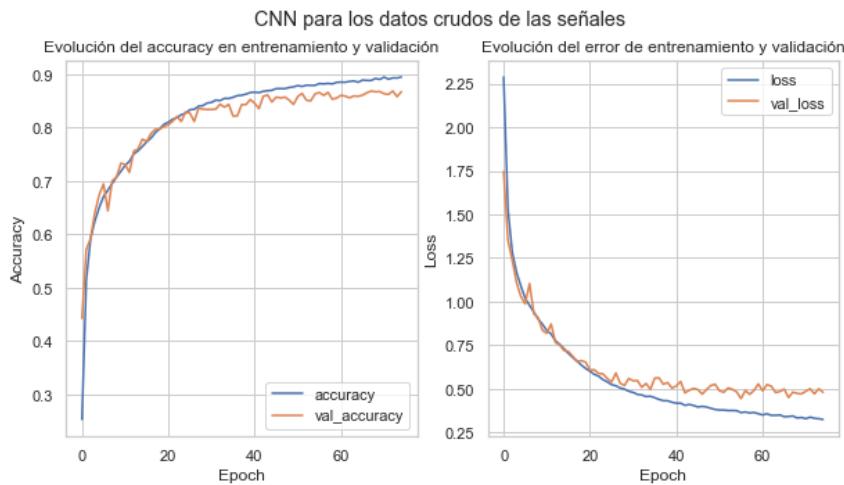


Figura 3.7. Evolución del accuracy y del loss para los conjuntos de entrenamiento y validación durante el entrenamiento del modelo convolucional para los datos crudos de los audios.

En la figura vemos que para este modelo obtenemos unos resultados notablemente diferentes en cuanto a la evolución del loss. El conjunto de entrenamiento vuelve a comportarse según lo esperado, con el accuracy aumentando lentamente (aunque en este caso no supera holgadamente el 90% como sí lo hacía en los otros casos) y el loss disminuyendo en consecuencia. Lo interesante es que sucede lo mismo en el conjunto de validación. Antes habíamos visto cómo el loss era muy irregular y aumentaba tras unas pocas épocas, pero ahora el loss tiene una evolución correcta, disminuyendo lentamente. Del mismo modo, el accuracy en épocas tempranas aumenta muy rápido, pero a partir de la época 30 lo hace de una manera mucho más lenta, estancándose alrededor del 86%. Todo esto parece indicar que el modelo aprende de manera constante sin llegar a sobreajustarse a los datos en ningún momento.

Con el modelo completamente entrenado, se calculó el accuracy final sobre los conjuntos de entrenamiento y de test. Obtuvimos que el accuracy en entrenamiento fue del 90.49% y en test fue del 87.16%. Para este modelo se obtiene una diferencia mucho menor, tan solo del 3%, igual a la observada en la figura entre los conjuntos de entrenamiento y validación. Esta diferencia tan baja puede indicar que el aprendizaje del modelo ha sido muy efectivo y generalizado. Mostramos en las Figuras (3.8) y (3.9) las matrices de confusión sobre los conjuntos de entrenamiento y de test para analizar dónde está fallando más el modelo.

		Matriz de confusión para el conjunto de train (CNN para los datos crudos de las señales)																	
		00-zero	01-one	02-two	03-three	04-four	05-five	06-six	07-seven	08-eight	09-nine	10-on	11-off	12-up	13-down	14-left	15-right	16-yes	17-no
Etiquetas reales	00-zero	92.77	0.00	1.18	0.13	0.46	0.00	0.13	0.42	0.13	0.13	0.00	0.00	0.00	0.97	0.08	0.00	1.22	2.39
	01-one	0.17	93.90	0.08	0.00	0.25	0.34	0.04	0.21	0.04	1.02	0.80	0.00	0.00	0.25	0.42	1.78	0.21	0.47
	02-two	2.95	0.04	87.16	3.49	1.79	0.04	1.04	0.54	0.87	0.04	0.04	0.04	0.08	0.87	0.04	0.00	0.25	0.71
	03-three	0.25	0.00	3.38	90.23	0.04	0.50	0.63	0.88	2.05	0.33	0.33	0.00	0.17	0.00	0.00	1.13	0.04	0.04
	04-four	0.34	1.14	0.80	0.04	91.91	0.93	0.04	0.04	0.21	0.04	0.85	1.91	0.17	0.38	0.17	0.64	0.17	0.21
	05-five	0.00	0.66	0.13	0.49	0.40	88.97	0.04	0.31	0.18	1.06	2.66	0.75	0.62	1.06	0.04	2.57	0.00	0.04
	06-six	0.38	0.08	1.69	0.46	0.38	0.00	87.44	2.49	2.45	0.00	0.00	0.08	1.18	0.13	1.05	0.97	1.22	0.00
	07-seven	0.09	0.26	0.30	0.56	0.17	0.39	0.43	93.26	0.13	0.34	0.34	0.00	0.30	2.83	0.09	0.30	0.09	0.13
	08-eight	0.26	0.00	2.44	3.12	0.51	0.21	2.31	0.43	88.57	0.04	0.13	0.09	0.34	0.17	0.30	0.73	0.13	0.21
	09-nine	0.04	0.77	0.17	0.21	0.00	0.64	0.04	0.47	0.21	92.49	0.43	0.09	0.00	0.90	0.21	1.50	0.00	1.80
	10-on	0.04	1.16	0.00	0.17	0.56	2.65	0.00	0.43	0.00	0.34	91.36	2.18	0.60	0.26	0.00	0.13	0.04	0.09
	11-off	0.00	0.00	0.04	0.00	1.02	0.27	0.00	0.04	0.18	0.09	1.29	92.41	4.08	0.04	0.36	0.04	0.00	0.13
	12-up	0.00	0.00	0.09	0.00	0.35	0.78	0.13	0.39	0.35	0.00	0.44	9.46	86.41	0.04	1.22	0.13	0.04	0.17
	13-down	0.35	0.30	0.48	0.09	0.13	0.13	0.04	0.91	0.04	0.99	0.39	0.04	0.13	94.17	0.04	0.09	0.13	1.56
	14-left	0.30	0.74	0.09	0.04	0.30	0.04	0.22	0.35	0.04	0.56	0.04	0.30	0.91	0.13	85.65	2.04	7.11	1.13
	15-right	0.00	1.68	0.13	0.39	0.09	1.16	0.22	0.35	0.39	2.24	0.09	0.13	0.26	0.22	1.94	90.12	0.13	0.47
	16-yes	0.89	0.21	0.42	0.00	0.13	0.00	0.46	0.08	0.04	0.04	0.00	0.00	0.04	0.51	5.53	0.55	89.73	1.35
	17-no	0.88	0.13	0.34	0.00	0.25	0.00	0.00	0.08	0.08	2.82	0.00	0.17	0.08	2.32	0.21	0.04	0.29	92.29
		00-zero	01-one	02-two	03-three	04-four	05-five	06-six	07-seven	08-eight	09-nine	10-on	11-off	12-up	13-down	14-left	15-right	16-yes	17-no

Figura 3.8. Matriz de confusión para el conjunto de entrenamiento del modelo convolucional para los datos crudos de los audios.

En la matriz de confusión sobre el conjunto de entrenamiento vemos que todas las clasificaciones se encuentran en un intervalo entre el 85% y el 94%. En este caso, son las etiquetas “left” y “up” las que presentan los valores más bajos (85% y 86% respectivamente). Vemos que la mayor parte del error para la etiqueta “left” viene con respecto a la etiqueta “yes”, con una tasa del 7%, mientras que para la etiqueta “up” el error viene con respecto a la etiqueta “off”, con una tasa superior al 9%. Estas mayores tasas de error pueden estar justificadas, ya que las pronunciaciones en inglés pueden considerarse similares en ambos casos.

En la matriz de confusión sobre el conjunto de test vemos que todas las etiquetas presentan un accuracy superior al 80%. Solo las etiquetas “seven”, “nine”, “on”, “down” y “no” superan ligeramente el 90%. Es interesante que etiquetas como “four” y “yes” que en el conjunto de entrenamiento habían obtenido accuracies altas, en el conjunto de test están bastante más por debajo. Se mantiene el hecho de que las etiquetas “left” y “up” son las que presentan las accuracies más bajas, con las mismas fuentes de error que en el conjunto de entrenamiento, y tasas de fallo del 11% y 9%, respectivamente.

Para el caso de los datos crudos de los audios ya hemos explicado que no se pudo llevar a cabo el proceso de cuantización del mismo modo que en los casos anteriores. Para este modelo se han utilizado capas convolucionales 1D, que no están permitidas según la placa Coral Dev. Esto implicaba que no se pudiese llevar a cabo la pre-cuantización

		Matriz de confusión para el conjunto de test (CNN para los datos crudos de las señales)																	
		00-zero	01-one	02-two	03-three	04-four	05-five	06-six	07-seven	08-eight	09-nine	10-on	11-off	12-up	13-down	14-left	15-right	16-yes	17-no
Etiquetas reales	00-zero	89.32	0.15	1.93	0.00	0.59	0.00	0.45	0.45	0.15	0.00	0.00	0.00	0.00	1.78	0.00	0.00	1.19	4.01
	01-one	0.30	88.99	0.30	0.15	0.45	0.15	0.15	0.45	0.30	1.49	2.83	0.00	0.00	1.04	0.30	2.53	0.45	0.15
	02-two	3.43	0.00	82.56	5.22	2.09	0.00	1.79	0.75	1.94	0.00	0.15	0.00	0.00	1.04	0.30	0.00	0.45	0.30
	03-three	0.15	0.15	5.74	81.72	0.91	1.36	1.06	1.36	3.17	0.45	0.30	0.00	0.15	0.00	0.00	3.47	0.00	0.00
	04-four	0.15	1.62	1.03	0.29	88.64	0.74	0.29	0.15	0.44	0.29	1.77	3.10	0.15	0.15	0.15	0.29	0.15	0.59
	05-five	0.00	0.59	0.15	0.44	0.73	86.68	0.15	0.59	0.59	1.17	3.51	1.17	1.32	0.59	0.15	2.20	0.00	0.00
	06-six	0.00	0.15	1.18	0.30	0.15	0.00	88.33	2.22	2.81	0.00	0.00	0.00	1.33	0.30	0.74	1.18	1.33	0.00
	07-seven	0.00	0.00	0.29	0.29	0.72	0.43	0.72	90.58	0.43	0.58	0.58	0.00	0.87	3.91	0.14	0.29	0.00	0.14
	08-eight	0.00	0.00	2.69	4.49	0.90	0.75	3.59	0.60	84.43	0.15	0.15	0.30	0.15	0.00	0.30	1.20	0.15	0.15
	09-nine	0.00	0.89	0.00	0.15	0.00	0.74	0.00	0.60	0.15	91.37	0.30	0.00	0.00	0.45	0.00	2.83	0.15	2.38
	10-on	0.00	0.89	0.00	0.74	1.18	2.81	0.00	0.44	0.30	0.30	90.25	1.92	0.74	0.15	0.00	0.30	0.00	0.00
	11-off	0.00	0.30	0.00	0.00	0.89	0.89	0.00	0.00	0.30	0.00	1.62	89.22	6.20	0.15	0.30	0.15	0.00	0.00
	12-up	0.00	0.29	0.29	0.00	0.00	0.44	0.29	0.88	0.00	0.00	0.59	11.16	83.55	0.29	1.76	0.29	0.00	0.15
	13-down	0.59	0.74	1.33	0.00	0.00	0.30	0.00	0.89	0.00	1.03	0.44	0.00	0.15	90.69	0.15	0.59	0.30	2.81
	14-left	0.15	1.92	0.30	0.00	0.15	0.00	0.44	0.59	0.00	1.04	0.00	0.30	0.89	0.00	82.40	2.07	8.73	1.04
	15-right	0.00	2.37	0.00	0.59	0.30	2.81	0.15	0.44	0.74	3.25	0.30	0.00	0.59	0.00	2.07	85.65	0.00	0.74
	16-yes	1.03	0.44	0.44	0.15	0.00	0.00	1.47	0.15	0.15	0.29	0.15	0.00	0.15	0.88	8.69	0.74	84.39	0.88
	17-no	0.60	0.15	0.30	0.00	0.00	0.00	0.00	0.00	0.15	3.28	0.00	0.00	0.45	2.68	1.79	0.15	0.45	90.01
		00-zero	01-one	02-two	03-three	04-four	05-five	06-six	07-seven	08-eight	09-nine	10-on	11-off	12-up	13-down	14-left	15-right	16-yes	17-no

Figura 3.9. Matriz de confusión para el conjunto de test del modelo convolucional para los datos crudos de los audios.

del modelo, por lo que directamente se tuvo que hacer una post-cuantización. Utilizando el correspondiente conjunto representativo de datos, se llevó a cabo la cuantización completa a enteros, partiendo del modelo normal. En la Tabla (3.3) mostramos el resumen de las accuracies que se obtuvieron sobre cada uno de los conjuntos para los 2 modelos diferentes.

Tabla 3.3. Resumen de los accuracies obtenidos sobre cada uno de los conjuntos para los 2 modelos convolucionales para los datos crudos de los audios.

Modelo	Accuracy en entrenamiento	Accuracy en validación	Accuracy en test
normal	90.49%	86.75%	87.16%
cuantizado	83.82%		80.75%

Analizando la tabla, vemos que en este caso obtuvimos un descenso muy significativo (de casi el 7%) de los accuracies de manera sistemática, en los conjuntos de entrenamiento y test. Se nota claramente cómo el hecho de no haber podido llevar a cabo la pre-cuantización ha afectado negativamente al modelo cuantizado final. Es cierto que los valores de los accuracies obtenidos no son bajos, pero la diferencia es notable con respecto a los otros casos. De este modo verificamos que la preparación para la cuantización del modelo es importante, ya que realizar directamente la post-cuantización tiene repercusiones

importantes en el rendimiento del modelo. Depende de los objetivos que tengamos para el modelo, será conveniente o no aceptar este descenso en el rendimiento para poder ejecutar los modelos en la TPU de la placa Coral Dev.

3.1.4. Comparación entre los diferentes modelos sencillos

Una vez analizados los resultados de manera individual para cada uno de los modelos, pasamos a realizar una breve comparación conjunta entre los 3 modelos. Tenemos el objetivo de intentar seleccionar cuál puede ser el mejor modelo teniendo en cuenta variables como la cantidad de preparación y procesado previo que hay que llevar a cabo, la complejidad del modelo según el número de parámetros a entrenar, y los resultados obtenidos del accuracy en test para los modelos normal y cuantizado.

En ningún momento hemos tenido un valor de referencia con el que comparar los resultados que obteníamos para cada uno de los modelos generados. No tenía mucho sentido generar un modelo simple para establecer un baseline, ya que los modelos necesariamente debían ser redes neuronales que poder ejecutar en la TPU de la placa Coral Dev. Por ello, se ha tomado como referencia un modelo entrenado aplicando transfer learning sobre el modelo complejo ya introducido, pero utilizando la versión 1 del conjunto de datos *speech commands*. Es decir, el modelo complejo ha sido entrenado con anterioridad con la versión 2 de los datos, pero utilizamos la versión 1, aplicando transfer learning, con un conjunto de test igual al 40% para generar unos valores del accuracy que podamos tomar como referencia superior, ya que entendemos que el modelo complejo ha sido optimizado para este problema⁽¹¹⁾.

Mostramos en la Tabla (3.4) un resumen comparativo entre los diferentes modelos sencillos que se han generado, teniendo en cuenta las diferentes variables que hemos considerado relevantes para escoger un mejor modelo. También se han añadido los accuracies del modelo obtenido al aplicar transfer learning sobre la versión 1 del conjunto de datos original, en inglés.

Tabla 3.4. Resumen comparativo entre los diferentes modelos sencillos entrenados, teniendo en cuenta diversas variables relevantes a la hora de seleccionar un modelo final.

Modelo	procesado previo	número de parámetros	accuracy en test (normal)	accuracy en test (cuantizado)
CNN espectrogramas de Mel	medio	411 050	88.50%	85.90%
CNN MFCCs	alto	98 733	87.81%	86.23%
CNN datos crudos	bajo	13 348	87.16%	80.75%
de referencia, aplicando TL			96.92%	91.31%

A partir de los datos mostrados en la tabla concluimos que si no es necesario cuantizar el modelo porque no queremos ejecutarlo en un dispositivo de Tiny ML, sin ninguna duda la mejor opción es el modelo basado en redes convolucionales 1D sobre los

⁽¹¹⁾Remarcamos que se ha llevado a cabo este procedimiento tan enrevesado porque no se han podido encontrar valores de referencia tabulados para el modelo complejo sobre el que se ha aplicado transfer learning.

datos crudos de los audios. Este modelo es el más simple, ya que tiene el menor número de parámetros a entrenar (con mucha diferencia con respecto a los otros dos), a la vez que ofrece un accuracy en test prácticamente igual, y con la ventaja extra de que el preprocesado que hay que aplicar sobre los datos es muy simple y reducido. Por otro lado, si sí que buscamos poder ejecutar el modelo en un dispositivo de Tiny ML y por ello necesitamos cuantizar el modelo, la opción a escoger se vuelve un poco indiferente entre las redes de capas convolucionales 2D para los datos como espectrogramas de Mel o como MFCCs. En el caso de los espectrogramas de Mel se obtiene un accuracy en test ligeramente inferior a expensas de un preprocesado previo reducido, a pesar de que el modelo tiene muchos más parámetros a entrenar. En cambio, en el caso de los MFCCs se obtiene un accuracy en test ligeramente superior, utilizando un modelo con un número de parámetros significativamente inferior, pero necesitando un preprocesado previo de los datos mucho más elaborado. Dependiendo de la velocidad con la que se pueda llevar a cabo el preprocesado y la importancia que se le den a pequeñas mejoras en el accuracy, será más conveniente un modelo u otro.

3.2. Transfer learning en una red compleja

Terminado el análisis de los resultados con los modelos sencillos, pasamos a analizar los resultados obtenidos aplicando transfer learning sobre la red compleja, más o menos con la misma estructura que se ha seguido hasta ahora.

Como ya hemos comentado, para realizar este proceso se utilizó un script público que no permitía hacer grandes modificaciones en el proceso de entrenamiento, por lo que, entre otras cosas, no se dispone de la evolución del accuracy y del loss. En este caso, el modelo se entrenó en 30 épocas con un batch size de 15 muestras. La red completa tenía 1 466 683 parámetros, de los cuales solo eran entrenables 38 019, correspondientes a la última capa densa.

Con el modelo entrenado, se calculó el accuracy sobre los conjuntos de entrenamiento y de test. Obtuimos que el accuracy en entrenamiento era del 100% y en test era del 93.49%. Hay una diferencia notable entre ambos valores, pero lo que llama la atención es el hecho de que la red proporcione un accuracy perfecto en entrenamiento. No podemos ver la evolución del loss, por lo que no podemos asegurar que el modelo esté sobreentrenado. En cualquier caso, el accuracy en test es también muy bueno (superior a todos los accuracies obtenidos con los modelos sencillos normales) por lo que a priori no supone un problema. Mostramos en la Figura (3.10) la matriz de confusión sobre el conjunto de test, para analizar los resultados. En este caso no mostramos la matriz de confusión sobre el conjunto de entrenamiento porque como el accuracy es del 100% la matriz es la identidad y no aporta información.

En la matriz de confusión sobre el conjunto de test vemos que todas las etiquetas presentan accuracies superiores al 85%, siendo la única excepción la etiqueta “si” con un accuracy del 77%. Para este caso concreto, vemos que la gran mayoría de errores vienen de fallos con respecto a la etiqueta “seis”, con una tasa del 12%. Sin embargo, este elevado error es justificable, ya que la pronunciación de ambas palabras puede considerarse

		Confusion matrix for the test set																		
		00-cero	01-uno	02-dos	03-tres	04-cuatro	05-cinco	06-seis	07-siete	08-ocho	09-nueve	10-aceptar	11-rechazar	12-arriba	13-abajo	14-izquierda	15-derecha	16-si	17-no	background
True label	00-cero	0.92	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.00		
	01-uno	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
	02-dos	0.00	0.00	0.92	0.00	0.00	0.00	0.00	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.00	0.00	
	03-tres	0.00	0.00	0.00	0.96	0.00	0.00	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
	04-cuatro	0.00	0.04	0.00	0.00	0.92	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
	05-cinco	0.00	0.00	0.00	0.00	0.00	0.92	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.08	0.00	0.00	0.00	
	06-seis	0.00	0.00	0.00	0.04	0.00	0.00	0.92	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.00	0.00	
	07-siete	0.00	0.00	0.00	0.04	0.00	0.00	0.92	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.00	0.00	
	08-ocho	0.00	0.00	0.00	0.00	0.12	0.00	0.00	0.88	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	09-nueve	0.00	0.00	0.00	0.00	0.04	0.00	0.00	0.04	0.04	0.88	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	10-aceptar	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.92	0.04	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	11-rechazar	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	12-arriba	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	13-abajo	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	14-izquierda	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
	15-derecha	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00
	16-si	0.04	0.00	0.00	0.08	0.00	0.00	0.12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.77	0.00	0.00	0.00	0.00
	17-no	0.04	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.00	0.88	0.00	0.00	0.00
	background	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00
		Predicted label																		

Figura 3.10. Matriz de confusión para el conjunto de test del modelo complejo sobre el que se ha aplicado transfer learning con el conjunto de datos generado, en español.

similar. Por otro lado, muchas de las etiquetas presentan una clasificación perfecta. El resto de etiquetas presentan accuracies entre el 88% y el 96% con tasas de fallo más o menos aceptables, menos la etiqueta “ocho” que presenta una tasa de fallo con respecto a la etiqueta “cuatro” del 12%. Esta tasa tan elevada no es fácil de justificar, ya que ambas palabras tienen pocos puntos comunes.

Como se ha comentado en el capítulo anterior, la cuantización del modelo complejo sobre el que se aplicó transfer learning fue similar a la del modelo que utilizaba los datos crudos de los audios, ya que no pudo llevarse a cabo un entrenamiento de un modelo pre-cuantizado (en este caso porque no se disponía del modelo base sin entrenar). En su lugar, se tuvo que aplicar directamente una post-cuantización utilizando un conjunto representativo generado a partir de los datos correspondientes, generando así el modelo TFLite completamente cuantizado a enteros. En la Tabla (3.5) mostramos el resumen de las accuracies que se obtuvieron sobre cada uno de los conjuntos para los 2 modelos diferentes.

Analizando la tabla, observamos que la cuantización del modelo afectó ligeramente de manera negativa al rendimiento del modelo, ya que las accuracies experimentaron un descenso similar en los conjuntos de entrenamiento y test de aproximadamente el 3%. En el caso del conjunto de entrenamiento este descenso era esperable, ya que no tendría sentido aplicar modificaciones al modelo y obtener de todos modos un accuracy

Tabla 3.5. Resumen de los accuracies obtenidos sobre cada uno de los conjuntos para los 2 modelos de la red compleja sobre la que se aplicó transfer learning.

Modelo	Accuracy en entrenamiento	Accuracy en validación	Accuracy en test
normal	100.00%	93.53%	93.49%
cuantizado	97.19%		90.34%

perfecto. Para el conjunto de test la disminución en el accuracy es notable pero no es excesivamente importante, teniendo en cuenta que el valor obtenido es superior a todos los valores de los accuracies en test obtenidos con los modelos sencillos. Podemos afirmar entonces que la cuantización del modelo proporciona buenos resultados y permite su ejecución en la placa Coral Dev (no en la TPU porque debido a los fallos en la definición dinámica de los tensores no se puede compilar el modelo) a expensas de una pequeña disminución en el rendimiento, disminución que se puede estar dispuesto a asumir sin grandes connotaciones negativas. En cualquier caso, viendo los resultados obtenidos, se puede concluir que la aplicación de transfer learning sobre el modelo complejo ha resultado un éxito, ya que los accuracies obtenidos son muy buenos, teniendo en cuenta que el conjunto de datos que se ha utilizado no era particularmente grande.

3.3. Ejecución de los modelos en la placa Coral Dev

Para terminar el capítulo, es interesante analizar cómo es la ejecución de cada uno de los modelos que hemos obtenido a lo largo del trabajo en la placa Coral Dev, con el objetivo de verificar si todo el trabajo realizado realmente se traduce en mejoras notables y significativas en el dispositivo de Tiny ML en el que queremos utilizar los modelos.

Para ello, se tomaron medidas de los tiempos de ejecución de los diferentes modelos generados para cada uno de los casos. Recordemos que en la parte de los modelos sencillos teníamos para cada tipo de información extraída de los datos un modelo TFLite normal, su correspondiente versión cuantizada y finalmente la versión compilada para ser ejecutada en la TPU. En la parte del transfer learning sobre el modelo complejo no disponemos de la versión preparada para la ejecución en la TPU, ya que el modelo no cumplía los requisitos relacionados con la definición estática de los tensores. Para cada una de las situaciones se ha calculado el tiempo de ejecución en ms de cada modelo, y por otro lado se ha añadido una estimación del tiempo requerido para procesar los datos y adecuarlos. Ambas magnitudes se han calculado como la media obtenida de diferentes ejecuciones y el error asociado viene dado por la desviación típica⁽¹²⁾. Mostramos en la Tabla (3.6) el resumen con los tiempos de ejecución y tiempos de preprocesado de cada modelo para cada caso.

⁽¹²⁾Durante la toma de los datos de los tiempos de ejecución se observó que las primeras veces que se ejecutaban determinados modelos los valores obtenidos eran muy elevados. Suponemos que esto se debe a que hay ciertos procesos que se llevan a cabo en la placa para prepararse para ejecutar los modelos, al margen de la ejecución del script desde el que se llaman. Por ello, para realizar los cálculos primero se ejecutaron todos los modelos un par de veces, con tal de asegurar que todos las medidas se realizaban en las mismas condiciones.

Tabla 3.6. Resumen de los tiempos de ejecución de los diferentes modelos en la placa Coral Dev, preparados o no para hacer uso la TPU. Se indica en una fila aparte el tiempo estimado que se tardó en realizar el preprocesado de los datos en la placa.

TFLite	Modelo CNN con los espectrogramas de Mel (ms)	Modelo CNN con los MFCCs (ms)	Modelo CNN con los datos crudos de la señal (ms)	Modelo de transfer learning en español (ms)
normal	84.2 ± 1.9	71.5 ± 1.1	151 ± 3	340 ± 20
cuantizado	71 ± 3	71.0 ± 0.6	148 ± 2	316.1 ± 1.4
edge TPU	63.3 ± 0.9	68.8 ± 1.7	152.7 ± 1.5	

tiempo de preprocesado	65 ± 5	65 ± 5	64 ± 4	74 ± 10
-------------------------------	--------	--------	--------	---------

De la tabla se pueden extraer conclusiones muy interesantes. Por un lado, vemos que el modelo convolucional para los datos como espectrogramas de Mel cumple perfectamente con lo esperado: el tiempo de ejecución del modelo normal es significativamente superior al del modelo cuantizado, y a su vez este es significativamente superior al del modelo ejecutado en la TPU. Algo similar ocurre con el modelo convolucional para los datos como MFCCs, pero en este caso la diferencia entre las ejecuciones normal y cuantizada es prácticamente despreciable, y la mejora al ejecutar el modelo en la TPU no es tan significativa. Por otro lado, los resultados del modelo convolucional para los datos crudos de los audios resultan negativamente sorprendentes. Los tiempos para los diferentes modelos son muy similares, pero son un orden de magnitud superior que en los casos anteriores. Se aprecia una ligera mejora entre el modelo normal y el modelo cuantizado, pero el modelo ejecutado en la TPU es peor incluso que el modelo normal, seguramente debido a procesos internos que suceden en la placa a raíz de que solo 1 de las operaciones se puede llevar a cabo en la TPU, mientras que el resto se deben realizar en la CPU. Finalmente, el modelo complejo sobre el que se aplica transfer learning es el que peores tiempos de ejecución tiene, con mucha diferencia, seguramente por la cantidad de operaciones que se deben ejecutar al no cumplir con las restricciones impuestas por la placa. Esto nos muestra lo importante que es cumplir con los requisitos para una buena ejecución, ya que el no cumplirlos se traduce en penalizaciones importantes y nada despreciables. Sí que se aprecia una disminución significativa entre los tiempos del modelo normal y el cuantizado.

Fijándonos en los tiempos de procesado, vemos que en general no suponen un factor relevante a tener en cuenta, ya que las diferencias que se obtienen para los diferentes casos son mínimas.

Por último, mostramos en el Anexo (A3) el código que se programó para ejecutar los diferentes modelos en la placa Coral Dev, y en el que se puede ver cómo se realiza el preprocesado y la inferencia para cada caso, llevando a cabo la creación del intérprete, la asignación de los tensores, la cuantización de los datos y la ejecución del modelo.

Capítulo 4

Conclusiones

Este trabajo se puede considerar como un punto de entrada a las ejecuciones de modelos de machine learning en dispositivos pequeños que no tienen todas las prestaciones que pueden ofrecer ordenadores y servidores grandes y potentes. Esto es lo que se conoce como Tiny ML, y busca explotar los beneficios que ofrece el Edge Computing, la filosofía que se basa en que los datos no tengan que salir del dispositivo en el que se recogen y que todos los cálculos, procesos y demás operaciones sobre los datos se puedan realizar en él. Es cierto que este planteamiento proporciona grandes ventajas, sobre todo en cuanto a la seguridad de los datos y la dependencia de una conexión a Internet, pero requiere que se desarrollen los modelos siguiendo unas pautas determinadas para su correcta ejecución.

El Tiny ML se puede aplicar en multitud de áreas de la inteligencia artificial, como puede ser todo el campo relacionado con imágenes o visión por computador, pero en este caso nos hemos centrado en el campo del audio y el reconocimiento del lenguaje. Más concretamente, nos hemos centrado en una rama particular del reconocimiento de lenguaje: el reconocimiento y detección de palabras de activación.

Se ha decidido estudiar y abordar el problema en 2 partes. Utilizando la versión 1 del conjunto de datos *speech commands* se han generado modelos sencillos basados en redes neuronales, cada una de ellas con estructuras determinadas que respondían mejor al tipo de información que se había extraído de los datos (espectrogramas de Mel, MFCCs o datos crudos de los audios). Por otro lado, se ha generado un conjunto de datos similar pero en español que se ha utilizado para, aplicando transfer learning sobre un modelo neuronal complejo, verificar si el modelo es capaz de aprender y detectar las palabras de activación en otro idioma diferente que el del utilizado en el entrenamiento inicial. Con todo lo realizado, se han extraído las siguientes conclusiones:

- como en cualquier problema de datos, el conjunto de datos que se utilice es esencial para obtener unos buenos resultados. Realizar un correcto tratamiento y limpieza es fundamental. En el tratamiento de los datos puede ser interesante aplicar data augmentation en caso de que el conjunto no sea especialmente grande, ya que ello enriquece el conjunto final y puede aportar mejoras interesantes. Aún así, siempre será mejor disponer de un conjunto de partida más grande, diverso y variado

- el enfoque que se le quiera dar al problema y el preprocessado que se quiera llevar a cabo en los datos es muy relevante. En el caso del reconocimiento del lenguaje es muy importante realizar una transformación de los datos y extraer de ellos información particular sobre la que aplicar los diferentes modelos. En nuestro caso, para explotar la potencia de las redes convolucionales, se han transformado los datos en espectrogramas de Mel y MFCCs para poder tratarlos como imágenes, aunque también se han utilizado los datos crudos porque el estado del arte se está centrando en este enfoque
- particularizando para el caso de ejecutar modelos en dispositivos de Tiny ML, el proceso a seguir y el cumplimiento de las restricciones impuestas por el dispositivo son fundamentales para obtener buenos resultados. Las TPUs de los dispositivos requieren que los modelos estén cuantizados. La cuantización de los modelos puede afectar negativamente a su rendimiento (aunque esto depende del tipo particular del problema que se esté tratando). En el caso concreto de reconocimiento de lenguaje se ha observado que el poder preparar el modelo para la cuantización (pre-cuantización) es importante, ya que los modelos cuantizados directamente (post-cuantización) presentan accuracies significativamente inferiores
- en los modelos sencillos generados se han obtenido accuracies entre el 87% y el 88% para los modelos TFLite no cuantizados. El proceso de cuantización empeora un poco los resultados, mostrando que el mejor modelo es el que utiliza los MFCCs, quedándose éste con un accuracy del 86.23% en su versión cuantizada
- el transfer learning ha resultado ser una forma de proceder muy acertada, obteniendo muy buenos resultados. Con un conjunto de datos no excesivamente grande con el que reentrenar el modelo, se han obtenido accuracies superiores al 90% (con el modelo normal y cuantizado), demostrando que el modelo ha aprendido a reconocer las palabras de activación en un idioma diferente al idioma de entrenamiento inicial
- por último, se ha descubierto la importancia y relevancia que puede tener a corto y medio plazo el Tiny ML, ya que se pueden encontrar aplicaciones muy diversas en las que juegue un papel fundamental. Es cierto que el proceso puede ser un poco complicado, y en ocasiones limitado por los requisitos que se deben cumplir, pero aparecen muchas opciones para llevar a cabo estudios y generar modelos con los que resolver problemas en los que el tiempo de ejecución sea importante, ya que se ha verificado que la ejecución de los modelos en la TPU puede ser significativamente más rápida (siempre y cuando el modelo lo permita)

4.1. Trabajo futuro: mejoras y ampliaciones

Este trabajo ha abarcado muchos enfoques y procedimiento diferentes, por lo que las mejoras y ampliaciones que se pueden llevar a cabo son numerosas. Centrándose en la parte del trabajo que comprende el tratamiento de los datos y el diseño y entrenamiento de los modelos, hay varios puntos sobre los que se podría iterar:

- en primer lugar, todos los modelos se han entrenado con la versión 1 del conjunto de datos *speech commands* debido a limitaciones en la gestión de la memoria y la potencia del hardware disponible. Esta versión contiene menos datos y con una peor calidad, así que se podría repetir el trabajo utilizando la versión 2 y ver si los modelos presentan un mejora significativa
- siguiendo con el tratamiento del conjunto de datos, se podrían aplicar más técnicas de data augmentation útiles en el caso de reconocimiento de lenguaje, ampliando aún más el conjunto para ver si se obtienen mejores resultados
- en la parte de los modelos creados desde 0, se han diseñado 3 redes sencillas que responden cada una a una información concreta de los datos: espectrogramas de Mel, MFCCs y datos crudos. Se podría estudiar cómo afectan las variables utilizadas al transformar los datos en este tipo de información: por ejemplo, se podría analizar cómo influye la cantidad de bandas de Mel generadas en los espectrogramas de Mel, el número de MFCCs escogidos, o incluso la frecuencia de muestreo de los datos crudos de los audios. Siguiendo en esta misma línea, se podrían investigar nuevas características de este tipo que se pudiesen extraer de los audios, con el objetivo de ver si hay otras características más adecuadas para problemas de reconocimiento de lenguaje en forma de clasificación de palabras de activación
- particularizando en los modelos sencillos creados, se podrían probar estructuras diferentes, aumentando la complejidad de la red, intentando buscar modelos que entrenen mejor y en general proporcionen mejores resultados. En el caso particular del modelo que utiliza los datos crudos de los audios, se podría buscar una estructura alternativa que no sobreajuste y que tenga capas que cumplen las limitaciones de la placa Coral DevMF
- por último, en el caso del transfer learning uno de los problemas más grandes es que no se ha podido cuantizar bien el modelo y compilarlo para ejecutarlo en la TPU porque se ha utilizado un modelo ya entrenado y que no se podía modificar. Entendiendo cuál ha sido el proceso para llevar a cabo ese entrenamiento y la estructura concreta del modelo, se podría intentar replicar dicho entrenamiento creando el modelo desde 0 y, aplicando posteriormente transfer learning como se ha hecho, analizar si el modelo presenta mejores resultados y si es posible ejecutarlo en la TPU como se ha hecho con el resto de casos

Por otro lado, centrándose en la ejecución de los modelos finales en la placa Coral Dev, hay mejoras interesantes que se podrían desarrollar:

- se ha conseguido que la placa procese audios previamente grabados e introducidos en ella, pero sería muy interesante llevar a cabo el desarrollo de un script más elaborado que se quedase escuchando con el micrófono activado, procesando y analizando las diferentes ventanas de audio de 1 segundo para reaccionar cuando detecte alguna palabra de activación
- siguiendo con el planteamiento anterior, también se podría desarrollar un programa que llevase a cabo diferentes acciones dependiendo de la palabra de activación que se

haya detectado. Esto abre un campo enorme de opciones donde aplicar lo realizado en el trabajo, ya que por ejemplo se podría desarrollar una interfaz que llevase a cabo determinadas acciones en un ámbito concreto según las palabras utilizadas. Por poner algún ejemplo, utilizando los números se podría subir o bajar el volumen de un altavoz, aumentar o disminuir la potencia de una bombilla inteligente, etc.

Por último, simplemente mencionar que TensorFlow pone a disposición del público muchos modelos que ya han sido previamente entrenados y que se pueden ejecutar en la placa Coral Dev. Además de esto, también ofrece diversos scripts para llevar a cabo transfer learning sobre algunos de esos modelos, por lo que se podrían realizar proyectos similares a la segunda parte de este trabajo, pero con otras finalidades. Por ejemplo, hay otro script basado en audio que en lugar de centrarse en reconocimiento de lenguaje, lo que hace es clasificar diversos ruidos. Aplicar transfer learning sobre un modelo de este estilo podría ser muy interesante para, por ejemplo, crear un modelo que reaccione cuando suena un timbre (y que mande una notificación para saber que están llamando a la puerta) o cuando se oigan sonidos de cristales rotos (que en ciertos casos se podría considerar como una alarma).

Bibliografía

- [1] Alexandre Gonfalonieri. [How Amazon Alexa works? Your guide to Natural Language Processing \(AI\)](#). Towards Data Science. Consultado en agosto, 2022
- [2] Labied, M., Belangour, A. (2021). Automatic speech recognition features extraction techniques: a multi-criteria comparison. International Journal of Advanced Computer Science and Applications [10.14569/IJACSA.2021.0120821](https://doi.org/10.14569/IJACSA.2021.0120821)
- [3] mlearnere. [Learning from Audio: The Mel Scale, Mel Spectrograms, and Mel Frequency Cepstral Coefficients](#). Towards Data Science. Consultado en agosto, 2022
- [4] Valerio Velardo. (2022) Mel-frequency cepstral coefficients explained easily. YouTube. [The sound of AI](#). Visitado en agosto, 2022
- [5] Swamy, S., Ramakrishnan, K.V. (2013). Evolution of speech recognition: a brief history of technology development.
- [6] Página web de Sonix. [A short history of speech recognition](#). Consultada en agosto, 2022
- [7] Pratap, V., Hannun, A., Xu, Q., Cai, J., Kahn, J., Synnaeve, G., Liptchinsky, V., Collobert, R. (2019). Wav2Letter++: a fast open-source speech recognition system. [10.1109/ICASSP.2019.8683535](https://doi.org/10.1109/ICASSP.2019.8683535)
- [8] Baevski, A., Zhou, H., Mohamed, A., Auli, M. (2020). wav2vec 2.0: a framework for self-supervised learning of speech representations. arXiv. [10.48550/ARXIV.2006.11477](https://doi.org/10.48550/ARXIV.2006.11477)
- [9] Hannun, A., Case, C., Casper, J., Catanzaro, B., Diamos, G., Elsen, E., Prenger, R., Satheesh, S., Sengupta, S., Coates, A., Ng, A. (2014). Deep Speech: scaling up end-to-end speech recognition. arXiv. [10.48550/ARXIV.1412.5567](https://doi.org/10.48550/ARXIV.1412.5567)
- [10] Calvete, D., Soria, E., Martínez, J.M., Gómez J., Martínez, M., Dimas, P., Barranquero, R. (2021). Tiny ML: la nueva revolución en la IoT
- [11] Página web de [TensorFlow](#). Consultada en agosto, 2022
- [12] Página web de [Keras](#). Consultada en agosto, 2022
- [13] Página web de la [Coral Dev Board](#), junto con todos los enlaces relevantes que se redirigen desde la misma. Consultada en agosto, 2022
- [14] Página web de [BalenaEtcher](#). Consultada en agosto, 2022

- [15] Warden, P. (2018). Speech commands: a dataset for limited-vocabulary speech recognition. Google Brain. arXiv. <https://doi.org/10.48550/arXiv.1804.03209>
- [16] Fukuda, T., Fernandez, R., Rosenberg, A., Samuel, T., Ramabhadran, B., Sorin, A., Kurata, G. (2018). Data augmentation improves recognition of foreign accented speech. IBM Research AI. Google. <10.21437/Interspeech.2018-1211>
- [17] Park, D.S., Chan, W., Zhang, Y., Chiu, C-C., Zoph, B., Cubuk, E.D., Le, Q.V. (2019). SpecAugment: a simple data augmentation method for automatic speech recognition. Google Brain. arXiv. <10.21437/interspeech.2019-2680>
- [18] McFee, B., Raffel, C., Liang, D., Ellis, D.P.W., McVicar, M., Battenber, E., Nieto, O. (2015). Librosa: audio and music signal analysis in Python. En las actas de la 14^a conferencia de Python en la ciencia, páginas 18-25. <https://zenodo.org/badge/latestdoi/6309729>
- [19] Lutz, R. (2017). Netron: visualizer for neural networks, deep learning and machine learning models. [Github](#) del proyecto. <10.5281/zenodo.5854962>. Consultado en agosto, 2022
- [20] [Compilador web para preparar modelos para su ejecución en la Edge TPU utilizando Google Colab](#). Consultado en agosto, 2022
- [21] Dai, W., Dai, C., Qu, S., Li, J., Das, S. (2016). Very deep convolutional neural networks for raw waveforms. arXiv. <https://doi.org/10.48550/arXiv.1610.00087>
- [22] [Retrain a speech recognition model with TensorFlow Lite Model Maker](#). TensorFlow for mobile & edge. Consultado en agosto, 2022

Anexos

A1. Ejemplos de la forma de onda y spectrograma de Mel de varios audios, agrupados por etiqueta

En las Figuras (A1.1 - A1.18) mostramos la forma de onda y el spectrograma de Mel de tres muestras escogidas aleatoriamente de cada tipo de etiqueta. Visualmente se pueden apreciar ciertas similitudes entre las diferentes muestras de una misma etiqueta, a la vez que se pueden encontrar diferencias apreciables entre algunas etiquetas diferentes.

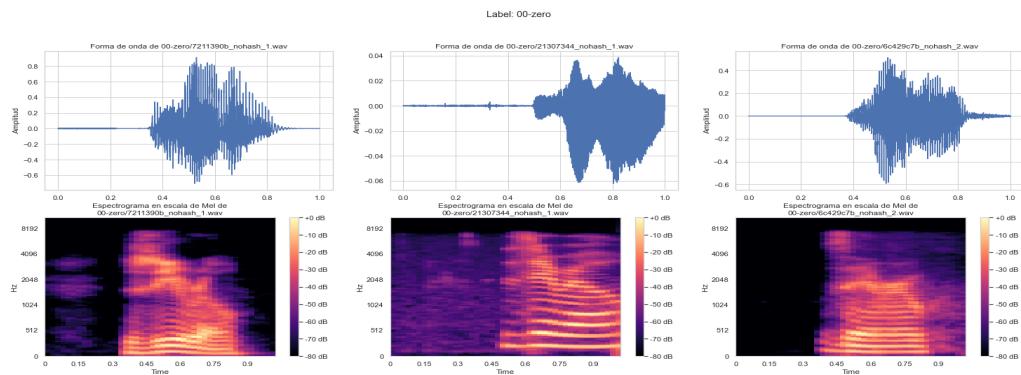


Figura A1.1. Forma de onda y spectrograma de Mel de algunas muestras de la etiqueta “zero”.

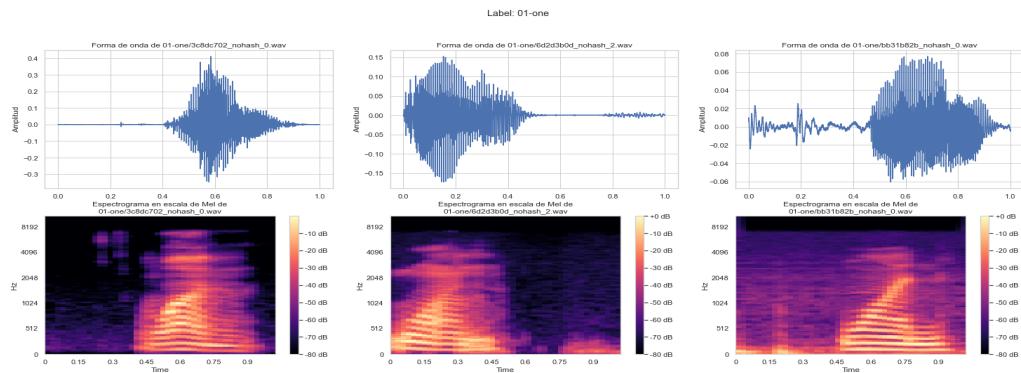


Figura A1.2. Forma de onda y spectrograma de Mel de algunas muestras de la etiqueta “one”.

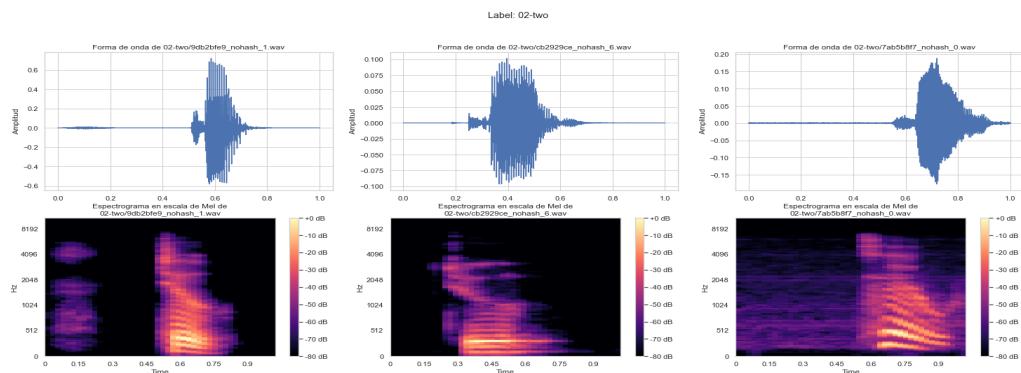


Figura A1.3. Forma de onda y spectrograma de Mel de algunas muestras de la etiqueta “two”.

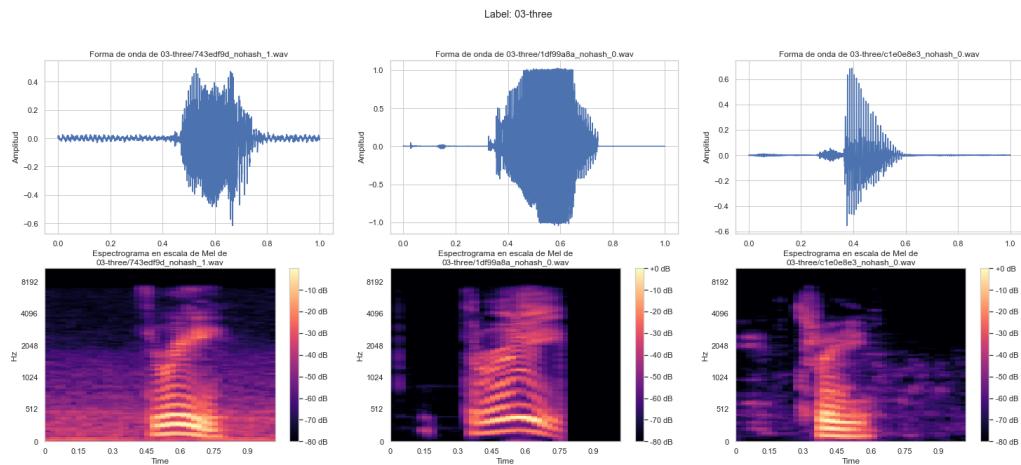


Figura A1.4. Forma de onda y espectrograma de Mel de algunas muestras de la etiqueta “three”.

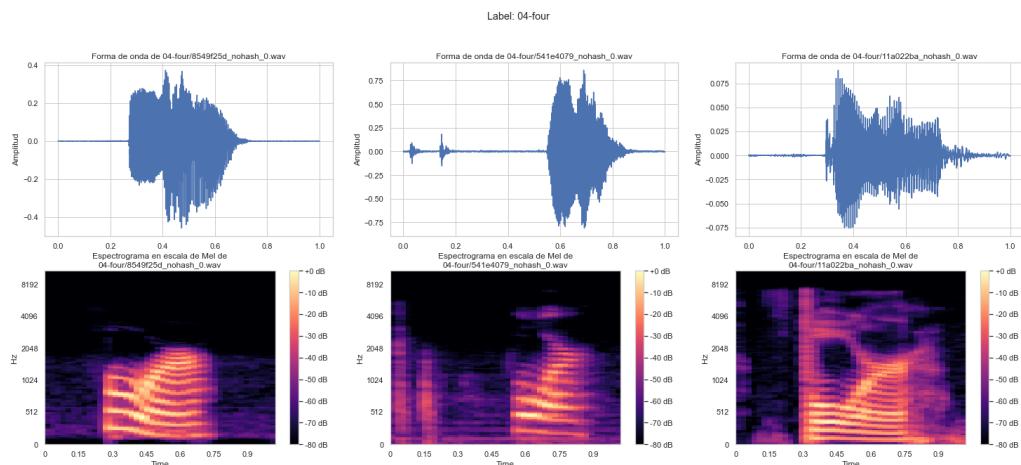


Figura A1.5. Forma de onda y espectrograma de Mel de algunas muestras de la etiqueta “four”.

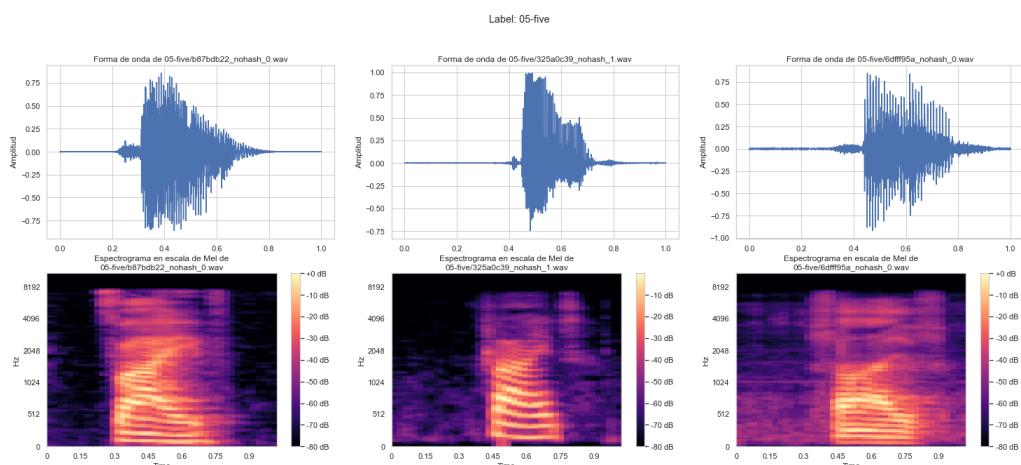


Figura A1.6. Forma de onda y espectrograma de Mel de algunas muestras de la etiqueta “five”.

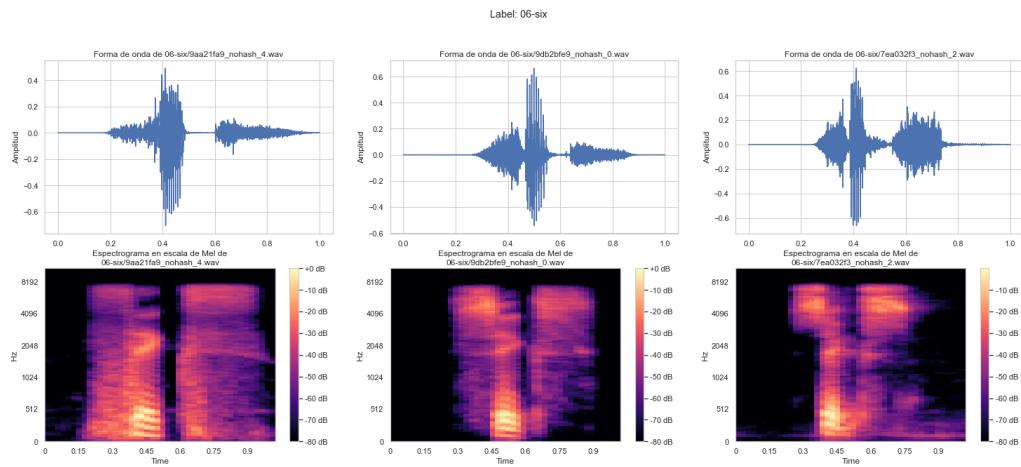


Figura A1.7. Forma de onda y espectrograma de Mel de algunas muestras de la etiqueta “six”.

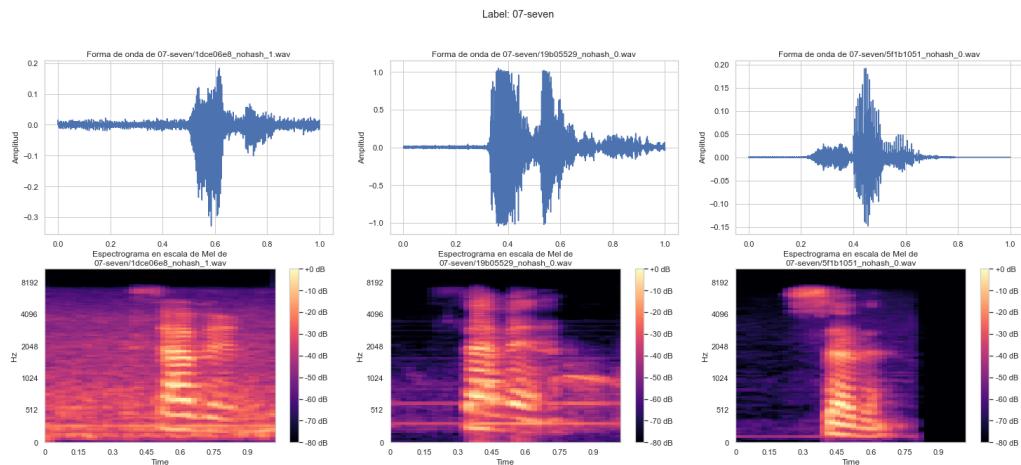


Figura A1.8. Forma de onda y espectrograma de Mel de algunas muestras de la etiqueta “seven”.

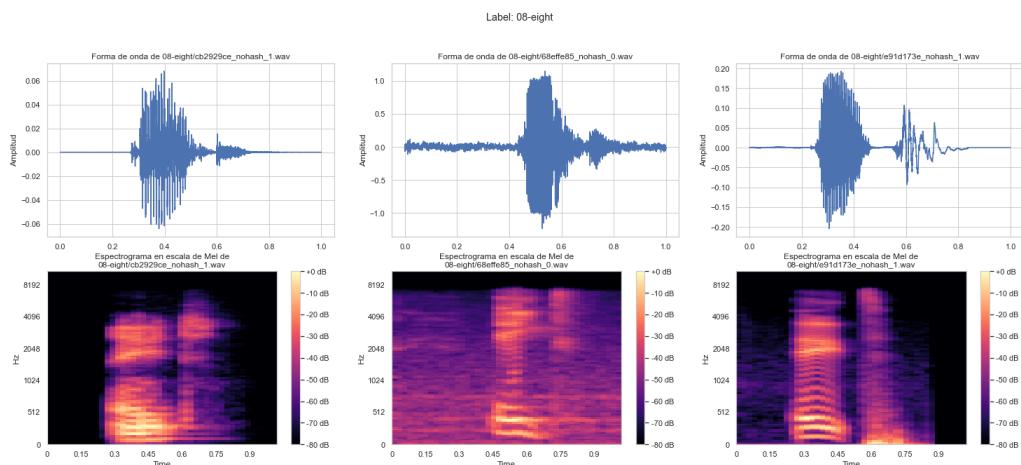


Figura A1.9. Forma de onda y espectrograma de Mel de algunas muestras de la etiqueta “eight”.

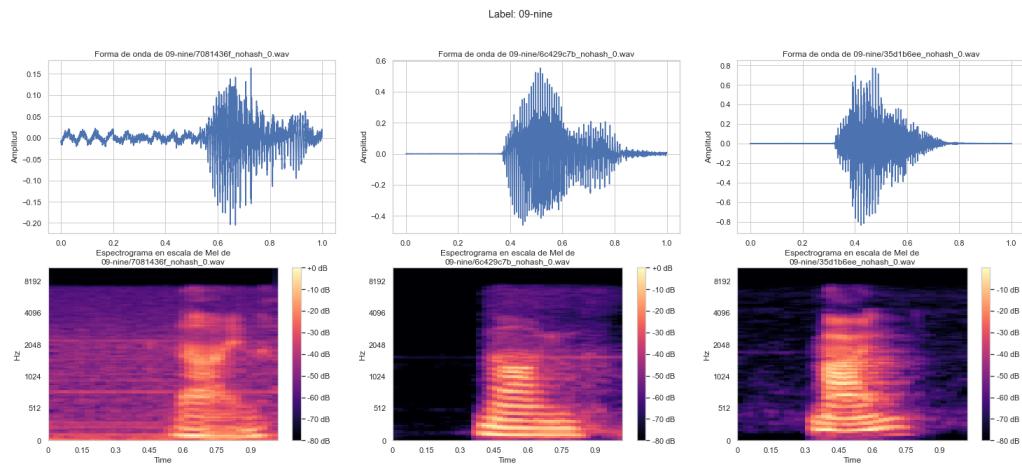


Figura A1.10. Forma de onda y espectrograma de Mel de algunas muestras de la etiqueta “nine”.

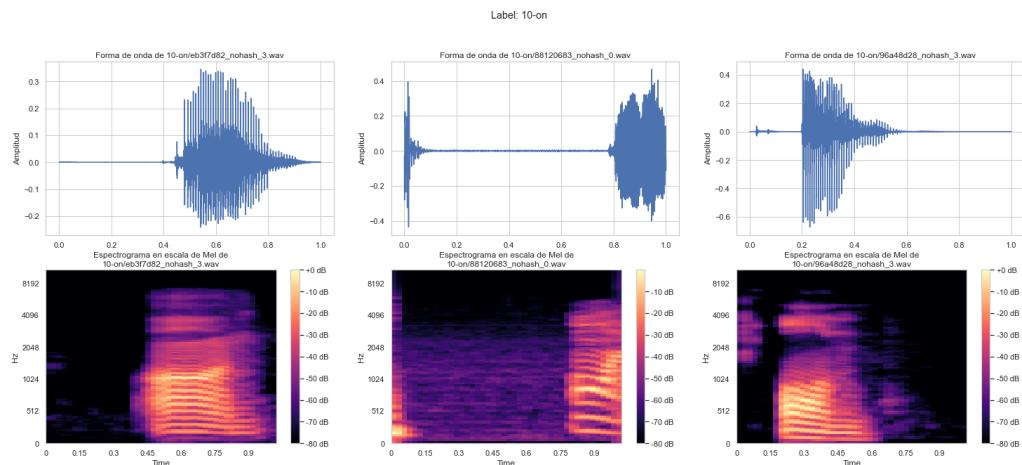


Figura A1.11. Forma de onda y espectrograma de Mel de algunas muestras de la etiqueta “on”.

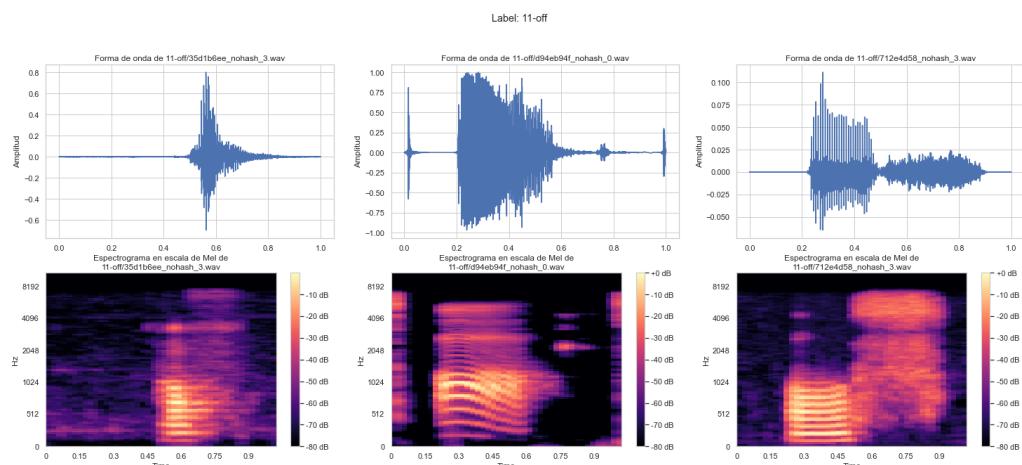


Figura A1.12. Forma de onda y espectrograma de Mel de algunas muestras de la etiqueta “off”.

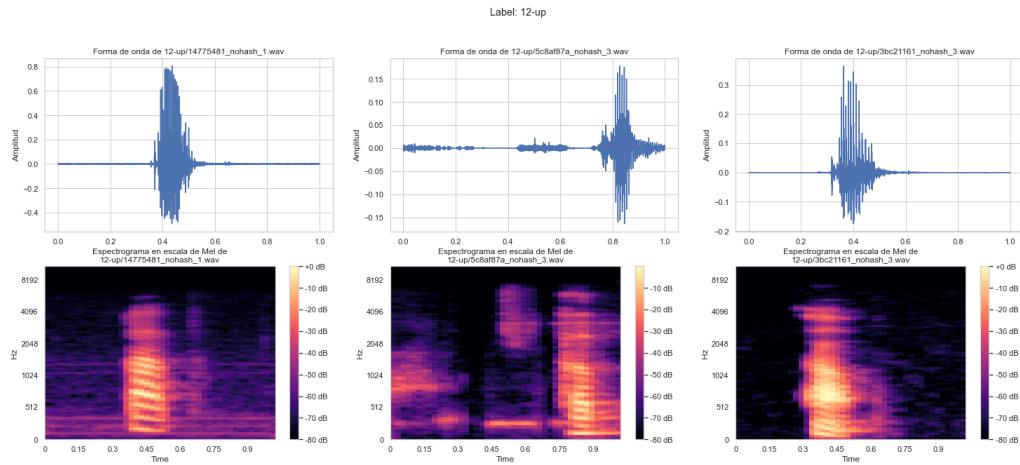


Figura A1.13. Forma de onda y espectrograma de Mel de algunas muestras de la etiqueta “up”.

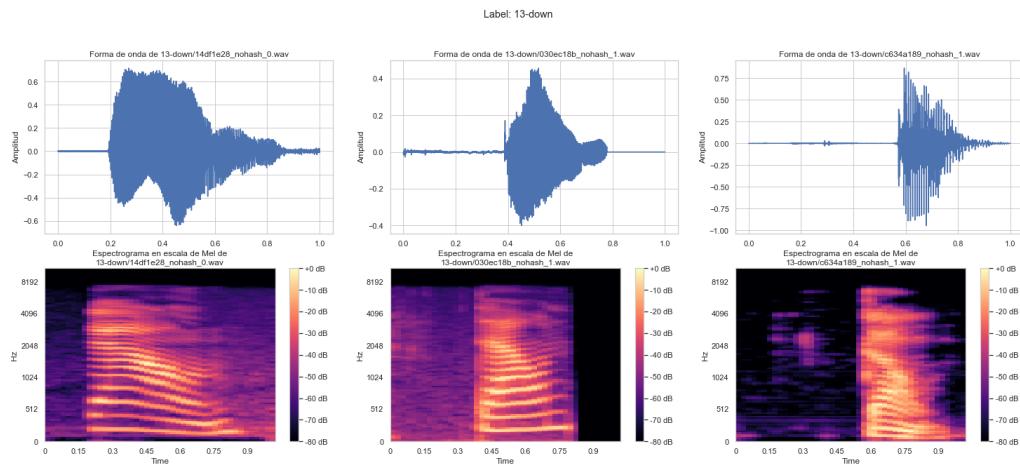


Figura A1.14. Forma de onda y espectrograma de Mel de algunas muestras de la etiqueta “down”.

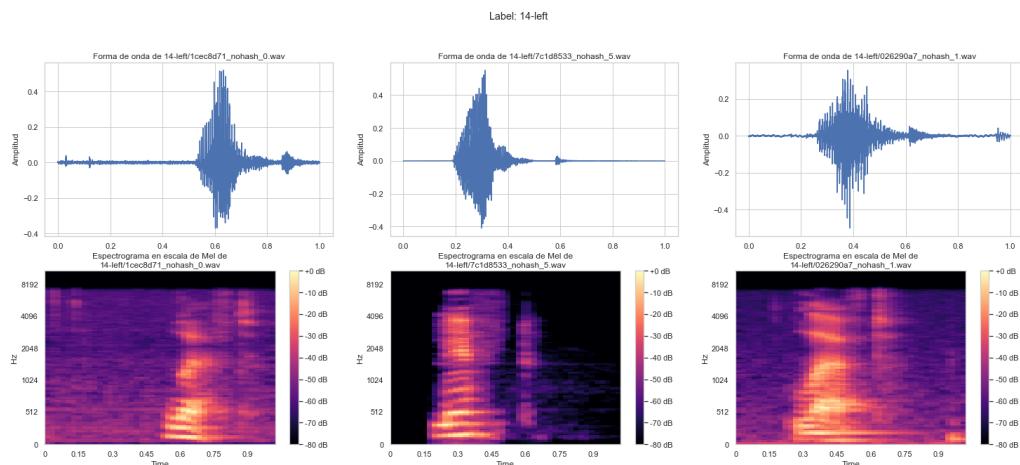


Figura A1.15. Forma de onda y espectrograma de Mel de algunas muestras de la etiqueta “left”.

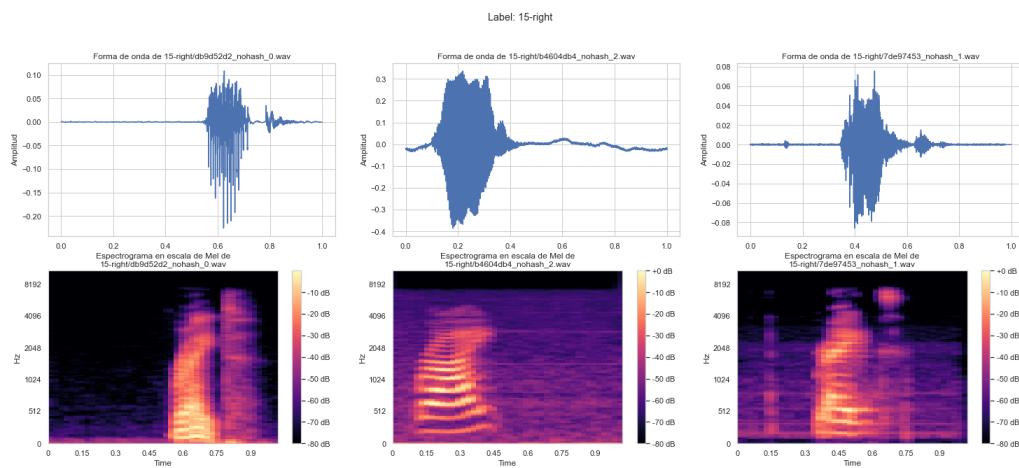


Figura A1.16. Forma de onda y espectrograma de Mel de algunas muestras de la etiqueta “right”.

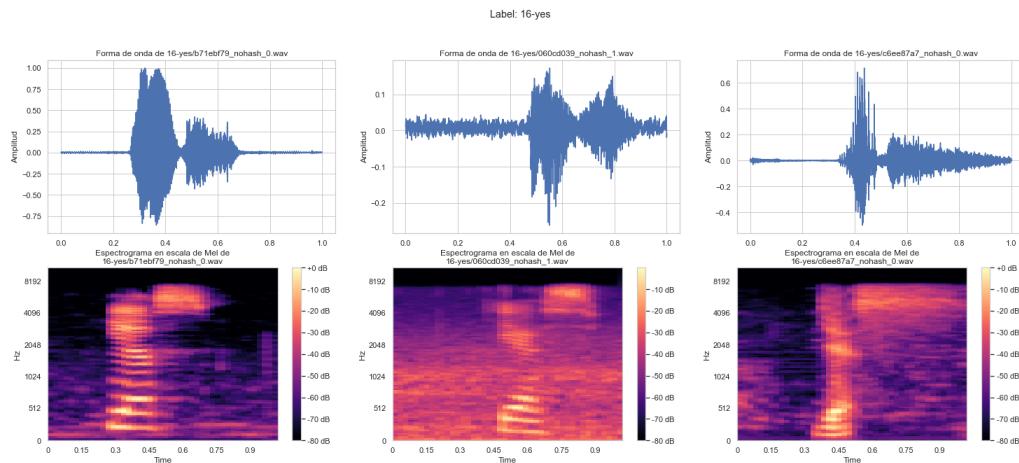


Figura A1.17. Forma de onda y espectrograma de Mel de algunas muestras de la etiqueta “yes”.

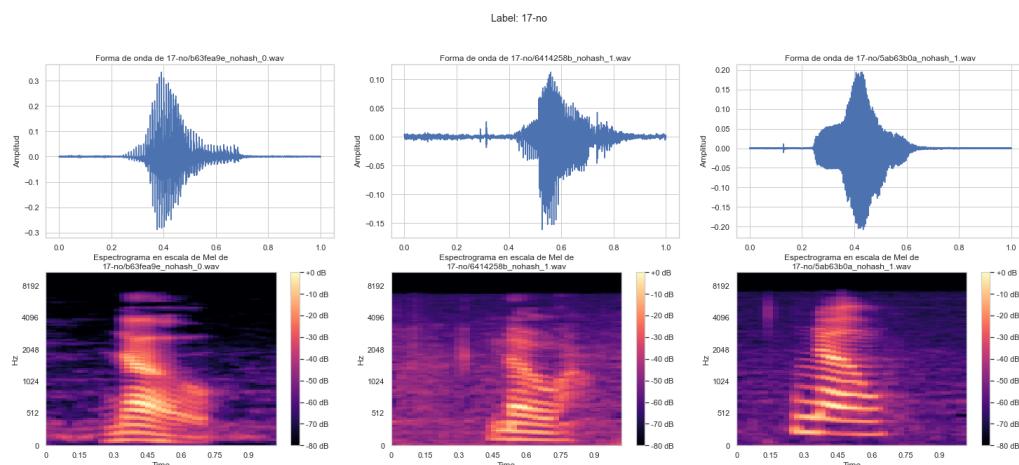


Figura A1.18. Forma de onda y espectrograma de Mel de algunas muestras de la etiqueta “no”.

A2. Gráficos de las estructura de los modelos utilizados

En primer lugar, se presentan en la Figura (A2.1) los gráficos de las estructuras de la red convolucional 2D utilizada con los datos en forma de spectrograma de Mel. En la Figura (A2.1a) se ve el gráfico del modelo normal, mientras que en la Figura (A2.1b) se observa el gráfico del modelo cuantizado, generado utilizando la herramienta Netron.

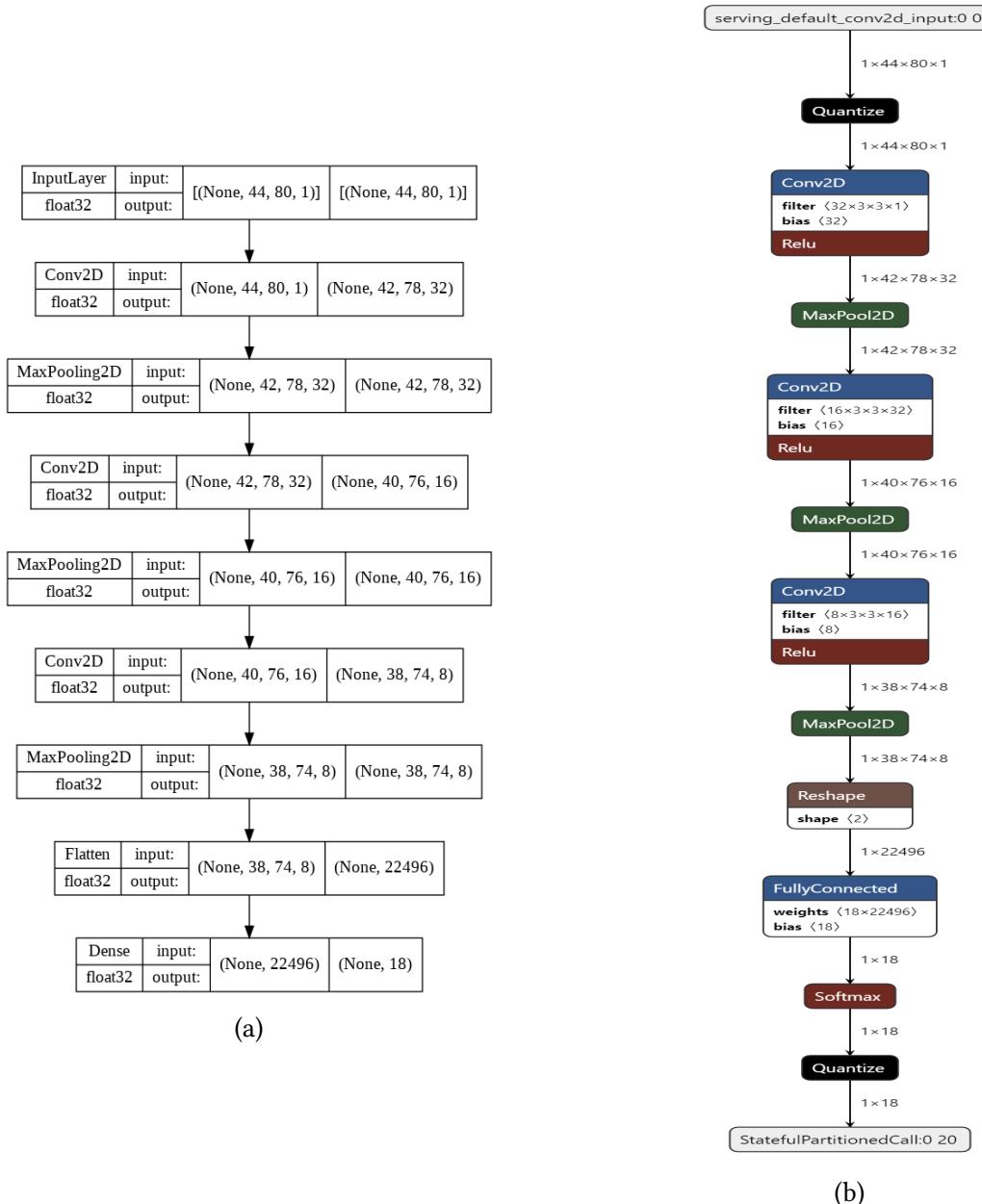
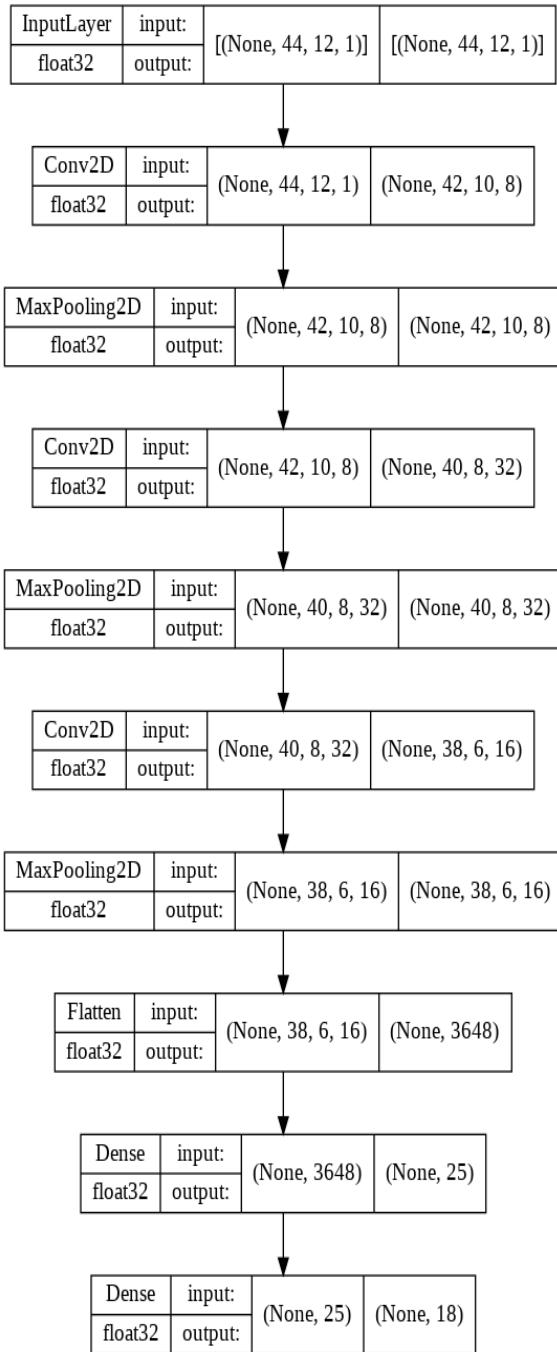
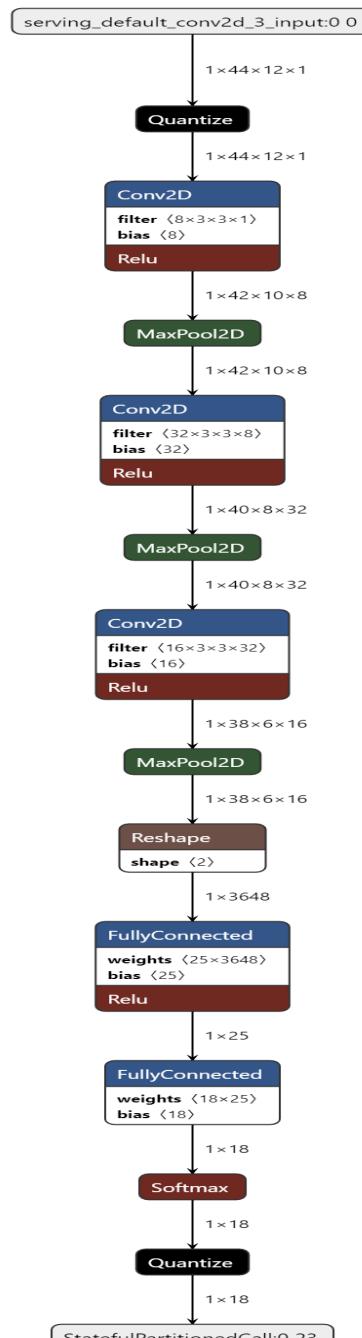


Figura A2.1. Gráficos de las estructuras de la red convolucional 2D utilizada con los datos en forma de spectrograma de Mel. En la Figura (A2.1a) se ve el gráfico del modelo normal, mientras que en la Figura (A2.1b) se observa el gráfico del modelo cuantizado generado utilizando la herramienta Netron.

En segundo lugar, se presentan en la Figura (A2.2) los gráficos de las estructuras de la red convolucional 2D utilizada con los datos en forma de MFCCs. En la Figura (A2.2a) se ve el gráfico del modelo normal, mientras que en la Figura (A2.2b) se observa el gráfico del modelo cuantizado, generado utilizando la herramienta Netron.



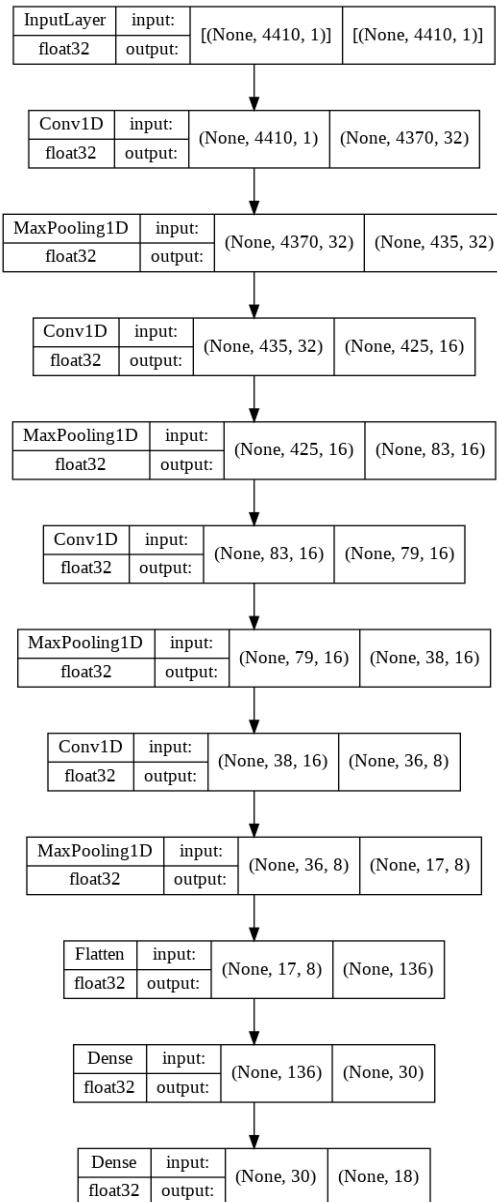
(a)



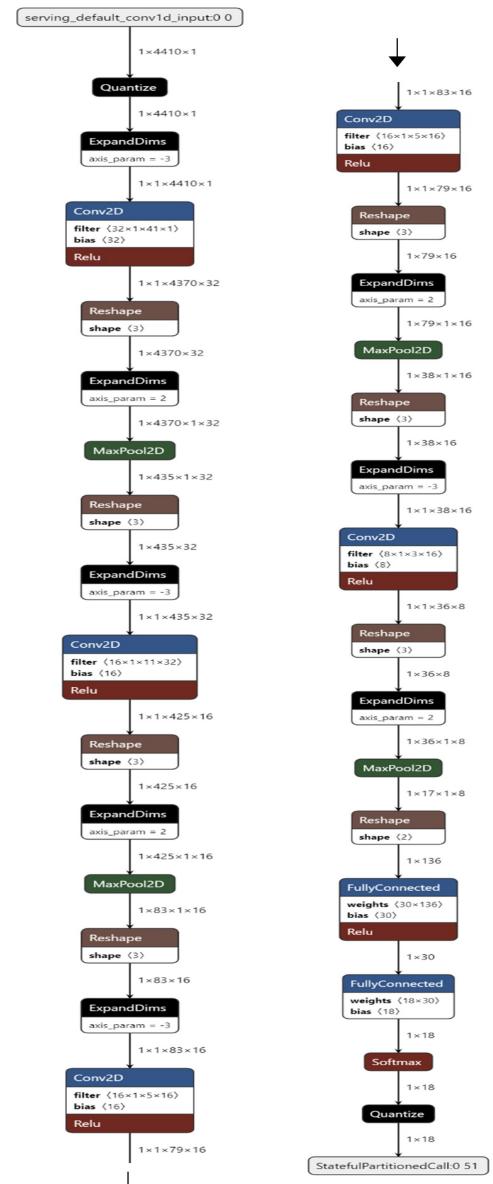
(b)

Figura A2.2. Gráficos de las estructuras de la red convolucional 2D utilizada con los datos en forma de MFCCs. En la Figura (A2.2a) se ve el gráfico del modelo normal, mientras que en la Figura (A2.2b) se observa el gráfico del modelo cuantizado generado utilizando la herramienta Netron.

A continuación, se presentan en la Figura (A2.3) los gráficos de las estructuras de la red convolucional 1D utilizada con los datos crudos de la señal. En la Figura (A2.3a) se ve el gráfico del modelo normal, mientras que en la Figura (A2.3b) se observa el gráfico del modelo cuantizado, generado utilizando la herramienta Netron. En este caso, vemos que este último gráfico representa una red notablemente más grande que el modelo normal asociado, debido a que las capas convolucionales 1D no se pueden convertir directamente a formato .tflite, por lo que se han tenido que recodificar y transformar en otras capas que realicen una función equivalente pero que sí que estén permitidas.



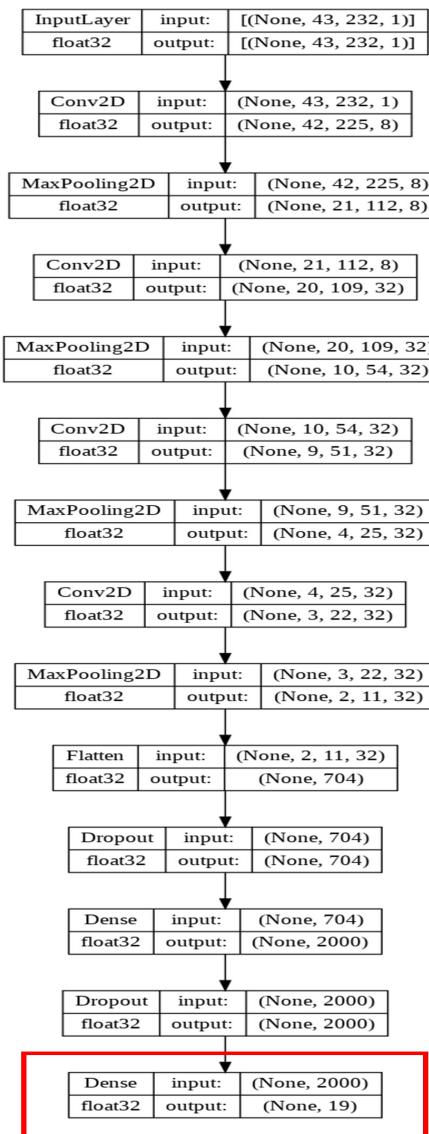
(a)



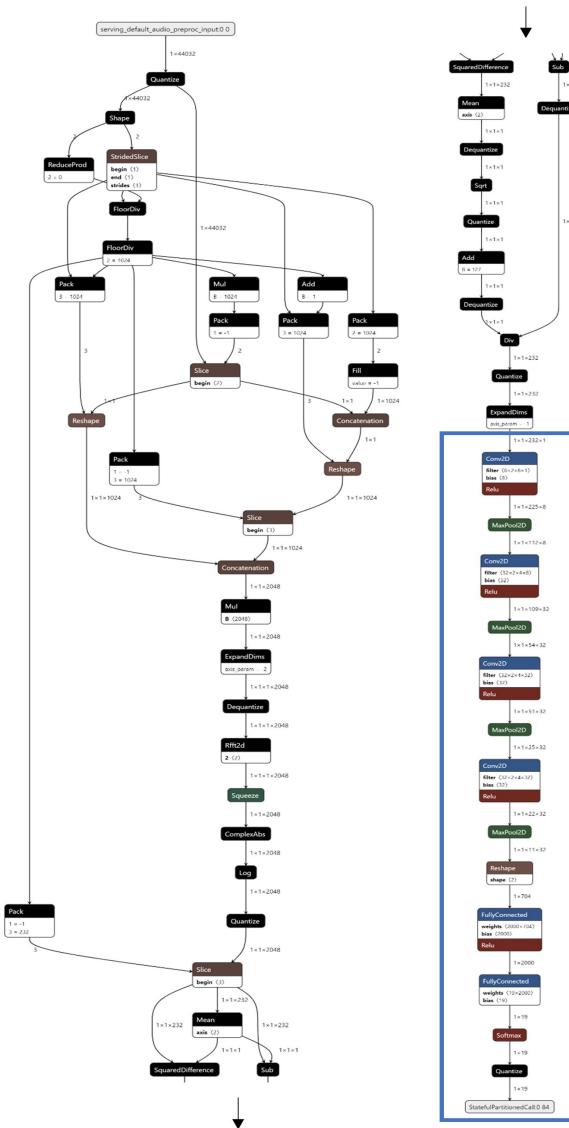
(b)

Figura A2.3. Gráficos de las estructuras de la red convolucional 1D utilizada con los datos crudos de la señal. En la Figura (A2.3a) se ve el gráfico del modelo normal, mientras que en la Figura (A2.3b) se observa el gráfico del modelo cuantizado, generado utilizando la herramienta Netron.

Por último, se presentan en la Figura (A2.4) los gráficos de las estructuras de la red compleja sobre la que se ha aplicado transfer learning con datos en español. En la Figura (A2.4a) se ve el gráfico del modelo normal, mientras que en la Figura (A2.4b) se observa el gráfico del modelo cuantizado generado utilizando la herramienta Netron. En el modelo normal se observa un recuadro rojo que marca la capa con los pesos reentrenables y con la que se realizó el transfer learning. Por otro lado, el gráfico generado con Netron muestra una red mucho más compleja que la normal, porque previamente al modelo en sí realiza muchas operaciones. En esta imagen se observa un recuadro azul que marca la parte de la red que se corresponde con la imagen del modelo normal.



(a)



(b)

Figura A2.4. Gráficos de las estructuras de la red compleja sobre la que se ha aplicado transfer learning con datos en español. En la Figura (A2.4a) se ve el gráfico del modelo normal, mientras que en la Figura (A2.4b) se observa el gráfico del modelo cuantizado generado utilizando la herramienta Netron.

A3. Código para la ejecución de modelos TFLite en la placa Coral

En las siguientes páginas se muestra el código desarrollado para llevar a cabo la ejecución de los modelos generados durante todo el trabajo en la placa Coral Dev. El script incluye todo el proceso que se sigue desde la lectura de los datos de un audio aleatorio hasta la clasificación que hace el modelo escogido de la palabra de activación que se dice en el audio.

El script está escrito de forma general para todos los casos, dando la opción al usuario que lo ejecuta de escoger el modelo que quiere utilizar, por medio de un input que actúa como selector. Una vez una selección válida se ha introducido, está hardcodeado en el propio script qué archivos se deben utilizar y dónde se debe buscar el audio aleatorio que se va a clasificar. La salida que ofrece el script no es más que la predicción de la clasificación realizada sobre el audio seleccionado, mostrando también qué audio se ha escogido para poder verificar si la predicción es correcta o no. También se muestra el tiempo de ejecución del modelo y el tiempo total de ejecución incluyendo el preprocesado del audio.

Algunas aclaraciones importantes:

- para la correcta ejecución y funcionamiento del código, es necesario que los modelos estén en un directorio “models”, que haya un directorio “data” con los dos subdirectorios “eng” y “esp” donde estén respectivamente los audios en formato .wav de entre los que se seleccionará uno aleatoriamente en cada ejecución y que estén presentes los archivos .txt con las etiquetas ordenadas con las que se han entrenado los modelos
- los audios no se han preparado ni preprocesado previamente (intentando simular que podrían ser audios que se han grabado desde la propia placa), por lo que también se han incluido las funciones que estandarizan los audios. Por un lado está la función que aplica padding con silencio hasta que el audio dure exactamente 1 segundo, y por otro lado está la función que selecciona la mejor ventana de 1 segundo de duración, utilizando el RMS para asegurarse de que se captura la mayor cantidad de audio relevante. Una vez el audio se ha procesado, se guarda en el directorio de ejecución para ser leído posteriormente del modo que corresponda
- con respecto a la ejecución de los modelos, se puede observar en el código cómo se lleva a cabo la inferencia creando el intérprete, asignando los tensores, adecuando y cuantificando los datos como corresponda según cada modelo y, finalmente, ejecutando cada modelo. Como pequeño matiz, es interesante ver cómo la ejecución del modelo en la TPU requiere que el intérprete se genere con parámetros extra propios del dispositivo en el que se está realizando la ejecución

```

# IMPORT PACKAGES AND LIBRARIES
import numpy as np
import random
import os
from time import time

# Audio specifics
import soundfile as sf
from pydub import AudioSegment
from librosa import load, power_to_db
from librosa.feature import rms, melspectrogram, mfcc

# Coral Dev Board Specifics
# import tflite_runtime.interpreter as tflite # to make tests on laptop
from pycoral.utils import edgetpu
#from pycoral.utils import dataset
#from pycoral.adapters import common
#from pycoral.adapters import classify

# GLOBAL VARIABLES
MODEL_FILE = [
    "./models/CNN_melspecs.tflite",
    "./models/CNN_melspecs_model_quantized_metadata.tflite",
    "./models/CNN_melspecs_model_quantized_metadata_edgetpu.tflite",
    "./models/CNN_MFCCs.tflite",
    "./models/CNN_MFCCs_model_quantized_metadata.tflite",
    "./models/CNN_MFCCs_model_quantized_metadata_edgetpu.tflite",
    "./models/CNN_raws.tflite",
    "./models/CNN_raws_model_quantized_metadata.tflite",
    "./models/CNN_raws_model_quantized_metadata_edgetpu.tflite",
    "./models/TL_model_browserfft_speech.tflite",
    "./models/TL_model_quantized_ESP_MM_metadata.tflite",
]
DATA_PATH = ["./data/eng", "./data/esp"]
LABELS_FILE = ["./labels_ENG.txt", "./labels_ESP.txt"]
COMMON_SAMPLE_RATE_LIBROSA = 22050
INPUT_SAMPLE_RATE = 44100
DOWNSAMPLED_SAMPLE_RATE = COMMON_SAMPLE_RATE_LIBROSA/5
N_MELS = 80
N_MFCC = 12
HOP_LENGTH = 512
N_FFT = 2048

# FUNCTIONS
def pad_short_audio(signal):
    pad_ms = 1000

    silence = AudioSegment.silent(duration=pad_ms-len(signal)+1)
    signal = signal + silence # adding silence after the signal
    signal = signal[:COMMON_SAMPLE_RATE_LIBROSA]

    return signal

def extract_loudest_second(signal, sample_rate):
    results = {
        "slices": [],
        "rms": []
    }

    for slice_down in range(0, len(signal)-COMMON_SAMPLE_RATE_LIBROSA,
                           int(COMMON_SAMPLE_RATE_LIBROSA/15)): # steps of 1470 frames

```

```

slice_up = slice_down + COMMON_SAMPLE_RATE_LIBROSA if slice_down + COMMON_SAMPLE_RATE_LIBROSA <
len(signal) else len(signal)
results["slices"].append([slice_down, slice_up])
results["rms"].append(np.sum(rms(signal[slice_down:slice_up])))
slice_down, slice_up = results["slices"][[np.argmax(results["rms"])]]

out_filename = "./processed_audio.wav"
signal = signal[slice_down:slice_up]
sf.write(out_filename, signal, sample_rate)

return signal

def preprocess_audio_file(signal, sample_rate):
    if len(signal) < COMMON_SAMPLE_RATE_LIBROSA:
        signal = pad_short_audio(signal)
    else:
        signal = extract_loudest_second(signal, sample_rate)

    return signal

def CNN_melspecs(option):
    # Load model, initializing the interpreter
    #interpreter = tflite.Interpreter(model_path=str(MODEL_FILE[option])) # interpreter to make tests on laptop
    if option == 2:
        interpreter = edgetpu.make_interpreter(str(MODEL_FILE[option]), delegate=edgetpu.load_edgetpu_delegate())
    else:
        interpreter = edgetpu.make_interpreter(str(MODEL_FILE[option]))
    interpreter.allocate_tensors()

    input_details = interpreter.get_input_details()
    output_details = interpreter.get_output_details()
    input_size = input_details[0]["shape"][1]

    signal, sample_rate = load("./processed_audio.wav", sr=COMMON_SAMPLE_RATE_LIBROSA)
    if len(signal) < input_size:
        signal.resize(input_size)

    # Extract features
    melspec = melspectrogram(signal, sr=sample_rate, n_mels=N_MELS, hop_length=HOP_LENGTH, n_fft=N_FFT)
    melspec_dB = power_to_db(melspec, ref=np.max)
    melspec_dB = melspec_dB.T # transpose to keep consistency with what the CNN expects (# temporal segments, # Mel bands)

    # Check if the input type is quantized, then rescale input data to uint8
    if input_details[0]["dtype"] == np.uint8:
        input_scale, input_zero_point = input_details[0]["quantization"]
        melspec_dB = melspec_dB / input_scale + input_zero_point

    # Prepare data to be fed into the CNN
    melspec_dB = melspec_dB[np.newaxis, ..., np.newaxis].astype(input_details[0]["dtype"])

    # Make predictions
    interpreter.set_tensor(input_details[0]["index"], melspec_dB)
    interpreter.invoke()
    output = interpreter.get_tensor(output_details[0]["index"])

    return output

def CNN_MFCCs(option):
    # Load model, initializing the interpreter
    #interpreter = tflite.Interpreter(model_path=str(MODEL_FILE[option])) # interpreter to make tests on laptop
    if option == 5:
        interpreter = edgetpu.make_interpreter(str(MODEL_FILE[option]), delegate=edgetpu.load_edgetpu_delegate())

```

```

else:
    interpreter = edgetpu.make_interpreter(str(MODEL_FILE[option]))
    interpreter.allocate_tensors()

    input_details = interpreter.get_input_details()
    output_details = interpreter.get_output_details()
    input_size = input_details[0]["shape"][1]

    signal, _ = load("./processed_audio.wav", sr=COMMON_SAMPLE_RATE_LIBROSA)
    if len(signal) < input_size:
        signal.resize(input_size)

    # Extract features
    MFCCs = mfcc(signal, n_mfcc=N_MFCC, hop_length=HOP_LENGTH, n_fft=N_FFT)
    MFCCs = MFCCs.T # transpose to keep consistency with what the CNN expects (# temporal segments, # MFCCs)

    # Check if the input type is quantized, then rescale input data to uint8
    if input_details[0]["dtype"] == np.uint8:
        input_scale, input_zero_point = input_details[0]["quantization"]
        MFCCs = MFCCs / input_scale + input_zero_point

    # Prepare data to be fed into the CNN
    MFCCs = MFCCs[np.newaxis, ..., np.newaxis].astype(input_details[0]["dtype"])

    # Make predictions
    interpreter.set_tensor(input_details[0]["index"], MFCCs)
    interpreter.invoke()
    output = interpreter.get_tensor(output_details[0]["index"])

return output

def CNN_raws(option):
    # Load model, initializing the interpreter
    #interpreter = tflite.Interpreter(model_path=str(MODEL_FILE[option])) # interpreter to make tests on laptop
    if option == 8:
        interpreter = edgetpu.make_interpreter(str(MODEL_FILE[option]), delegate=edgetpu.load_edgetpu_delegate())
    else:
        interpreter = edgetpu.make_interpreter(str(MODEL_FILE[option]))
    interpreter.allocate_tensors()

    input_details = interpreter.get_input_details()
    output_details = interpreter.get_output_details()
    input_size = input_details[0]["shape"][1]

    signal, _ = load("./processed_audio.wav", sr=DOWNSAMPLED_SAMPLE_RATE)
    if len(signal) < input_size:
        signal.resize(input_size)

    # Extract features
    #Not required in this case

    # Check if the input type is quantized, then rescale input data to uint8
    if input_details[0]["dtype"] == np.uint8:
        input_scale, input_zero_point = input_details[0]["quantization"]
        signal = signal / input_scale + input_zero_point

    # Prepare data to be fed into the CNN
    signal = signal[np.newaxis, ..., np.newaxis].astype(input_details[0]["dtype"])

    # Make predictions
    interpreter.set_tensor(input_details[0]["index"], signal)
    interpreter.invoke()
    output = interpreter.get_tensor(output_details[0]["index"])

return output

```

```

def TL_spanish(option):
    # Load model, initializing the interpreter
    #interpreter = tflite.Interpreter(model_path=str(MODEL_FILE[option]))
    interpreter = edgetpu.make_interpreter(str(MODEL_FILE[option]))
    interpreter.allocate_tensors()

    input_details = interpreter.get_input_details()
    output_details = interpreter.get_output_details()
    input_size = input_details[0]["shape"][1]

    signal, _ = load("./processed_audio.wav", sr=INPUT_SAMPLE_RATE)
    if len(signal) < input_size:
        signal.resize(input_size)

    # Extract features
    #Not required in this case

    # Check if the input type is quantized, then rescale input data to uint8
    if input_details[0]["dtype"] == np.uint8:
        input_scale, input_zero_point = input_details[0]["quantization"]
        signal = signal / input_scale + input_zero_point

    signal = np.expand_dims(signal[:input_size], axis=0).astype(input_details[0]["dtype"])

    # Make predictions
    interpreter.set_tensor(input_details[0]["index"], signal)
    interpreter.invoke()
    output = interpreter.get_tensor(output_details[0]["index"])

    return output

```

```

# MAIN
if __name__ == "__main__":
    msg = """Model options to choose:
\l1 - CNN model using Mel spectrograms
\l2 - CNN quantized model using Mel spectrograms
\l3 - CNN quantized model using Mel spectrograms, prepared for TPU execution
\l4 - CNN model using MFCCs
\l5 - CNN quantized model using MFCCs
\l6 - CNN quantized model using MFCCs, prepared for TPU execution
\l7 - CNN model using raw signals
\l8 - CNN quantized model using raw signals
\l9 - CNN quantized model using raw signals, prepared for TPU execution
\l10 - Spanish model with TL
\l11 - Spanish quantized model with TL\n"""
    print(msg)

    options = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    while True:
        option = None
        while option not in options:
            option = input("Select a model (send 'h' to see model options or press 'Ctrl+C' to stop): ")

        if str(option) in ["h", "H", "help"]:
            print(msg)

        try:
            option = int(option)-1
        except:
            pass

```

```

if option not in options and str(option) not in ["h", "h", "help"]:
    print("Invalid option. Valid options are: [1,2,3,4,5,6,7,8,9,10,11]\n")

aux = MODEL_FILE[option].split("/")[-1]
print(f"Using {aux} model")

tic1 = time()

# Get labels
with open(LABELS_FILE[0 if option not in [9,10] else 1], "r") as f:
    labels = f.readlines()
    labels = [label.strip() for label in labels]

# Load random audio
filename = random.choice(os.listdir(DATA_PATH[0 if option not in [9,10] else 1]))
file = os.path.join(DATA_PATH[0 if option not in [9,10] else 1], filename).replace("\\", "/")
signal, sample_rate = load(file, sr=COMMON_SAMPLE_RATE_LIBROSA)

# Preprocess audio -> transform to standard form
signal = preprocess_audio_file(signal, sample_rate)

tic2 = time()

# Prepare and run model
if option in [0,1,2]:
    output = CNN_melspecs(option)
elif option in [3,4,5]:
    output = CNN_MFCCs(option)
elif option in [6,7,8]:
    output = CNN_raws(option)
elif option in [9,10]:
    output = TL_spanish(option)

tac2 = time()

top_index = np.argmax(output[0])
label = labels[top_index]
score = output[0][top_index]

tac1 = time()

# Display prediction, ground truth and times
print("---prediction---")
print("Class:", label)
print("Score:", score, "\n")
print("---truth---")
print(file.split("/")[-1][:-4], "\n")
print("---times---")
print(f"Model executed in {np.round(tac2-tic2, 5)} seconds")
print(f"Whole process done in {np.round(tac1-tic1, 5)} seconds\n")

```